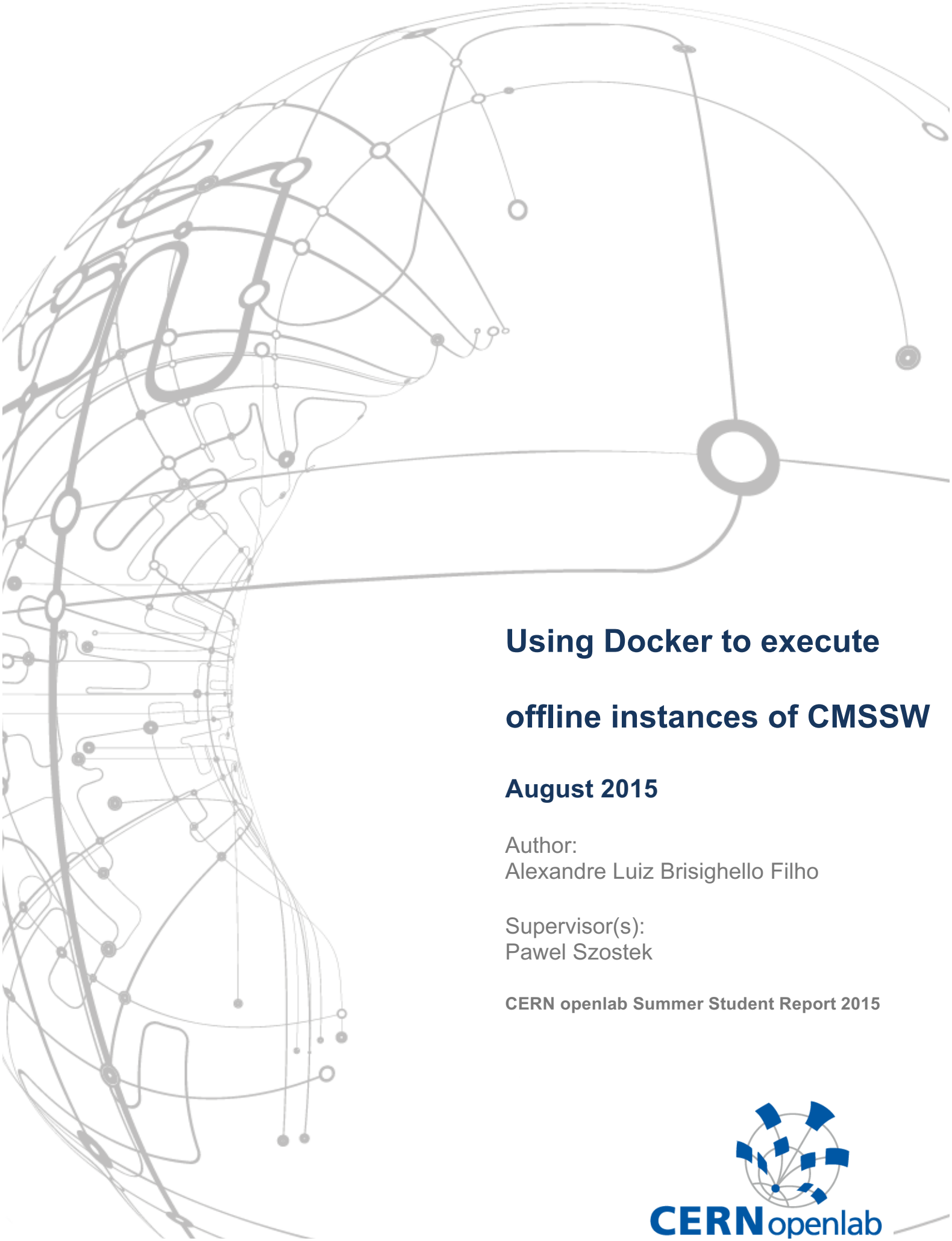

Using Docker to execute offline instances of CMSSW

August 2015

Author:
Alexandre Luiz Brisighello Filho

Supervisor(s):
Pawel Szostek

CERN openlab Summer Student Report 2015

CERN openlab

Project Specification

CMSSW[13] is a framework with simulation, calibration, alignment and reconstruction modules that perform several analyses required by physicists. It's relatively lightweight and uses configuration files, in Python, to specify data, modules, parameter and other relevant information used in the execution. One of the ways of using it, mainly for test purposes, is calling the auxiliary script runTheMatrix.py, included in the default installation, which greatly facilitates the way of invoking various processing steps.

Docker[19] is a relatively new solution (first version released in March 2013) that provides automated deployment of applications through the use of linux containers. It aims to provide an easy way for packing an application and its dependencies, resulting in an always working container that could be shared among different users and machines. There has been some discussion around using Docker to pack scientific projects[20, 21].

The main objective of this project is to generate a containerized version of CMSSW with Docker that is system-agnostic and could work without network and various online services required by the framework, providing a painless way to pack, share and run this software in different environments. The dockerized version could be used as an alternative easy-to-run real-life benchmark or to allow the execution in rough environments, e.g on machines without CMS-specific configuration.

Abstract

With the newer versions, Docker is getting more and more stable and comes up with new features. Porting CMSSW workflows into it could make it easier to reproduce the execution on machines without access to the framework or to make a frozen benchmark that will always execute correctly.

This report shows the main problems faced when generating the Docker container, the chosen solutions and some avoidable dead ends. In the end, the reader should be able to understand the main steps involved in achieving this objective.

The result of the work is an command-line tool developed in Python named Morpheus. generating containers of CMSSW workflows, allowing usage of external tools and some customization. By using of the CMSSW environment installed in the host machine, it can generate working containers for the desired workflows. All the code have been documented and it should be easy to be extended.

Table of Contents

1 Introduction

1.1 Objectives

The main objective of the project is to provide an easy way to generate offline CMSSW containers, in a clean and relatively quick way. Previous work includes a working container that allows the installation of the framework inside a Docker image, providing useful information on how to work with not only CMSSW but its dependencies using a container. One important thing is that the usage of the existing project would require network access and credentials, running only exactly like outside Docker. In other words, it still has external dependencies and doesn't allow the usage of extra tools.

Docker is becoming very popular nowadays due to increased demand for PaaS services (platform as a service). In that context, it provides not only a lot of features that may greatly save administration costs, but also a better performance when compared to Virtual Machines, closer to native execution, allowing users to share containers that will always work, regardless of which of the Docker-compatible Linux distribution is used.

Because of already existing CMS containers, relative maturity, and good support it has been receiving in the community, it was chosen as the containerizing. With the objective of making things simple and easy to code, Python is used as the programming language. The project wouldn't work in Windows though, as a Linux operating system with CMSSW environment installed is expected. The image, on the other hand, can work in windows if Docker Machine[22] is correctly installed.

1.2 Installation of CMSSW environment

Installing CMSSW was done using the same scheme present in the existent Dockerfiles[14], resulting in a similar environment inside and outside the container. At first, some dependencies are installed, including puppet (which might require a ruby update). Once this is done, a puppet file that will install the requested version is provided, allowing the user to select which one to use with SCRAM once installed.

SCRAM[11] is a tool used to manage different versions of the framework, making it really easy to separate and select which version of the framework to use in a linux environment. Once set up, all the framework will be present in the path, making possible to find all needed binaries and tools to run the workflows. It provides a set of predetermined workflows to be used with an auxiliary script named runTheMatrix.py. To perform the installation, a Puppefile is needed:

```
mod "cms-sw/cmsdist",

       :git => "https://github.com/ktf/puppet-cmsdist",
```

```
:ref => "ac5133e"
```

A manifest.pp file:

```
# An example puppet file which installs CMSSW into
# /opt/cms.

#package {["HEP_OSlibs_SL6", "e2fsprogs"]:
#  ensure => present,

#}->
file {"/etc/sudoers.d/999-cmsbuild-requiretty":
   content => "Defaults:root !requiretty\n",
}->
package {"cms+cmssw+CMSSW_7_4_7":
  ensure             => present,
  provider           => cmsdist,
  install_options    => [{
    "install_prefix" => "/opt/cms",
    "install_user"   => "cmsbuild",
    "architecture"   => "slc6_amd64_gcc491",
    "server"         => "cmsrep.cern.ch",
    "server_path"    => "cmssw/cms",
  }]
}->
package {"cms+local-cern-siteconf+sm111124":
  ensure             => present,
  provider           => cmsdist,
  install_options    => [{
    "install_prefix" => "/opt/cms",
    "install_user"   => "cmsbuild",
    "architecture"   => "slc6_amd64_gcc491",
    "server"         => "cmsrep.cern.ch",
    "server_path"    => "cmssw/cms",
  }]
}->
file {"/etc/profile.d/scram.sh":
  ensure   => present,
  content  => "source /opt/cms/cmsset_default.sh\n\n",
  mode     => 755
}->
file {"/etc/profile.d/scram.csh":
  ensure   => present,
  content  => "source /opt/cms/cmsset_default.csh\n\n",
  mode     => 755
}
```

The installation steps:

```
yum update -y && yum install -y wget git sudo HEP_OSlibs_SL6 e2fsprogs
useradd cmsbuild && install -o cmsbuild -d /opt/cmsbuild
librarian-puppet install
puppet apply -d --modulepath=/modules /manifest.pp
```

Before using it, the user must select the version in SCRAM:

```
source /opt/cms/cmsset_default.sh
scram project CMSSW_7_4_7
```

```
cd CMSSW_7_4_7
eval `scram run -sh`
```

All these files can be found in the previous work github[14], but uses a different version of the framework (just replace the version number for the required one). Once CMSSW is correctly installed, everything will be on PATH and runTheMatrix can be used. Follows a simple example:

```
runTheMatrix.py -i all -l 25
```

The command above will not simulate ("-i all" parameter means it will use the root files from CMS) and will execute only the workflow numbered as 25. More examples about runTheMatrix command and its parameters can be found in its documentation[6].

1.3 Morpheus

CMSSW works flawlessly in CERN site using the standard SLC6 distribution and having all the access rights (which means being a CMS member) and certificates. However, if you don't have access to the ROOT input files provided by EOS you will not be able to execute any of the workflows that require the data samples. The same applies to accessing the condition data base.

Aside the regular usage, it could be interesting to use part of the workflows as a benchmark, something not previously possible due to the difficulty of isolating it from the rest of the execution and generate a shareable package. With that mind, we propose a easy to use command line tool that can generate Docker containers, making it easy to share with and work in any place or machine.

More details about implementation and usage will be provided in the rest of this report.

Here is a usage example:

```
python Morpheus.py docker -i all -l 25
```

The "docker" parameter choose the docker creation option and the rest works just like runTheMatrix (it will actually, under the hood, call runTheMatrix using these parameters). The output of this command is a folder with all the files necessary to generate the container that, once created, it's able to reproduce the execution offline and share, since Docker offers the possibility to dump the image to a tar file[15].

2 Docker

2.1 Overview

Being a topic of many discussions in the last months, Docker is a tool that will not only allow you to execute linux containers, but will offer a lot of features to make as simple as possible to develop new containers. It offers a hub (named docker hub), which works as a remote service for helping in the process of building and shipping containers in a centralized fashion, providing an easy way to reuse and share. It even has some commands targeting scalability problems (see Docker Swarm[23]) and others, like mounting an external folder inside the container.

There are some reports comparing its performance with both the traditional Virtual Machines schemes and native[10,16], showing positive results because of its small overhead. In summary, in many workloads, it can save up a lot of time and computer resources.

2.2 Installation

The Linux distribution used at CERN is based on Redhat, providing all its functionalities, including the package manager. For that reason, you can use yum to install Docker in your machine:

```
yum -y install docker-io
```

Once installed, you can check the service status and start it, if it's not running yet.

```
service docker status
```

For the current versions, all the commands require superuser privileges by default. This can be achieved by using a root user or sudo (depending on your distribution) for all the commands. Although there is a workaround for the commands in some distributions like ubuntu, the root privileges seem to be mandatory for Red Hat[24].

2.3 Configuration

Due to large size of the files that a container might have, it's recommended to modify the maximum allowed size in the configuration of Docker, which by default is set to 10GB. To do that, there are two options. The simpler one[1] is to change the Device Mapper configuration, which in SLC can be achieved with the following commands:

```
service docker stop
rm -rf /var/lib/docker
cat <<\EOF > /etc/sysconfig/docker-storage
DOCKER_STORAGE_OPTIONS="--storage-opt dm.basesize=50G"
EOF
service docker start
```

The second one requires AUFS support, not present by default in Red Hat distributions, resulting in requiring a kernel update. This might look inconvenient at first, but, if the user could perform this, it would be possible to install the more recent versions of Docker, providing a more stable system. After doing that, you only need to start the daemon with AUFS option:

```
docker -d -s aufs > /dev/null 2> /dev/null &
```

3 Data Samples

3.1 Data Aggregation System (DAS)

Some workflows included in the framework require data samples not available locally. This is handled by the Data Aggregation System (DAS)[3], which is basically a search engine for data. It receives a request in the format of a query and outputs the path of all the files needed by the execution.

It provides two interfaces, a web version, available through the DAS web page (https://cmsweb.cern.ch/das/) and a command line version. The latter is the one used by the workflows, being the first step of the execution when data samples are required. CMSSW also provides tools for copying and picking only a chosen number of events of these files.

Access to CMS samples is restricted, since they may contain confidential data. Consequently, it's mandatory to be registered as a CMS user and use a personal certificate. This registration can take some time, therefore if you still don't have access to this data, be aware this is not a 1-day process.

3.2 das_client.py

The command line interface is accessed through a python script named das_client.py, which receives a das query as input and saves a list of all the remote data samples required for the next steps in a output file. Since the objective is to make a container work regardless network access, the remote access of data samples is one the things that must be chance changed. The DAS also requires internet access, which would be nice to avoid.

The das_client was then modified to intercept and cache all the requests, making use of a python dictionary that maps each das query to the respective list of files it would normally output. The output is modified to point to local files instead of remote ones and the cache is saved, for later use, with the pickle library[26], making it persistent. A lot of scripts and parsers to locate the required files have been developed and the local files must be in the folder present in the environment variable "ROOT_FILES" or, if it's not defined, in /data/ (which is the folder used for execution in the existent Docker images available).

One important detail: das_client.py output is verified in step 2 by another CMSSW script (WorkFlowRunner.py) that checks if at least one remote file was added in the created output file. To bypass this there are two options, each one with a different negative side. The first option would be to modify also the verification script to accept only local files as a correct output, which can introduce problems if there is an update in the verification in new version. And the second one is to just put anything that looks like a remote file in the end of your DAS output, what can generate problems if cmsRun actually tries to open that. Both solutions were tried and since the latter seems to work in all tested workflows,

it was the final choice. For the workflow 25, the output would be the following (/blue_pill is the non-existent file):

```
/blue_pill
file:/data/221E066C-69D1-E311-8E63-0026189438FA.root
file:/data/46102B27-7DD1-E311-8DFC-002618943834.root
file:/data/AC879CD6-38D1-E311-A80B-002618943914.root
file:/data/C0A95745-47D1-E311-B520-002590593878.root
```

3.3 Truncating input files

In order to save space we encapsulate in the container only the part of the data that will actually be used. In runTheMatrix.py workflows, this is specified by the "-n" parameter and it's usually 10 or 100.

Some parsers to identify this number were made. The most successful strategy was to execute all the workflows without access to the data samples and try to identify using the execution logs which files were requested. Other possibility is parsing a log generated by runTheMatrix when the following command is used:

```
runTheMatrix.py -i all --what standard --wmcontrol init --noCafVeto | tee creating.log
```

This command is responsible for creating all standard workflows configurations, calling cmsRun with "--no_exec" flag. Although all the calls to das_client.py will be made, it's not enough to define exactly what files will be actually used in each workflow, since many of them only use the first file of the list. Taking that into account, copying every file would be a mistake.

After identifying the files required by each workflow, it's just a matter of getting the number of events in the next step and pick the events (reducing the size). There is documentation regarding this topic1, but, to make life easier, a function that automate automates the process was implemented (called "cut_root_file")

4 Conditional Database

4.1 Frontier

Another source of external data used by the workflows is intermediated by the Frontier5. It's a "distributed database caching system that distributes data from data sources to many clients". In the context of CMSSW, it's uses an https connection to request conditional data from the detectors, usually using a remote Squid[17].

Just like the das_client.py problem, in order to make the container completely independent of any network connections, it's imperative that this outside communication doesn't occur. Three options for removing the need for connecting to Frontier were investigated, trying to mock the https request in different ways. A brief description of each one is given in the following paragraphs, with the results obtained.

4.2 LD_PRELOAD trick

There is a well known Linux hack[7] that allows a shared object to be loaded before any other library, including libc. This allow the programmer to replace any function call by another one he defined, thus allowing any call from the Frontier library to be intercepted. Once a call is intercepted, it's possible, among other things, to call the original function from the modified one[25]. An advantage of this approach is that it would still work after a possible update if the function signatures remain the same.

In an ideal world, it would be easy to intercept every function call responsible for http requests, cache it and avoid calling the original function after the first time. But, even though frontier is open source and it's possible to find the signatures of each function, caching the data structure used by frontier showed to be very problematic. It uses a not well defined data structure (including void pointers, casts and converting a memory address to an integer) making this approach very prone to errors.

After some time spent looking at different solutions, including Protocol Buffers and serialization libraries, the conclusion is that the LD_PRELOAD trick is not the right strategy to use. In other words, if you need to cache frontier, this might be the worst possible way (assuming it's actually possible).

4.3 Sqlite files

The second option would be to replace the frontier client entirely with an SQLite local database. Our colleagues from CMS (Andreas Pfeiffer and Gianluca Cerminara) kindly prepared the required database files and also the necessary code modifications. This is a viable option, it works and, for the tested workflows, the increase in the container size due the usage of this scheme is not relatively big.

One problem that surface from this is the ability to generate new files. If these files have to manually be generated for different versions, for example, this could be a very big downside. Aside from that, this is a good solution and adding this as an option to Morpheus is desirable as this option would require fewer executions in the container creation phase.

4.4 Offline Squid proxy

Last but not least, there is the offline Squid proxy option. Frontier uses a special version of squid (called frontier-squid[18]) that has some additional features and configurations. The idea behind this strategy is to cache all the https requests, allowing them to be used again once present in the cache. Since it's not possible to assume that the user will have a local squid running, a new execution during docker creation is necessary.

The harder part is in the configuration of squid: it first need to be configured to cache all the files, without exception, and to never delete the cache. Later, it necessary to make it never check if some request is expired (TCP_REFRESH_MISS)[12]. The reason is simple: the execution should be exactly the same, so expired requests is not a concern. Again, since the execution is completely offline, every file used needs to be cached and included.

To achieve the point where every conditional request is cached, a bash script (squid.sh) that creates all the required configurations was made, being called during the container creation. Since all the workflows use the same of connection, the same file can be used to all the possible workflows. For that reason, this solution is the easier one to use and seems to be working without any issues.

5 Care

5.1 Motivation

As stated before, one of the problems we had to face concerns the size of the created containers, as they can become big rather easily. The CMSSW framework alone corresponds to something between 4 or 5 GB, when root files and conditional data are included the size will grow over 10GB for a small number of workflows.

Once the list of root files required for the execution is defined (using das_client.py) and the Squid cache is full, the paths of all used files required for execution must be known. Not only to avoid the addition of useless files in the container but also to assure that the programs used are the same, guaranteeing correct behaviour. A fresh installation could result in a container containing a different version of the framework, resulting in different results.

Generating a strace dump and parsing it would be difficult and may generate very big logs, making this a bad solution. In reality, the perfect solution would be a tool that selects and packs every file used in a given execution. It's also important that it doesn't create insanely giant temporary files or result in a very expensive computation cost. That's where Care comes into scene.

5.2 Care tool

Care[9] is a very interesting linux tool, that aims to provide a quick way to make an execution reproducible. Since it doesn't record any system events (as system calls or keystrokes), it is relatively fast and it outputs a really nice package with everything that was used in the execution. After extracting this package, the user is supposed to run a re-execute script (re-execute.sh, created by Care) that will make the job of repeating the execution, exactly right as the first time.

Not only the format used is really nice and easy to move, but this tool works on basically all linux distributions without any dependencies: it's a neat static binary. For these reasons, Care is not a dependency to run Morpheus, it was just inserted in the project. Using it with the framework didn't result in any side-effect and was the chosen as the solution to deal with identifying all the files needed in a given workflow.

5.3 Re-execution

In order to reexecute, Care makes use of proot and, for that reason, uses an intermediate layer in the execution, appearing in system monitors (top, for example) with a different process name. Aside from the bad aesthetic effect, this extra layer might introduce extra complexity to the execution, which is undesired. The perfect solution would be a container that doesn't have this extra layer.

To override the problem, some modifications in the Docker structures are needed. The solution of choice also tries to avoid a second possible issue : usage of absolute paths (acessing /data/221E066C-69D1-E311-8E63-0026189438FA.root, for example, both the filename name and path must be the same). When the execution is done outside Care, it's important to guarantee that files will be where they were in the original execution if an absolute path is used. Any crypt access made by CMSSW would not result in a crash.

With that in mind, two extras steps were added:

a) The rootfs folder in the package file created by Care is moved to the root ("/") folder. A important thing to note: you can't overwrite any file (specially system files) or some applications will be broken.
b) A script that sets all the environment variables like the original execution is created. It's used just before the new execution, making them exactly the same. One important detail to have in mind: these variables should only be used to execute the framework and nothing else. When using an extra tool, for example, the tool should use different configuration than the CMSSW calls.

The result of the work until now would be a Docker image with the same environment of the original execution but with every data needed for execution. Our modified das_client.py would point to local files instead of remote files, the offline squid that has been installed has all the needed queries in cache and Care would select all the needed files for the execution.

6 Morpheus

6.1 Overview

Morpheus is a command-line python script that aims to join all the steps previously discussed, to make the process of generating new CMSSW containers as easy and painless as possible. It will try to run all the steps locally, look for needed data samples and the commands used by each workflow, dynamically generate a custom execution script, generate the package with all the files needed using Care, and create the container with Docker.

The dependencies are the following:

- bash (not zsh or tcsh)
- Python 2.* (Tested with both 2.6 and 2.7)
- ArgParse (yum install python-argparse in SLC)
- Paramiko (only if using remote ssh)
- CMSSW framework installed and active in SCRAM.

The next sections will give more details about Morpheus and use cases.

6.2 Parameters

Morpheus accepts differents positional parameters during the creation phase.

- creating — Parse the creating.log and get info from it.
- truncate — Truncate the root files and save them locally
- matrix — Parser runTheMatrix output.
- das_recovery — Recovery the full path outputted by das_client.py.
- docker — Dockerize some workflows.
- docker_results — Outputs a table with the results obtained in the docker.

All the options have additional parameters needed for the execution, it's possible to see a list with a description with the following command:

```
python morpheus creating -h
```

Replace creating for the command needed. The arguments parsing was provided by argparse, a very well known python library.

6.3 Container Generation

For creating an image we have to run the desired workflow(s) three times for the squid version and two for the SQL version (the two options for caching conditional data for the

workflows). It's assumed that the the user has access to the root files. A summary of each execution:

1) Normal Execution
   - Identify ROOT files opened.
   - Grab the files and truncate them using the number of events from the command.
   - Generate a list of commands to be executed for each workflow.
   - Create a bash script that contains the whole execution. Make it possible to use the return value from each command, which avoids having to parse the output of runTheMatrix.py allowing different commands to be attached.
   - Determine steps of the execution

2) Care Execution
   - Usage of the script created above using the Care tool. Thanks to it we don't have to include the whole CMSSW and dependencies in the image.
   - Create Care file, which is later unpacked inside docker.

3) Docker Execution (squid version only)
   - This is executed during the creation of docker, used to fill up the frontier-squid cache.
   - Uses a custom configuration for the frontier-squid, forcing it to cache everything. Change it to only use offline files in the end.

Although the SQL version doesn't need the third step, which could result in saving some time, the squid option seems to be slightly smaller and easier to create once with the right configuration in hand. It's the most stable one and doesn't require neither the SQL dumps nor any modification in CMSSW.

6.3.1 Squid example

Using Squid cache is the default option for the conditional data. All the root files are expected to be in /data folder and, if you need to change their location, just modify the "exec_path" variable in morpheus.py file. An example of usage:

```
python morpheus.py docker -i -l 25
```

The command above will create a folder (named "morpheusialll25") with all the files required to build a docker with the 25 workflow. After that, to actually build the container, use the following command:

```
sudo docker build -t morpheusialll25 .
```

It will not only build the container but also run it once, to create the Squid cache. If you want to do it automatically, you can use:

```
python morpheus.py docker -i -l 25 -create –sudo
```

With the container created, just run:

```
sudo docker run morpheusialll25
```

If only one workflow was included, the step can be selected and an optional tool can be used:

```
sudo docker run morpheusialll25 –s 2 –u time
```

The commmand above would execute only the step 2 and use time, to get the time spent. Both time and perf were tested. Other more sophisticated tools (like Intel Vtune) might require more complex modification, like cmsDriver.py or Docker volumes.

6.3.2 SQL example

The SQL version is experimental, and because of that, more complex. It's necessary to use the version 7.5.0, and, once the framework is in path, execute the following commands in /data/CMSSW_7_5_0/src:

```
cd /data/CMSSW_7_5_0/src
git cms-addpkg Configuration/PyReleaseValidation
tar zxf /afs/cern.ch/user/a/andreasp/workspace0/public/GT-inSqlite.tgz
scram b -j 10
```

Different from the other version, it will remove all the data from previous executions in /data/CMSSW_7_5_0/src/tmp and /data/CMSSW_7_5_0/src/exec. Since the path used can change and to force the user about data being removed, a environment variable must be exported. Besides avoiding accidental loss of files it can support different installations configuration.

```
python morpheus.py docker -l 1 –sql
```

The output is similar to the Squid version, and if the parameters "-create -sudo" are not used, it's needed to create it manually, in the folder created:

```
sudo docker build -t morpheusl1 .
```

Execution is done in the same way of the Squid version and, again, if there is only one workflow was selected, the –s and –u parameters are available.

6.4 Parsers

Identifying and copying the necessary data samples can be a pain. To make the process easier, many parsers were implemented, each one targeting a different log or output. There is a brief description and example for the different options.

6.4.1 creating

A debug command, used to parse creating.log and output some useful information in a table format. It will not copy files, only help the visualization of the information contained in log file. To generate the creating.log, check section 3.3 or the documentation[6]. A simple example:

```
python morpheus.py creating -steps
```

6.4.2 truncate

Using a remote folder, the parser will first verify which of the remote files are already

present in the local files. It will, then, look into the das_client.py dictionary, copying and cutting the files not yet available in local.

Lastly, it will then retrieve all the commands for each workflow from the creating.log file generated by runTheMatrix and parse the resulting logs, looking for any attempts of data access. If any uncessefull file request was found it will try to copy from remote and, if successfully, try to run again. The full process can take quite some time but it's was possible to identify some of the necessary files not present in das_client.py output. All the files will be moved to /data.

```
/usr/bin/python morpheus.py truncate --path olsnbc03:/data/guest --remote -
-remote_login guest --remote_password 5\`}NVY#w\(d+G7px} --parse_logs
```

One thing to note: this is the only option that requires /usr/bin/python and not only python. CMSSW includes a different version of python, without access to the paramiko library. Therefore, for the truncate command, don't use the python present in CMSSW framework.

6.4.3 matrix || docker_results

Will display a table with the results of a given execution using the stdout from runTheMatrix.py (matrix) or the docker script output (docker_results) command. It accepts commands with one or more workflows and will output a table, allowing the user to see what of was executed correct a what wasn't.

```
python morpheus.py matrix --file runTheMatrix.log
```

6.4.4 das_recovery

Output the real path of a given list of data samples. It may be useful when only the name of the file is known but the full path is needed for copying reasons.

Follows an example:

```
python morpheus.py das_recovery --file das_test --das_file das_dic
```

7 Conclusion

The idea of being able to execute CMSSW framework in different machines and as a benchmark can be useful for CERN. With that idea in mind, the tool developed aims to solve that problem, trying to save time and making it as easy as possible for the users. All the code has been done in python and is well documented, being easy to use and continue the work or extend the features if necessary.

During the work a lot of possibilities were considered and many of them were tried. The proposed solution is the ensemble of the ones that have worked best. Different steps for the creation were defined and an attempt to join all the work in an automated script was made. The main objective of the tool is to allow the generation of CMSSW containers without the worry of all the matters described in the previously chapters.

The implemented command line tool makes it possible to create a working container for different workflows, allowing them to be used completely offline with some useful features and in any machine with Docker support. Using the created container as a benchmark is just a question of choosing a desired workflow and creating a shareable tar file.

8 Acknowledgement

My stay at CERN was very pleasant, especially due the presence and help of Pawel Szostek, possibly the very best **BOSS** around, and my colleague Estefania, giving her position on several topics.

9 References

1. Docker size configuration : http://ktf.github.io/2015/04/run-marathon-on-laptop/
2. Access to CMS data : https://twiki.cern.ch/twiki/bin/view/CMSPublic/SWGuideLcgAccess
3. DAS : https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookLocatingDataSamples
4. Picking events : https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookDataSamples#Copy_Data_Locally
5. Frontier : http://frontier.cern.ch/dist/
6. runTheMatrix : https://twiki.cern.ch/twiki/bin/view/CMSPublic/CompOpsRelValWMAOperations
7. LD_PRELOAD trick : http://www.linuxjournal.com/article/7795
8. Protobuffers : https://developers.google.com/protocol-buffers/?hl=en
9. Care : http://reproducible.io/
10. Performance : http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf
11. Scram : https://twiki.cern.ch/twiki/bin/view/CMSPublic/SWGuideScram
12. Squid Reference : https://books.google.ch/books?id=fVuWayXLdYIC&pg=PA228&lpg=PA228&dq=TCP_REFRESH_MISS+squid&source=bl&ots=HrqSQUnl3b&sig=FE7BwM-IAbecla2iP73g04FP34M&hl=en&sa=X&ved=0CD8Q6AEwBWoVChMI5OvytM2MxwIVBQ8sCh3hGgm1#v=onepage&q=TCP_REFRESH_MISS%20squid&f=false
13. CMSSW : https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookCMSSWFramework
14. CMSSW Docker repository : https://github.com/cms-sw/cms-docker/tree/master/cmssw
15. Docker Save : https://docs.docker.com/reference/commandline/save/
16. Hypervisors vs. Lightweight Virtualization: a Performance Comparison : http://metrics.it.uc3m.es/wp-content/uploads/ic2e.pdf
17. Squid : http://www.squid-cache.org/
18. Frontier-squid : https://twiki.cern.ch/twiki/bin/view/Frontier/InstallSquid
19. Docker : https://www.docker.com/
20. DOCKER FOR REPRODUCIBLE COMPUTATIONAL ANALYSIS : https://rc.duke.edu/dan-leehr-duke-docker-day-2/
21. Nextflow : http://www.nextflow.io/blog/2014/nextflow-meets-docker.html
22. Docker Machine : https://docs.docker.com/installation/windows/
23. Docker Swarm : https://docs.docker.com/swarm/
24. Red Hat Docker : https://access.redhat.com/articles/881893
25. LD_PRELOAD example : https://scaryreasoner.wordpress.com/2007/11/17/using-ld_preload-libraries-and-glibc-backtrace-function-for-debugging/
26. Pickle : https://docs.python.org/2/library/pickle.html