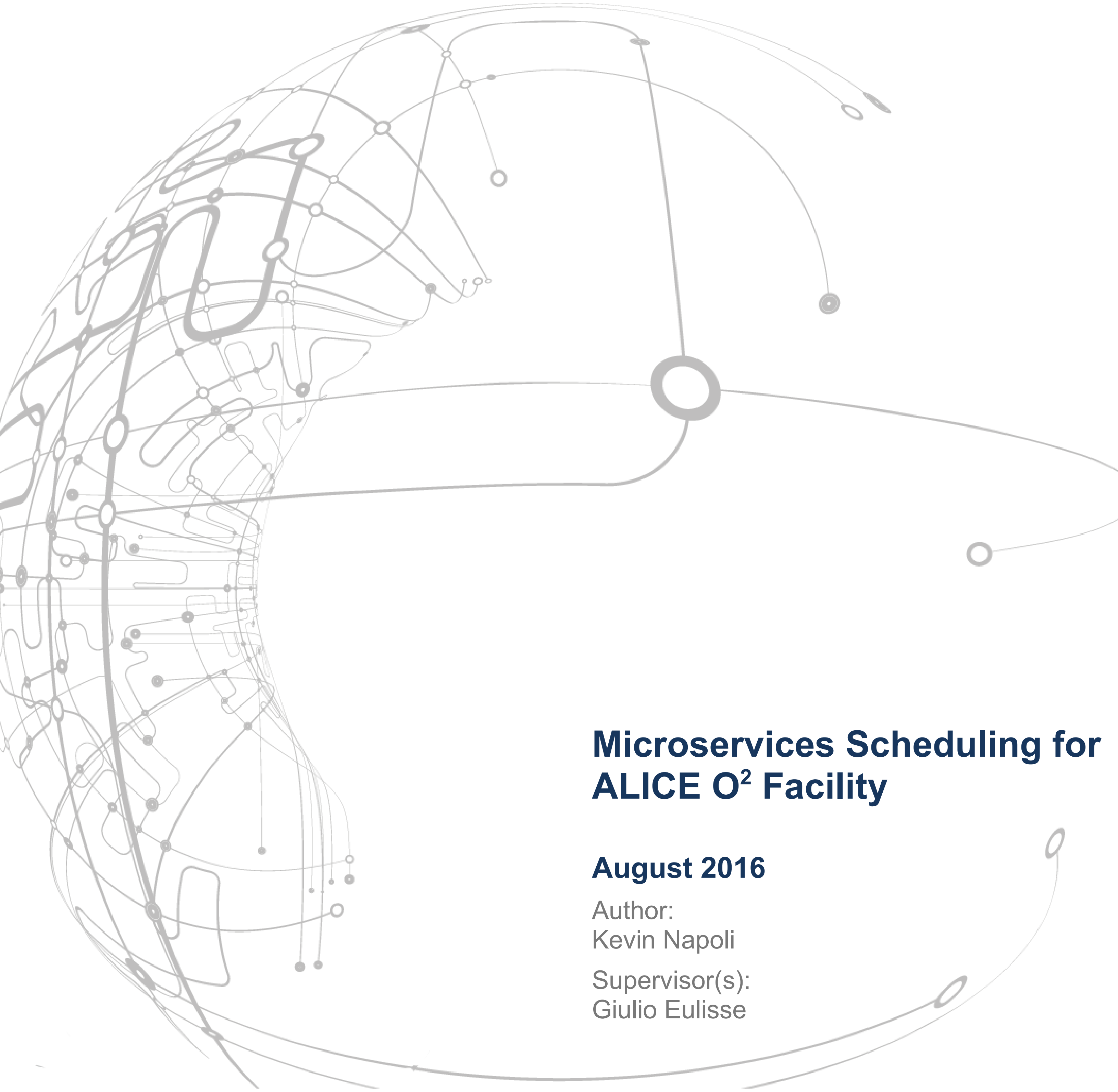


Microservices Scheduling for ALICE O^2 Facility

August 2016

Author:
Kevin Napoli

Supervisor(s):
Giulio Eulisse

CERN openlab Summer Student Report 2016

Project Specification

This project seeks to research ways to deploy DDS (Dynamic Deployment System) jobs across cluster nodes using Apache Mesos. It is subdivided in three subtasks

1. DDS Mesos Plugin – This task involves writing a plugin for DDS such that it can deploy agents on cluster node by using Apache Mesos. The DDS plugin interfaces with Mesos using the Mesos Framework API.

2. Trying out Mantl – This task involves deploying Mantl.io on a test cluster, try Mantl.io and analyse the complexities of deploying DDS agents using the Mantl GUI interface. A summary of the advantages and disadvantages of deploying DDS agents through Mantl is to be reported.

3. Automatic Network Topology Detection – This task involves researching ways to automatically infer the underlying network switch topology in Layer 2. This means that protocols for network discovery such as SNMP cannot be used. After this step, one idea is to use the discovered topology in Mesos.

As a result, one will be able to submit DDS jobs on a Mesos controlled cluster through the usual dds-submit interface/procedure which is currently in use.

Abstract

As academic and industrial computational needs rise, organisations employ the use of computer clusters in order to keep up with these computational needs and CERN is no exception. Several distributed software frameworks exist, each of which solves a particular problem. However, these frameworks assume total control of cluster resources making it difficult to run them concurrently on the same cluster. Additionally, it is apparent that there is no scheduling algorithm or policy that satisfies all types of jobs. Apache Mesos, a meta-scheduler for distributed systems, tries to mitigate this problem without resorting to statically partitioning a cluster. In this work we have explored ways of integrating the Dynamic Deployment System (DDS) at CERN with Mesos. As a result, DDS jobs can be run on a Mesos governed cluster.

Table of Contents

1 Introduction....5
2 Dynamic Deployment System....6
2.1 RMS Plugins and DDS Agents....6
2.2 Topology Example....7
2.3 Usage....8
3 Apache Mesos....8
3.1 Framework Guidelines....10
3.2 Scalability....10
4 DDS-Mesos Plugin....11
    4.1 Design and Implementation....11
        4.1.1 DDS Plugin....11
        4.1.2 Mesos Framework....13
    4.2 Usage & Demo....15
5 Mantl....16
5.1 Evaluation....17
5.2 Integrating other Services....19
5.3 Experience....20
6 Network Discovery....20
6.1 Switches and Ethernet....21
6.2 Switch Topology Detection....21
6.3 Results....25
6.4 Limitations....26
6.5 Alternatives....27
7 Conclusion and Future Work....27
8 References....28

1 Introduction

In computer science, scheduling is a well known optimisation problem where a number of computer jobs residing in a queue are assigned to specific resources owned by a system. This task is carried out by a scheduler. Schedulers are used in a wide range of applications, from embedded devices [1] to distributed computers [2]. One example is a process scheduler where the aim is to maximise resource efficiency and minimise the time that each process takes to execute [3]. The properties of a scheduler vary depending on the application of the scheduler and on the scheduling policy. Moreover, scheduling pertains to the class of NP-Complete problems, and as such, only approximate solutions can be produced.

ALICE (A Large Ion Collider Experiment), one of a number of experiments at CERN's Large Hadron Collider, features a detector designed to investigate nucleus-nucleus interactions at high energy densities. The objective of the experiment is to recreate and study the quark gluon plasma, believed to have been the state of the Universe up to a few milliseconds after the Big Bang [4].

In order to allow for maintenance, the LHC is periodically shutdown and the timeline of the ALICE experiment is subdivided into runs and long-shutdowns. For Run 2 of the experiment, which is ongoing, dedicated clusters are responsible for the online tasks of data acquisition and storage while offline analysis is delegated to the Worldwide LHC Computing Grid (WLCG). In order to meet the elevated data requirements of Runs 3 and 4 following Long Shutdown 2, ALICE computing will upgrade to O^2 [5, 6], a combined online and offline high-throughput system that incorporates heterogeneous computing platforms. With the O^2 software framework, a common ALICE computing facility will share data acquisition and processing responsibilities such as detector read-out, event building, data recording, detector calibration, detector reconstruction and physics simulation and analysis.

O2 will use the ALFA software framework [7] to provide data-flow concurrency with message queues to a cluster job. The O2 computing infrastructure will also be required to dynamically partition the cluster among a broad variety of online and offline jobs with a high degree of flexibility and resilience. Dynamic Deployment System (DDS) [8] is ALFA's own component for dynamically distributing processes across a cluster. DDS is able to deploy processes on a cluster using any resource management system (RMS) in a predefined topology dictated manually by the user through an XML file. Tasks within a topology can have properties and this allows users to, for example, create dependences between tasks [9].

2 Dynamic Deployment System

The Dynamic Deployment System (DDS) is a software framework developed by GSI Helmholtz Centre for Heavy Ion Research (GSI) that simplifies the process of deploying tasks on a cluster [10]. It provides a common API to communicate with an arbitrary Resource Management Systems (RMS). Tasks are defined in an XML file most commonly referred to as the **topology** file. The topology file contains information about each task. Users can define properties, task dependencies, hosts where the tasks can run and more.

The process of running DDS is simple and consists mainly of three steps.

1. Users start a DDS server (also known as Commander) on their machine

2. Deploy DDS agents on the required worker nodes through the use of an RMS

3. Enable and run a DDS topology

The outlined steps show how simple it is to run jobs on a cluster using DDS although there are some limitations. The users' machine must be able to accept incoming connections to reach the DDS server that they spawned in the first place. The DDS server provides task coordination for the worker nodes and also allows the user to monitor the status of jobs and running nodes. Each DDS agent connects to the server in order to exchange information. This also means that worker nodes must allow outgoing connections.

2.1 RMS Plugins and DDS Agents

In order for DDS to be able to communicate with an RMS, a suitable DDS plugin must be written for it. Therefore, DDS supports a plugin architecture that, in theory, allows communication with a large number of RMSes.

DDS uses an RMS plugin specifically for one task; to deploy the **DDS worker package** on a number of nodes on the cluster. The worker package is named **DDSWorker.sh**. In short, it is a large package containing all the dependencies and consists of shared objects that will be loaded at runtime and scripts required to be able to run the **DDS agent**.

Currently DDS supports three plugins as tabulated below:

Name	**Description**
Localhost	This plugin is used to deploy agents on the same machine as the one used to start DDS
SSH	This plugin is used to deploy agents on remote machines/nodes than the one used to start DDS. It assumes a readily configured public/private key between this machine and remote nodes. Using SSH, it will then deploy the DDS worker package and start execution of the package on each node.
Slurm	This plugin is used to deploy agents on a cluster that is running Slurm [11]. It is designed to work on small and large Linux clusters and has a set of tools to be able to monitor work. Slurm is also pluggable and different scheduling algorithms can be implemented to expand its functionality.

Once the worker package is deployed and run on each node, the RMS is no longer needed. Each DDS agent connects to the server that is started on the user's machine and control to each node is thus acquired. Tasks would now be able to be deployed freely without requiring further tools/software.

2.2 Topology Example

A DDS topology defines the tasks to be run on the agents that were setup previously. To demonstrate how a simple topology file is structured, an example involving sleep tasks is shown below (sleepTopology.xml):

```
<topology id="sleepTopology">

   <var id="appNameVar" value="/bin/sleep" />

   <decltask id="task1">
      <exe reachable="true">${appNameVar} 120</exe>
   </decltask>

   <decltask id="task2">
      <exe reachable="true">${appNameVar} 121</exe>
   </decltask>

   <!-- Definition of the topology itself -->
   <main id="main">
      <task>task1</task>
      <task>task2</task>
```

```
    </main>

</topology>
```

This file defines two tasks: *task1* and *task2*. The task has an **exe** node with a **reachable** attribute. The command to be run is inside this node and the reachable attributes tells DDS that the executable is already located on the worker node. *task1* will run '`sleep 120`' while *task2* will run '`sleep 121`'. The *appNameVar* variable defines the path of the executable to run. Properties can also be defined under each task and can be used for communication between worker nodes.

2.3 Usage

DDS supplies various commands to be able to execute the various stages mentioned above. In order to deploy the sleep topology defined above, the following steps are required.

1. Start DDS server – `dds-server start`
2. Deploy the agents using an arbitrary rms (in this case localhost) – `dds-submit -rms localhost -n 2`
3. [Optional] Verify that the agents are actually online using – `dds-info -l`
4. Set the topology – `dds-topology -set sleepTopology.xml`
5. Activate the topology – `dds-toplogy -activate`

One should be able to verify that the sleeping tasks are indeed being executed on each node.

3 Apache Mesos

According to Guatam et al., it is apparent that there is no scheduling algorithm or policy that satisfies all types of jobs [12]. In fact, many computational problems in the industry nowadays require special software applications. For instance, Hadoop [13] (derived by MapReduce [14]) is an application that aids with the computation of Big Data problems while Google Pregel [15] is an application that is especially good at processing graph data. While there are ways to perform graph data processing on Hadoop, the performance is low compared to computing it with Pregel. Additionally, these applications cannot easily coexist and may take control of the resources of the whole cluster.

One solution to allow two or more applications to run together is to statically partition the cluster thereby isolating one application from another. However, this option is not very optimal. At any point in time, an application might require the use of more resources or it might not even use all the resources allocated to it with static partitioning.

To solve this problem, Hindman et al. developed Apache Mesos [16], a granular and thin resource sharing layer providing a common interface to frameworks (applications are called frameworks in the terminology of Meses) for accessing cluster resources. Instead of implementing a complex centralised scheduler that caters for all framework requirements, Mesos takes a different approach. It implements a resource offer mechanism, an abstraction which delegates scheduling control to the frameworks. This mechanism requires two components for each framework communicating with Mesos; A framework scheduler and an executor. The framework scheduler is responsible for negotiating resources offered by Mesos and allocating these resources to jobs that need to run. It can allocate resources using any scheduling algorithm and policy that it requires. The framework scheduler, as well as the executor, can be developed in Java, C++ or other languages by using the respective API that Mesos offers. The executor is a separate component responsible for running the job selected by the framework scheduler.

Therefore, Mesos enables a two-level scheduling scheme and allows multiple frameworks to run on the same cluster. At one level, Mesos uses a pluggable resource allocation module to decide what resources to offer to which framework. Two organisational policies are implemented, one that is fair and another that is priority based. At the second level, the framework scheduler decides how to allocate the jobs it needs to schedule on its executor.

Mesos is fair; it guarantees a minimum resource allocation to every framework that is running on the cluster. It also penalises frameworks that are not elastic and that latch onto resources beyond their minimum allocation for a long time by revoking their tasks in order to obtain resources back. This can happen if a new framework starts running while other frameworks have already acquired all the cluster resources. The new framework would not be able to reach its guaranteed allocation if other frameworks do not scale down. Before revoking any tasks, Mesos gives a grace period to the framework so that it can clean up prior to forcefully killing its tasks.

In some cases, Mesos could offer resources that are incompatible with a framework. This results in a framework rejecting the resources every time they are offered to it. It could also reduce the efficiency of the cluster and waste network bandwidth; these resources are not being used while Mesos is offering resources to the wrong frameworks. To tackle this issue, Mesos features filters. Frameworks can specify which resources they can accept at registration time. Thus, Mesos will never offer resources to a framework that cannot ever utilise them. This increases the scalability of resource allocations.

The Mesos master is fault-tolerant as it has been designed to be soft state. This means that if the master fails, its internal state can be regenerated by the next elected master node. The information to reconstruct this state is held by the slaves and the framework schedulers.

3.1 Framework Guidelines

Hindman et al. give general suggestions which they refer to as framework incentives. These incentives are aimed to promote job response times. A framework is incentivised to use short tasks, scale elastically and to reject unknown resources immediately. Having short tasks minimises the impact of work lost due to failures or revocations. Using an elastic framework, as opposed to a rigid one, maximises resource utilisation allowing jobs to start and finish earlier. Rigid frameworks generally do not start their jobs until all required resources are acquired and do not release resources until all jobs finish. Finally, frameworks should never accept resources that they cannot utilise; this would waste them as other frameworks that can use them would not be able to.

With this in mind, the authors implement a Hadoop framework for Mesos using Delay Scheduling [17]. Delay Scheduling is a scheduling algorithm aimed at increasing *fairness* and *data locality* – two properties that Gautam et al. term to be important in MapReduce. To attain this, Zaharia et al. show that to achieve fairness, waiting for tasks to finish is better than killing running tasks in order for new tasks to run instantly. They find that waiting does not really affect job response time assuming that jobs are longer than the average task length and that the cluster is being utilised by many users. Particularly, job response time will not be notably affected if the cluster is running many jobs, small jobs or long jobs.

3.2 Scalability

Results have shown that using Delay Scheduling in the Hadoop framework raised locality above 90% as opposed to 50%. This was achieved as a result of Mesos' resource offer mechanism where in conjunction with the Hadoop framework, most of the tasks were run on nodes that contained local data essential for the task.

Hindman et al. also evaluate a mix of Hadoop, Spark and Torque jobs running concurrently on a 96 node cluster. They run two instances of Hadoop, one consisting of small to large workloads and another consisting of typical Facebook workloads, an iterative learning job on Spark and a ray tracing process on Torque. Each instance is first evaluated in static partitioning assigning 24 (one fourth of the cluster) nodes per instance. Running the four instances concurrently on Mesos yielded positive results. Mesos enabled each instance of a framework to scale beyond 24 nodes when other frameworks had low resource usage and it also managed to reallocate resources fast. All instances except for Torque took less time to finish than when they were statically partitioned. Hadoop with typical Facebook workloads acquired a speed-up of 2.1 while Torque ran slightly slower with a speed-up of 0.96.

4 DDS-Mesos Plugin

The first part of this project consisted of developing a plugin for DDS with a suitable Mesos framework. The problem is that DDS, as any other framework, takes exclusive control of cluster resources. Running another framework on the same cluster as the one running DDS creates conflict unless static partitioning is set up. The first part of this project deals with allowing the DDS framework to run alongside other frameworks on the same cluster and allowing other frameworks to scale up while DDS is inactive or using few resources, thus maximising cluster utilisation. Apache Mesos is designed to solve these types of issues and has thus been chosen as the ideal candidate.

4.1 Design and Implementation

The solution has been subdivided into two loosely coupled solutions namely DDS plugin and Mesos framework. In short, the DDS plugin interacts with the user through `dds-submit` and communicates with a REST service implemented inside the Mesos framework. The Mesos framework receives this REST submission, parses the data and deploys the agents as requested. The agents can also be run in Docker containers to guarantee isolation and worker package binary compatibility. A user who is running DDS on a different Linux distribution than the Mesos cluster can use this feature to run the worker packages without problems.

A high level architectural view is shown below:

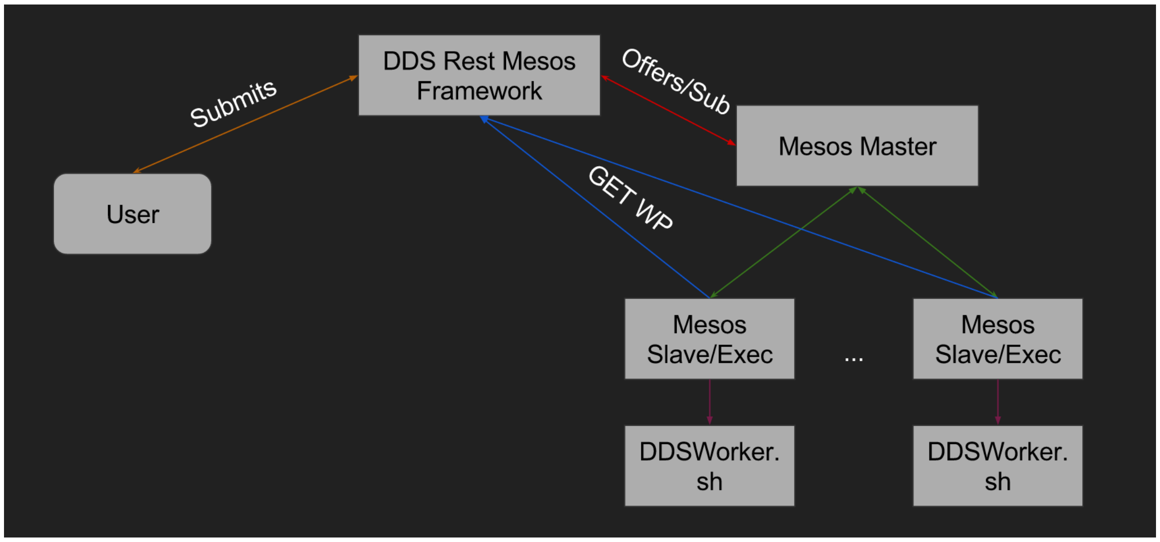


Figure 1. DDS-Mesos High Level Architecture

4.1.1 DDS Plugin

Whenever the user performs the `dds-submit` command in the terminal, an appropriate RMS plugin is chosen. A configuration file can also be included using the **-c** switch when invoking `dds-submit` and is mandatory for this plugin. In order to use Mesos as an

RMS to deploy DDS agents, a Mesos plugin for DDS has been written. This plugin is an executable written in C++11 that communicates with DDS through a form of interprocess communicaton. DDS provides appropriate libraries that contain registration functions for the plugin to register callbacks for appropriate DDS events.

The functions that the DDS library provides in the **dds_intercom.h** header file, specifically in the **CRMSPluginProtocol** class, are tabulated below:

Function Prototype	**Description**
`void onSubmit(signalSubmit_t::slot_function_type _subscriber);`	Subscribe for submit notifications. The registered callback function is called when the user invokes dds-submit. The parameter passed to the callback functions (type SSubmit) includes information about the path to the worker package, DDS submission identifier and configuration file path/number of instances.
`void onMessage(signalMessage_t::slot_function_type _subscriber);`	Subscribe for message notifications.
`void sendMessage(EMsgSeverity _severity, const std::string& _msg);`	Send message to the DDS commander (server). Can be used to send log/debug information to the commander.
`void start(bool _block = true);`	Send initial request to the commander and start listening for notifications. Under normal conditions, function blocks until stop is called.
`void stop();`	Stop waiting. Will make start return.

A configuration file is required for the plugin to run. The format of each line of the configuration file should be as follows:

1. Mesos Master IP:Port
2. Number of agents to deploy
3. Docker image to use for the agents to run in
4. Folder inside the docker image where to copy the DDS Worker Package
5. Number of CPU Cores to utilise for each Agent
6. The size of memory to use for each agent (in Megabytes)
7. The IP:Port of the Rest service API

The DDS plugin is detached from any Mesos code. Instead of communicating directly with Mesos, it calls a centralised REST service implemented in the DDS Mesos framework. Therefore, the plugin translates the information coming from the onSubmit event and from the configuration file, generates a JSON object and through an HTTP POST calls a REST method on the server (endpoint http://<ip:port>/dds-submit). The format of the JSON object generated is show below:

```
Listing – Json DDS Submission
{
    DDSSubmissionId: <id>,
    WorkerPackageName: <worker-package-file-name>,
    WorkerPackageData: <worker-package-file-in-base64>,
    Resources: {
            NumAgents: <number-of-agents>,
            CpusPerTask: <number-of-cpus-per-agent>,
            MemorySizePerTask: <memory-size-per-agent>
    },
    Docker: {
            ImageName: <docker-image-name>,
            TemporaryDirectoryName: <working-directory-in-container>
    }
}
```

On success, the server replies with HTTP 200 and the following JSON:

```
{
    Id: <Rest-Id-Submission>
}
```

The DDS plugin will print this identifier in the logs and indicate a successful submission. From this point onwards, the DDS plugin polls the /status endpoint every 500msec until all pending agents are running. As soon as `dds-submit` returns, all agents are submitted. The DDS agents should appear on the Mesos UI. One can check whether the agents are online by invoking dds-info -l.

On error, an HTTP error code is returned together with a text response of the error. The error can also be found in the relevant log. Note that currently, only the HTTP code 400 (Bad Request) is returned for any error that might occur.

4.1.2 Mesos Framework

The second part involves the implementation of a REST service and Mesos Framework. A REST service has been exposed using the Microsoft C++ Rest SDK (Casablanca). Currently, this service has three endpoints:

Endpoint	**Type**	**Description**

/dds-submit	POST	This endpoint is used to submit a number of DDS agents on the cluster. A JSON object is expected as input and its format is described in Listing – Json DDS Submission
/dds-work-package?id=<size_t>	GET	This endpoint is used to download the worker package submitted using the identifier returned by the dds-submit endpoint.
/status?id=<id>	GET	Currently returns the number of submissions performed on the server. If a submission identifier is provided, it returns the number of agents that are still waiting for resource offers and thus still need to be deployed on the cluster. The result is a JSON object of the form: `{ NumSubmissions: <size_t>, PendingAgents: <size_t> }`

When a DDS client makes a submission through the `/dds-submit` endpoint, a list of tasks corresponding to each DDS agent is created on the server. A Mesos task generally defines a number of properties. These include which Mesos executor is to execute, the command information, any files required and resource allocation. The Mesos framework provides the following information to the Mesos master:

- `ContainerInfo`: In case a docker image is specified, this is the image inside which the custom executor will run
- `CommandInfo_URI`: Worker package URL set as executable, non-cacheable
- `CommandInfo`: A command that creates a working directory and copies the worker package inside it
- The final command that starts to execute the working package inside the newly created working directory

This list is then passed onto the Mesos framework’s FIFO queue.

As cluster resources become free, the Mesos master will make offers to the Mesos framework. If these resources are adequate for the DDS agent tasks, the Mesos framework will accept them and submit the task information to the Master.

Consequently, the master delegates each task to the corresponding slave. The slave starts the default Mesos executor (a custom executor was not required) which according to the task information will perform the following:

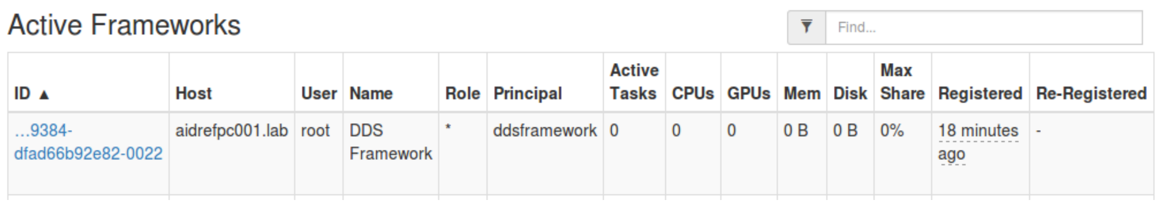
- Load the required Docker image and start the container if necessary.
- Perform a GET request to download the DDS worker package by calling the `/dds-worker-package` REST endpoint
- Execute the command requested in the `CommandInfo` object specified before

4.2 Usage & Demo

Instructions on how to use the DDS plugin and the DDS Mesos/Rest service can be found in the README.md file of the mesos-dds git repository [18]. When the DDS Mesos/Rest service is successfully initialised, it should be listed in the Mesos UI. In this example, the command:

```
dds-mesos-server        --master=10.162.61.10:5050        -
resthost=10.162.61.10:1234
```

was executed and as a result the following framework entry appears in the Mesos UI:



Active Frameworks

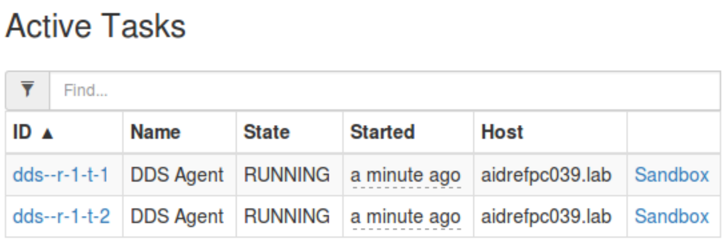
ID ▲	Host	User	Name	Role	Principal	Active Tasks	CPUs	GPUs	Mem	Disk	Max Share	Registered	Re-Registered
...9384-dfad66b92e82-0022	aidrefpc001.lab	root	DDS Framework	*	ddsframework	0	0	0	0 B	0 B	0%	18 minutes ago	-

Figure 2. DDS Framework in the Mesos UI

On the client side, the DDS Commander was initialised and a submission performed using the DDS plugin for Mesos. The following commands were executed:

```
dds-server start -s && dds-submit -r mesos -c conf.txt
```

The configuration file contained the required information to submit two agents on the cluster (see git repository for more details). As a result, the following tasks are now visible in the Mesos UI:

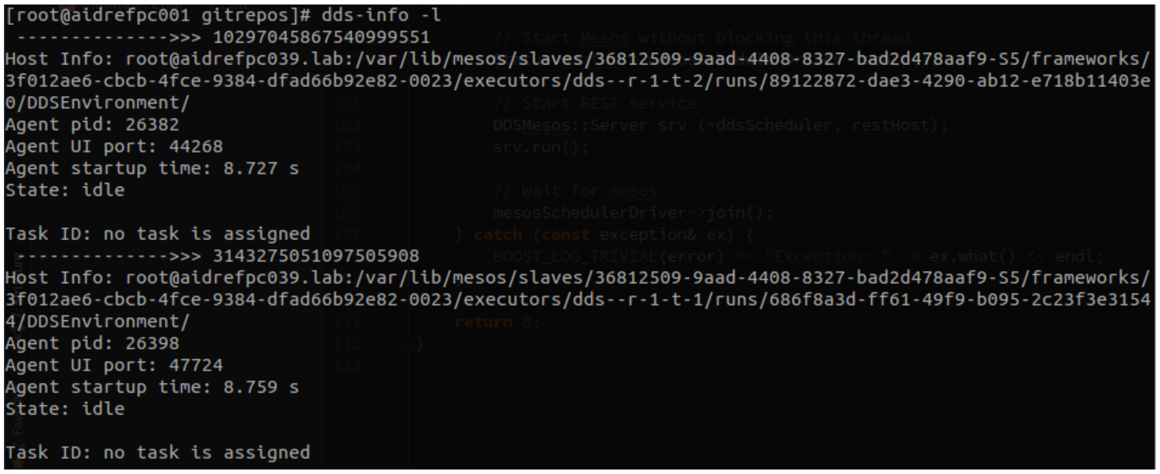


Active Tasks

ID ▲	Name	State	Started	Host	
dds--r-1-t-1	DDS Agent	RUNNING	a minute ago	aidrefpc039.lab	Sandbox
dds--r-1-t-2	DDS Agent	RUNNING	a minute ago	aidrefpc039.lab	Sandbox

Figure 3. DDS Agents running in the Mesos UI

Additionally, `dds-info -l` yields:



```
[root@aidrefpc001 gitrepos]# dds-info -l
 --------------->>> 10297045867540999551
Host Info: root@aidrefpc039.lab:/var/lib/mesos/slaves/36812509-9aad-4408-8327-bad2d478aaf9-S5/frameworks/
3f012ae6-cbcb-4fce-9384-dfad66b92e82-0023/executors/dds--r-1-t-2/runs/89122872-dae3-4290-ab12-e718b11403e
0/DDSEnvironment/
Agent pid: 26382
Agent UI port: 44268
Agent startup time: 8.727 s
State: idle

Task ID: no task is assigned
 --------------->>> 3143275051097505908
Host Info: root@aidrefpc039.lab:/var/lib/mesos/slaves/36812509-9aad-4408-8327-bad2d478aaf9-S5/frameworks/
3f012ae6-cbcb-4fce-9384-dfad66b92e82-0023/executors/dds--r-1-t-1/runs/686f8a3d-ff61-49f9-b095-2c23f3e3154
4/DDSEnvironment/
Agent pid: 26398
Agent UI port: 47724
Agent startup time: 8.759 s
State: idle

Task ID: no task is assigned
```

Figure 4. dds-info command showing running agents

5 Mantl

A very important trend in service-based distributed systems is the adoption of microservices, a more granular, scalable and fault tolerant approach to the monolithic counterpart. For all their advantages, microservice architectures introduce problems of their own, such as more complex deployment, orchestration and management processes; Mantl.io by CISCO is an open source end-to-end solution for deploying and managing a microservices infrastructure. It consists of a curated set of software packages including Apache Mesos and recently Kubernetes. Mantl provisions its infrastructure via Terraform [22] and installs itself using Ansible [26]. A set of Ansible addons are also provided to deploy additional Mesos frameworks.

It would therefore be interesting to see if DDS can be easily included in the Mantl distribution for ease of deployment. We therefore decided to evaluate Mantl.

5.1 Evaluation

The first approach taken was that of setting up Mantl on a local machine using Vagrant [27]. However, this process was not successful. One issue was that the virtual machines setup by Vagrant would not boot successfully during some parts of the process. This procedure has then been discontinued and another approach was taken.

The other approach involved setting up Mantl on the OpenStack cloud running at CERN. The first step that was taken involved creating the OpenStack instances using the OpenStack Terraform provider. The sample Terraform script **terraform/openstack-modules.sample.tf** was copied and changed according to the instructions given in the documentation of Mantl. This file configures the instance names to use, the number of nodes to use for Mantl and more. Due to limited resources of the test account we used, Kubernetes worker nodes were disabled in the aforementioned Terraform configuration.

At first running the script failed due to the setup of the OpenStack cloud at CERN. Specifically, **Floating IPs** and **Neutron ports** are disabled due to network performance issues. Therefore, the following changes have been made in order to setup the required Mantl instances correctly.

- **mantl/roles/calico/tasks/openstack.yml** – Commented out the role with the name 'unlock neutron ports to allow calico traffic'
- **mantl/terraform/openstack/instance/main.tf** – Commented out the following lines:
  - `variable floating_ips { default = "" }`
  - `variable network_uuid {}`
  - `floating_ip = "${ element(split(",", var.floating_ips), count.index) }"`
  - `network = { uuid = "${var.network_uuid}" }`

After these changes are made, one can successfully run `terraform plan` and `terraform apply`. After these two commands are run, the required OpenStack instances will be setup.

The second step involved configuring the newly created node instances using Ansible. Mantl provides ready made scripts for this as well. However, simply following the instructions in the documentation is not enough. The reason for this is that the CentOS images at CERN in OpenStack come pre-set with a yum plugin called **protectbase**. This plugin hides the Cisco repository which is setup by the Ansible scripts and therefore the setup will fail. To solve this, protectbase has been disabled on all nodes. This was done manually by editing the **/etc/yum/pluginconf.d/protectbase.conf** and changing **enabled = 0** to **enabled = 1**. After this adjustment was made, the setup continued as follows:

- Run security setup script – `security-setup`
- Upgrade packages on nodes - `ansible-playbook playbooks/upgrade-packages.yml`
- Install Mantl and all dependencies - `ansible-playbook sample.yml -e @security.yml`

Following this, the setup completed successfully. As a result, Mantl and some of its default components were installed on the OpenStack nodes that were setup previously after applying Terraform. The Mantl UI web page opened successfully as it can be seen in Figure 5. Figure 5 shows that the applications are running normally on the cluster.

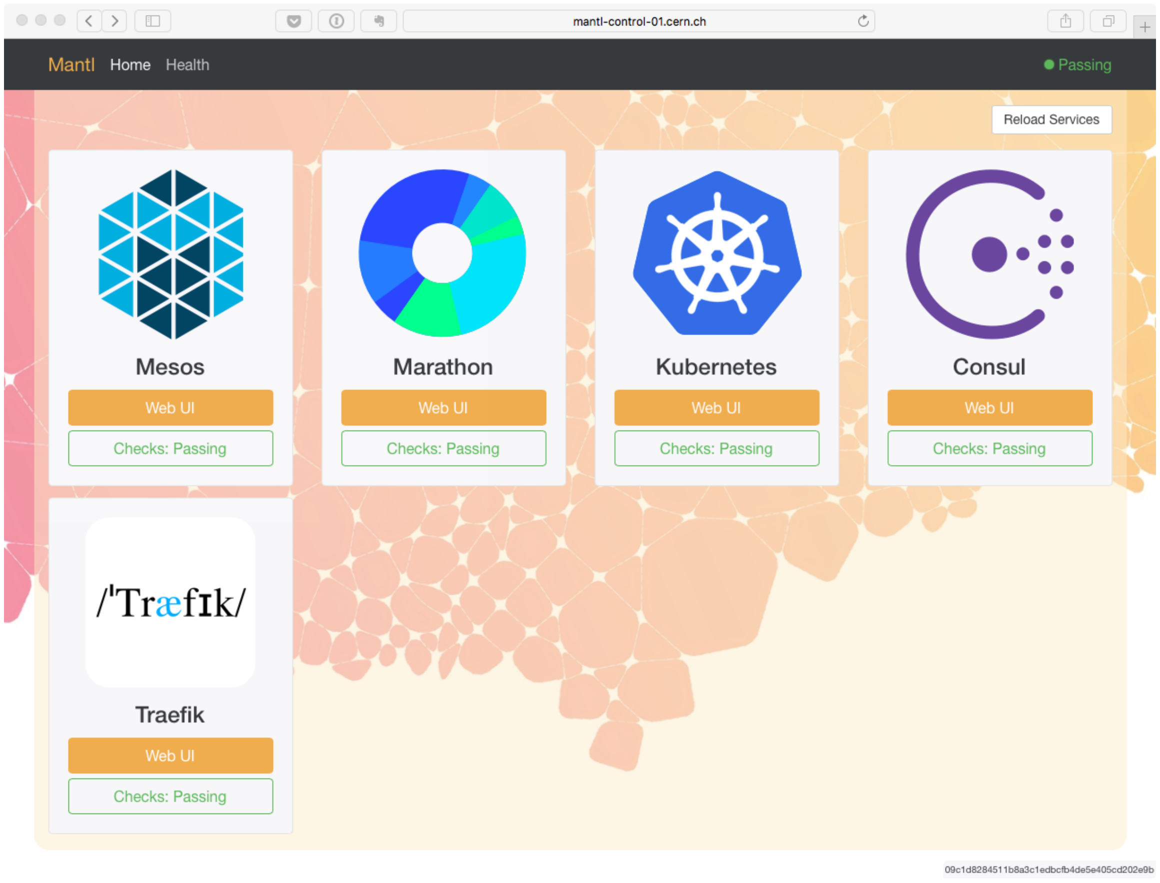
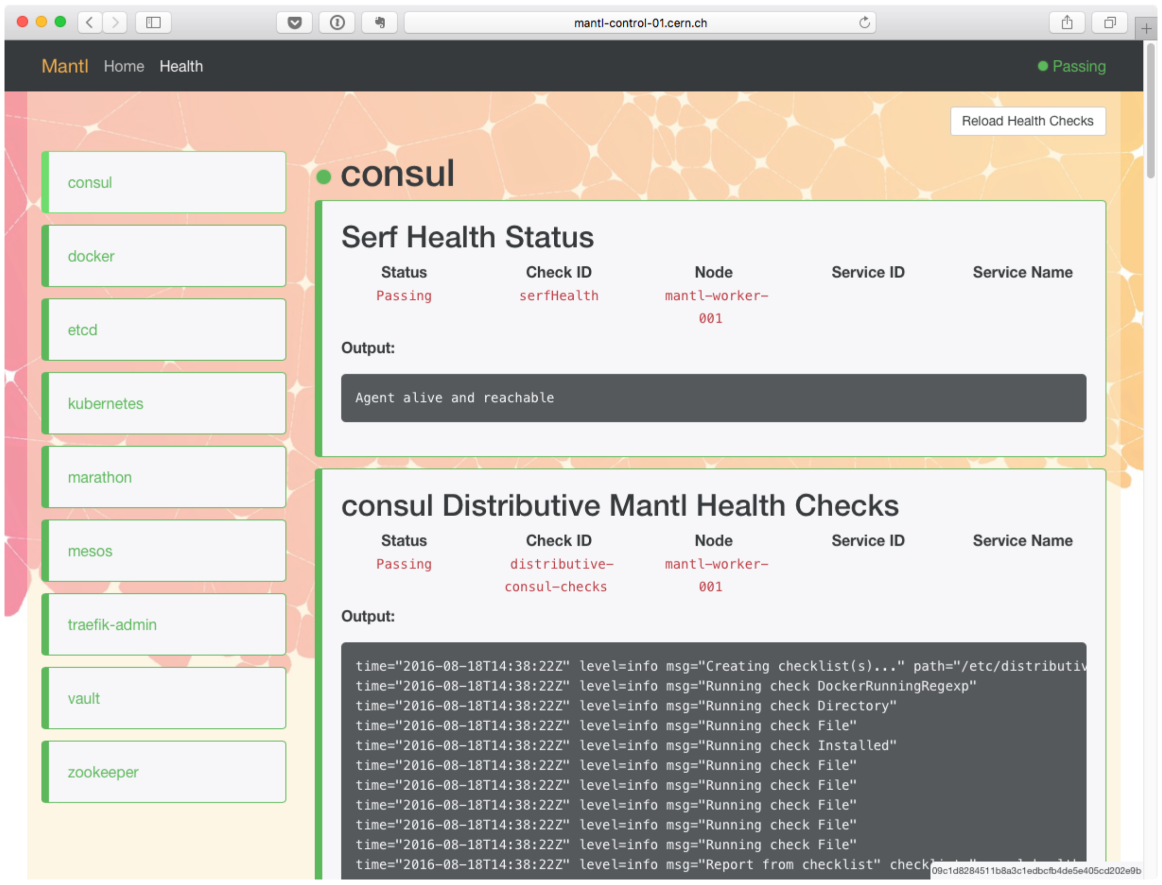


Figure 5. Mantl Main Page

If one wants to check the health of an application/service, one can click on the Health link at the top of the page. For example, clicking on Health and then on consul gives in-depth detail about it. Figure 6 shows the result of this:



consul

Serf Health Status

Status	Check ID	Node	Service ID	Service Name
Passing	serfHealth	mantl-worker-001		

Output:

```
Agent alive and reachable
```

consul Distributive Mantl Health Checks

Status	Check ID	Node	Service ID	Service Name
Passing	distributive-consul-checks	mantl-worker-001		

Output:

```
time="2016-08-18T14:38:22Z" level=info msg="Creating checklist(s)..." path="/etc/distributiv
time="2016-08-18T14:38:22Z" level=info msg="Running check DockerRunningRegexp"
time="2016-08-18T14:38:22Z" level=info msg="Running check Directory"
time="2016-08-18T14:38:22Z" level=info msg="Running check File"
time="2016-08-18T14:38:22Z" level=info msg="Running check Installed"
time="2016-08-18T14:38:22Z" level=info msg="Running check File"
time="2016-08-18T14:38:22Z" level=info msg="Running check File"
time="2016-08-18T14:38:22Z" level=info msg="Running check File"
time="2016-08-18T14:38:22Z" level=info msg="Running check File"
time="2016-08-18T14:38:22Z" level=info msg="Running check File"
time="2016-08-18T14:38:22Z" level=info msg="Report from checklist" check
```

09c1d8284511b8a3c1edbcfb4de5e405cd202e9b

Figure 6. consul health details

5.2 Integrating other Services

Once Mantl was up and running, the next step taken was that of adding other services to the cloud. In order to do this, we used the ready-made Ansible scripts by Mantl. Specifically, we tried installing Kafka and Elastic search using the commands `ansible-playbook addons/kafka.yml -e @security.yml` and `ansible-playbook addons/elk.yml -e @security.yml` respectively.

Ansible's script reported the elastic search to have installed successfully. However, elastic search did not show up in the Mantl UI. We suspect the reason to be related to the lack of resources in the test environment as the resources are lower than those suggested in the documentation for installing elastic search. Additionally, Ansible's script failed when trying to install Kafka giving the following error: **No healthy Kafka scheduler found in 300 seconds**.

5.3 Experience

Mantl is a good tool; it allows one to easily monitor the health and deploy applications entirely through the use of the Mantl dashboard. However, when deployed to the CERN OpenStack cloud a lot of problems were encountered. The main reason for this is due to the different OpenStack configuration employed at CERN. The time spent debugging errors by far exceeded the time to manually deploy cluster instances with the same services installed. The most time consuming problem was understanding why nodes were not able to install kubernetes binaries. An issue was raised on github to try and determine the reason[23]. The lack of Neutron ports in the CERN OpenStack also contributes in making Mantl a less useful solution in the CERN environment.

Consequently, due to the time lost debugging these issues, there was not enough time left to implement the scripts necessary to automate the deployment of the DDS-Mesos solution on cluster nodes.

6 Network Discovery

With the advent of dynamic resource allocation and microservices in computer clusters, jobs of different types can run on the same cluster. It is safe to assume that some of these jobs might have strict real-time constraints in order to produce useful results. A cluster that partitions resources dynamically might allocate resources that are not adapt for real-time applications. For instance, Mesos could offer resources to a real-time application that might increase the network latencies incurred by the said application.

A real-time application running on Mesos could counteract this if it knows the network topology a priori and then wait for resources pertaining to a select number of nodes to be offered. However, this is not an ideal solution as in practice, nodes can fail. If an application expects a specific resource from a specific node, then this would slow down computation even further.

One solution to this problem is for Mesos to be able to determine the network topology dynamically. Given the knowledge of the network topology, Mesos can make better scheduling decisions for real-time applications as the nodes with the least number of hops could then be allocated/offered to these applications.

While it is relatively easy to determine the network topology of Layer 3 devices such as routers, it is harder to determine the network topology of Layer 2 devices such as switches. Layer 3 devices generally implement protocols like SNMP (Simple Network Management Protocol) that work using IP to help determine the network structure. Switches, on the other hand, work at a lower level and do not decode IP packets.

This part of the project explores techniques used to determine the network topology of Layer 2 devices, specifically Ethernet switches. Knowing this information could increase the performance of dynamically partitioned cluster computers as better scheduling decisions could be performed.

6.1 Switches and Ethernet

Ethernet switches communicate using Ethernet frames. The most common Ethernet frame format, Ethernet Type II is illustrated below:

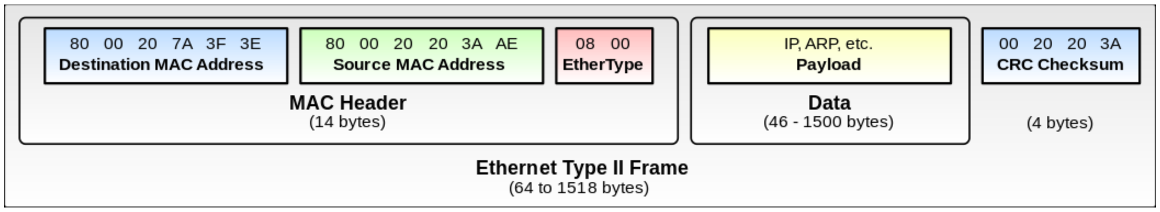


Figure 7. Ethernet Type II Frame [19]

The MAC header has all the required information such that a packet can travel to its destination. Ethernet network cards on node devices are associated with a unique MAC address. A MAC address is 6 bytes long. The MAC header thus consists of a destination MAC address so that a packet can arrive to its destination, a source MAC address so that the recepient knows the origin of this packet and EtherType field. The EtherType fields identifies the protocol that the payload is encoded with. This field helps the kernel to use the right decoding algorithms for the payload. An example value for the EtherType field is 0x0800 which identifies the IPv4 protocol. Therefore, the kernel would know that it expects the packet's payload to start with an IP header.

During operation, switches 'learn' on which port a MAC address resides by using a tabular data structure that maps a MAC address to a port (a physical ethernet port located on the switch). This data structure is filled and manipulated whenever a packet is received on the switch and forwarded to another location. If the switch has no knowledge of where it should forward an Ethernet packet because it finds no corresponding entry in the map, then it will forward it to all its ports except on the port where it received the packet. Additionally, if a switch knows that MAC address A is to be found on Port 1 and it receives a packet originating from MAC address A on Port 2, then that means that the device has been rewired to port 2 during runtime and the switch updates its forwarding table accordingly.

6.2 Switch Topology Detection

The behaviour of switches can be exploited to acquire information about the structure of a network. Algorithms to get the network topology have already been developed[20] but Nowicki & Malinowski[21] develop an improved method in terms of practicality. An implementation of this has been developed and published on Github [24].

The algorithm is divided into three stages and consists of a node (device/pc) running as a Master and the rest of the nodes running as slaves (daemons).

1. Discovery of all nodes in network

2. Discovery of all bottom layer switches – the switches that contain nodes as children

3. Discovery of the rest of the switches using relational analysis

The first step is straight forward; the master sends a message on the broadcast MAC address (FF:FF:FF:FF:FF:FF) and all the nodes reply back. A list of slave nodes can then be generated by using the source MAC address in the MAC header.

The second step can be determined by testing whether two nodes are on the same switch or not and doing this for all the nodes. A connectivity matrix is thus formed and this can be used to group nodes to their respective parent switch. An example of this is shown below where two fictitious MAC addresses UA1 and UA2 are used:

Assume the following network topology:

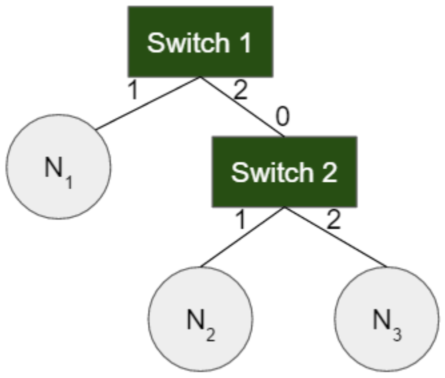


Figure 8. Example Network Topology

Determine whether N_1 and N_2 reside on the same switch:

Step	**Description**	**Switch 1 Map**	**Switch 2 Map**
1	Let N_1 send the empty from UA1 to BCAST N_1: UA1 --> BCAST	MAC UA1, Port 1	MAC UA1, Port 0
2	Let N_1 send: N_1: UA2 --> UA1 Switch 1 knows that UA1 resides on Port 1 and registers UA2 to be on Port 1 as well. Thus, the switch will not forward the message anywhere.	MAC UA1, Port 1 MAC UA2, Port 1	MAC UA1, Port 0
3	Let N_2 send the empty from UA1 to BCAST N_2: UA1 --> BCAST The switches will update their map table accordingly with regards to the location of UA1.	MAC UA1, Port **2** MAC UA2, Port 1	MAC UA1, Port **1**
4	Let N_2 send: N_2: UA2 --> UA1 Switch 2 knows that UA1 resides on Port 1 and registers UA2 to be on Port 1 as well. Thus, the switch will not forward the message anywhere.	MAC UA1, Port 2 MAC UA2, Port 1	MAC UA1, Port 1 MAC UA2, Port 1
5	Let N_1 send test message: N_1: MAC_{N1} --> UA2 Switch 1 registers the new entry. Switch 1 will do nothing else since the source and destination port are the same.	MAC UA1, Port 2 MAC UA2, Port 1 MAC MAC_{n1}, Port 1	MAC UA1, Port 1 MAC UA2, Port 1

To determine whether N_1 and N_2 reside on the same switch, one checks whether N_2 received the test message coming from N_1. In this case, Switch 1 did not forward the message and therefore it can be concluded that they reside on separate switches.

Determine whether N_2 and N_3 reside on the same switch:

Step	**Description**	**Switch 1 Map**	**Switch 2 Map**
1	Let N_2 send the empty message from UA1 to BCAST N_2: UA1 --> BCAST	MAC UA1, Port 2	MAC UA1, Port 1
2	Let N_2 send: N_2: UA2 --> UA1 Switch 2 knows that UA1 resides on Port 1 and registers UA2 to be on Port 1 as well. Thus, the switch will not forward the message anywhere.	MAC UA1, Port 2	MAC UA1, Port 1 MAC UA2, Port 1
3	Let N_3 send the empty message from UA1 to BCAST N_3: UA1 --> BCAST Switch 2 updates the port change for that source Mac address.	MAC UA1, Port 2	MAC UA1, Port **2**
4	Let N_3 send: N_3: UA2 --> UA1 Switch 2 knows that UA1 resides on Port 2 and registers UA2 to be on Port 2 as well. Thus, the switch will not forward the message anywhere.	MAC UA1, Port 2	MAC UA1, Port 2 MAC UA2, Port 2
5	Let N_2 send test message: N_2: MAC_{N2} --> UA2 Switch 2 registers the new entry. Switch 2 will forward the message to Port 2 since the source and destination ports are different.	MAC UA1, Port 2	MAC UA1, Port 2 MAC UA2, Port 2 MAC MAC_{n2}, Port 1

To determine whether N_2 and N_3 reside on the same switch, one checks whether N_3 received the test message coming from N_2. In this case, Switch 2 forwarded the message on Port 2 where N_3 resides, and therefore it can be concluded that they reside on the same switch.

In the third and final step of this algorithm, similar techniques exploiting the behaviour of switches are used to determine the network topology. The difference being that the final step works with topologies large enough to have three bottom layer switches. This method works by listing all the possible combinations in groups of 3 of bottom layer switches and determining which bottom layer switch has a different parent switch than the rest. If one such switch exists, then it can be concluded that this switch is at a level higher in the network graph than the other two and a relationship of the form S1 < {S2, S3} (S1 is at a higher level than S2 and S3) can be stated. When all the facts from each combination are collected, they are combined together and using logical inference the network topology can be approximately defined.

6.3 Results

This algorithm has been tested on two different topologies. One topology was constructed by using four Raspberry Pi while another one was constructed using four nodes on a test cluster here at CERN. The following figures depict the topologies involved.

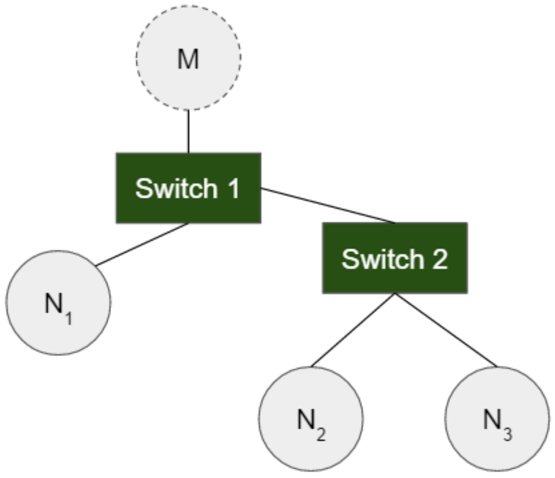


Figure 10. Raspberry Pi Network Topology

The topologies depicted in Figures 10 and 11 have been correctly determined. However, these network topologies are too small when compared to actual production environments.

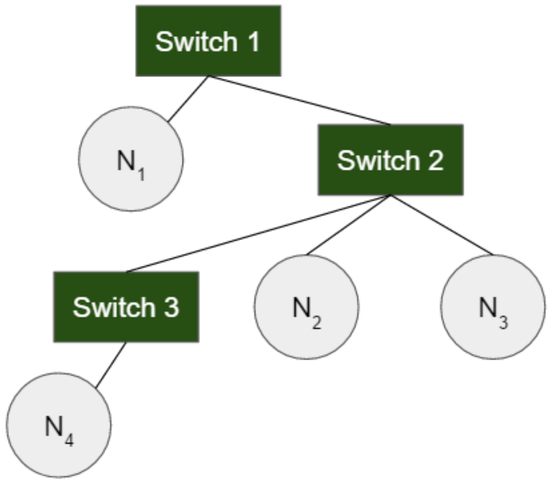


Figure 11. Alienvtest Test Cluster

Testing larger and complex network topologies is not always possible and it is impractical. Therefore, an Ethernet network simulation layer has been implemented into the application. Test topologies were declared in an XML file and the results were promising. In some cases, the algorithm does not detect the topology correctly specifically when a switch has only one node as a child. This problem has been described in the limitations section.

6.4 Limitations

The algorithm is prone to packet loss and this can decrease the accuracy of the derived network topology. While some parts of the implementation can be made packet loss proof using acknowledgment packets similar to TCP/IP, there are other parts where packet loss cannot be avoided. These involve test messages where the receipt of messages determines connectivity between two nodes as seen in the previous example. One solution to this problem is to repeatedly send the test message n times, such that the probability of packet loss becomes very low.

Another limitation is that switches having only one child switch cannot be detected by this algorithm. This is due to insufficient data generated in Step 3 of this algorithm. A minimum of two child switches is required in order for test messages to be able to determine the presence of a switch.

Currently, the master node only coordinates the slave daemons. A slave daemon and a master daemon cannot be run on the same node. This means that the node on which the master node is situated cannot be detected and will not be visible in the result generated. This issue can be solved by making the master code act as a slave when it is involved in connectivity queries. The temporary solution for the time being, is to run the whole algorithm twice with differing master nodes and merge both graphs.

It is worth noting that root access is needed on each node. The reason for this is that the master and slave daemons need access to raw sockets running in promiscuous mode. The Linux kernel allows only root users the access to this functionality.

6.5 Alternatives

An alternative algorithm for network topology detection has also been studied. The algorithm, originally proposed to be used as an MPI tool by Lawrence and Yuan[25], is based on ping times between nodes. A time matrix is first generated, then it is converted to a hop count matrix using data clustering techniques and finally from the hop count matrix, the network graph can be accurately derived. This algorithm has been implemented and tested and is available on Github [24].

This algorithm has been found not to perform well in practice. While a statistical method is employed to reduce noise in ping time measurements and give a 95% confidence on the accuracy of the measurement, the flaw in this algorithm is due to heterogeneous hardware. This algorithm is not effective on clusters where nodes are not identical in terms of hardware. The reason is that this algorithm derives a hop count matrix based on the time a packet takes to do a round trip. It performs data clustering in order to determine the time of one hop. However, given different nodes where one is slower than the other, the ping times vary greatly and the time of one hop can no longer be accurately determined. Due to this, the network graph generated is highly inaccurate.

7 Conclusion and Future Work

As described previously, the project has been divided into three components, namely, Mesos Framework & DDS Plugin, Mantl evaluation and Network Topology detection. The Mesos Framework/DDS Plugin was successfully implemented and worked correctly on a 40-node cluster. The Mantl evaluation took a lot of debugging effort, but a working Mantl environment has been installed. Although, other applications such as elastic search did not work and there was no more time to implement a solution to deploy the DDS environment and plugin using Mantl. Finally, the network topology detection algorithm gave accurate results for a number of random topologies.

As for future work, the Mesos-DDS work needs to be evaluated for scalability. One metric which is of interest is response time and scalability, specifically, how long it takes to get all the agents running and what happens to the response time as the number of agents requested increases. For Mantl, an Ansible role to deploy DDS and the Mesos DDS plugin needs to be implemented. Additionally, further investigations to uncover the reason why deploying other service instances under Mantl failed. Finally, for the network topology detection, a Mesos allocator could be written implementing this algorithm. This would allow Mesos and potentially the frameworks connected to it to make better decisions with regards to applications requiring low latencies between the cluster nodes.

8 References

[1] Freertos. http://www.freertos.org, 2016.

[2] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011, 2011.

[3] Paul Krzyzanowski. Process scheduling. https://www.cs.rutgers.edu/pxk/416/notes/07-scheduling.html, 2015.

[4] The ALICE Collaboration. http://alice-collaboration.web.cern.ch, 2016.

[5] The ALICE Collaboration. O2: A novel combined online and offline computing system for the ALICE experiment after 2018. Journal of Physics: Conference Series, 513(1):012037, 2014.

[6] The ALICE Collaboration. Technical design report for the upgrade of the online-offline computing system. CERN, 2015

[7] M. Al-Turany1, P. Buncic, P. Hristov, T. Kollegger, V. Lindenstruth, and P. V. Vyvre. ALFA: A new framework for ALICE and FAIR experiments. Technical report, GSI Helmholtz Centre for Heavy Ion Research, 2013.

[8] GSI Helmholtz Centre for Heavy Ion Research. The DDS User Manual (v0.10), 2014.

[9] A. Rusinov. Graphical Editor of the DDS Topology Configuration. DDS - Dynamic Deployment System. Sep 2015.

[10] DDS – http://dds.gsi.de/, 2016

[11] Simple Linux Utility for Resource Management (Slurm) – http://slurm.schedmd.com/overview.html, 2016

[12] Jyoti V Gautam, Harshadkumar B Prajapati, Vipul K Dabhi, and Sanjay Chaudhary. A survey on job scheduling algorithms in big data processing. In Electrical, Computer and Communication Technologies (ICECCT), 2015 IEEE International Conference on, pages 1–11. IEEE, 2015.

[13] Apache Hadoop – http://hadoop.apache.org/, 2016

[14] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” Commun. ACM, vol. 51, no. 1, pp. 107– 113, 2008.

[15] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for largescale graph processing,” in Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010, pp. 135– 146, 2010.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center,” in Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011, 2011.

[17] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In Proceedings of the 5th European Conference on Computer Systems, EuroSys ’10, pages 265–278, New York, NY, USA, 2010. ACM.

[18] Mesos-DDS – https://github.com/alisw/mesos-dds, 2016

[19] Ethernet Frame Image – https://commons.wikimedia.org/wiki/File:Ethernet_Type_II_Frame_format.svg, 2016

[20] R. Black, A. Donnelly, and C. Fournet, “Ethernet topology discovery without network assistance,” in Proceedings of IEEE International Conference on Network Protocols ICNP 2004, Oct. 2004, pp. 328–339. DOI: 10.1109/ICNP.2004.1348122.

[21] Nowicki, Krzysztof, and Aleksander Malinowski. "Topology discovery of hierarchical Ethernet LANs without SNMP support." Industrial Electronics Society, IECON 2015-41st Annual Conference of the IEEE. IEEE, 2015.

[22] Terraform – https://www.terraform.io/, 2016

[23] Installation Issue - Kubernetes Version 1.3 not available in yum – https://github.com/CiscoCloud/mantl/issues/1792, 2016

[24] Ethernet & Ping Discovery – https://github.com/kvnnap/ethernet-frames, 2016

[25] Lawrence, Joshua, and Xin Yuan. "An mpi tool for automatically discovering the switch level topologies of ethernet clusters." Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on. IEEE, 2008.

[26] Ansible – https://www.ansible.com/, 2016

[27] Vagrant – https://www.vagrantup.com/, 2016