Towards reproducibility of computational environments for Scientific Experiments using Container-based virtualization.

Maximiliano Osorio, Carlos Buil-Aranda, and Hernán Vargas

Informatics Department, Universidad Técnica Federico Santa María, Chile
{mosorio, cbuil, hvargas}@inf.utfsm.cl

Abstract Experiment reproducibility is the ability to run an experiment with the introduction of changes to it and getting results that are consistent with the original ones. To allow reproducibility, the scientific community encourages researchers to publish descriptions of the these experiments. However, these recommendations do not include an automated way for creating such descriptions: normally scientists have to annotate their experiments in a semi automated way. In this paper we propose a system to automatically describe computational environments used in in-silico experiments. We propose to use Operating System (OS) virtualization (containerization) for distributing software experiments throughout software images and an annotation system that will allow to describe these software images. The images are a minimal version of an OS (container) that allow the deployment of multiple isolated software packages within it.

CEUR-WS.org/Vol-2184/paper-05.pdf

1 Introduction

Experiment reproducibility is the ability to run an experiment with the introduction of changes to it and getting results that are consistent with the original ones. Introducing changes allows to evaluate different experimental features of that experiment since researchers can incrementally modify it, improving and repurposing the experimental methods and conditions [12]. To allow experiment reproducibility it is necessary to provide enough information about that experiment, allowing to understand, evaluate and build it again. Usually, experiments are described in scientific workflows (representations that allow managing large scale computations) which run on distributed computing systems. To allow reproducibility of these scientific workflows it is necessary first to address a workflow conservation problem, since experimental workflows need to guarantee that there is enough information about the experiments so it is possible to build them again by a third party, replicating its results without any additional information from the original author [7].

To achieve conservation the research community has focused on conserving workflow executions by conserving data, code, and the workflow description, but not the underlying infrastructure (i.e. computational resources and software components). There are some approaches that that focused on conserving the

environment of an experiment such as the work in [10] or the Timbus project[1] [4] that focuses on business processes and the underlying software and hardware infrastructure. The authors in [10] identified two approaches for conserving the environment of a scientific experiment: physical conservation, where the research objects within the experiment are conserved in a virtual environment; and logical conservation, where the main capabilities of resources in the environment are described using semantic vocabularies to allow a researcher to reproduce an equivalent setting. The authors defined a process for documenting the workflow application and its related management system, as well as their dependencies. However this process is done in a semi-automated manner, leaving much work left to the scientists. Furthermore, usually most works leave out of the scope the physical conservation of the execution of scientific workflows. In this paper we approach to both, physical and logical conservation problems.

Herein we propose to improve the physical conservation solution by using operating-system-level virtualization. This technology, also known as containerization, refers to an Operating System (OS) feature in which the OS kernel allows the existence of multiple isolated user-space instances called containers. One of the most popular virtualization technologies is Docker[2], which implements software virtualization by creating minimal versions of a base operating system (a container). Docker Containers can be seen as lightweight virtual machines that allow the assembling of a computational environment, including all necessary dependencies, e.g., libraries, configuration, code and data needed, among others. Docker distributes this computational environment through software images. The main problem for conserving the physical environment of an experiment is the amount of space needed. With container virtualization it is possible to reduce to the minimum the size of the virtual machine needed for running the scientific workflow. We propose first to use Docker images as means for preserving the physical environment of an experiment. We use containers since they are lightweight and more importantly, they are easier to automatically describe so we improve the process of documenting scientific workflows. We do that by describing the workflow management system, as well as their dependencies by developing an annotator system for the Docker images before.

In the paper, we present a new system that scans the DockerHub portal for Docker images and annotates them, achieving logical conservation by using container-based virtualization. We present this as a Proof-of-Work that will allow to conserve the logical environment of an experiment. We assume that the physical environment is already preserved at the Docker images we describe semantically.

2 Docker Overview

Docker is a technology that allows virtualizing a minimal version of an Operating System. Therefore users can run applications within it. Throughout this section,

[1] http://www.timbusproject.net/
[2] https://www.docker.com/

we introduce how Docker and its registry (Docker Hub) work, starting with how Docker images are created and stored in Docker Hub.

2.1 Docker repositories and files

Docker builds a software image by reading a set of instructions from a Dockerfile. A Docker file is a text file that contains all commands to build a Docker image. Docker files usually have multiple lines, which are translated into image layers whereas Docker builds the image. In the build process, the command is executed sequentially, creating one layer after the other. When an image is updated or rebuilt, only modified layers (i.e., modified lines) are updated.

2.2 Publishing and Deploying Docker images from Docker Hub

Docker Hub is an online registry that stores two types of public repositories, both official, and community. Official repositories contain public, verified images such as Canonical, Nginx, Red Hat, and Docker. At the same time, community repositories can be public or private and are created by any user or organization. By using that registry and a command line, it is possible to download and deploy Docker images locally as a running container into a host executing thus the software within the image. Anyone has the chance to create and store images into the Docker Hub registry by first creating a descriptor file called Dockerfile. This descriptor describes what software packages will be within the image, builds the image and finally uploads it to Docker Hub. However, Docker Hub does not control what packages are in the images, whether the image will deploy correctly or the images might have any security problem. Thus, Docker images work as a black box, which refers that users know that the main software package runs within the container but they do not know the other packages needed to run it.

There are two ways of uploading images to a user repository, either by a push from a local host or automating that process from a Github repository. In order to push a repository to the Docker Hub, the users need to name their local images using their Docker Hub username, and the repository name they had created. Afterwards, users add multiple images to a repository by adding a specific `:<tag>` to it. This is all the information that normally Docker images have, being thus almost impossible to reproduce the execution environment if any of the used software packages within the images is modified.

3 DockerPedia Resource

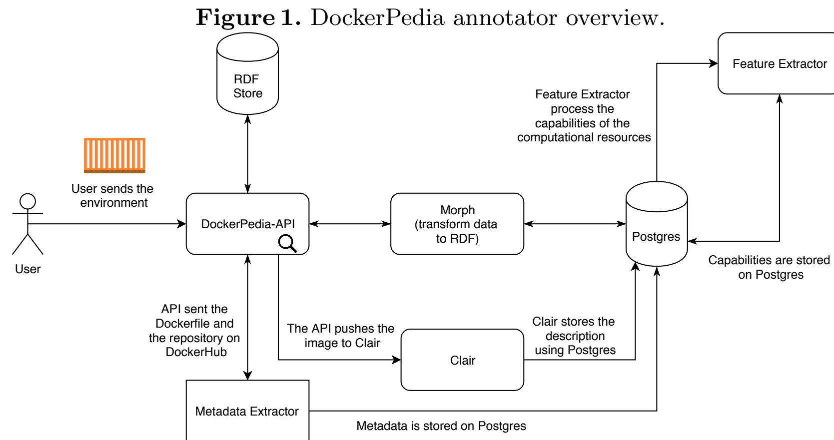
In this section, we describe how we automatically annotate the software images stored at the DockerHub. Once these images are annotated we claim that we can obtain the execution environment of a scientific experiment by using a single SPARQL query. We store the descriptions generated at our RDF database called DockerPedia (available at https://dockerpedia.inf.utfsm.cl/). This RDF database is available as a 5-star linked data resource.

3.1 DockerPedia Annotator

We first describe the images at DockerHub by using the DockerPedia Annotator. This tool extracts and annotates the software components, their version and metadata of a software image automatically. DockerPedia Annotator relies on four components: DockerPedia API, Metadata Extractor, Feature Extractor and Clair. First, DockerPedia-API receives the user environment and sent it to two components: Metadata Extractor and Clair. The Metadata Extractor extracts metadata from Dockerfile and DockerHub and saves the data on Postgres through DockerPedia API. In parallel, Clair downloads all layers from an image as a filesystem, mount it and analyzes it, determining the layer's operating system, and the packages added and removed from the layer. Finally, the Feature Extractor organizes the image layers and computes the resultant packages.

The Figure 1 shows an overview of the process.

Figure 1. DockerPedia annotator overview.



Having the Docker images semantically described it is possible to obtain their descriptions all the packages (including their versions) and dependencies by using a SPARQL query. Notice that this is not possible by just looking at the Docker file from the original image (assuming it exists such file) since the Docker file only shows the main packages to be installed (i.e. through commands like apt-get install package_name). In Listing 1.1 we show all packages that are installed within the TensorFlow Docker image:

Listing 1.1. Query to obtain TensorFlow image packages

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX vocab: <http://dockerpedia.inf.utfsm.cl/vocab#>

SELECT ?packagename ?packageversionint WHERE {
  ?image vocab:id 3013931 .
  ?image vocab:hasLayer ?layer .
  ?modification vocab:modifiedLayer ?layer .
  ?modification vocab:relatedPackage ?packageversion .
  ?package vocab:hasVersion ?packageversion .
  ?package rdfs:label ?packagename .
```

```
 ?packageversion rdfs:label ?packageversionint
} limit 500
```

The results from the query in Listing 1.1 are the 183 packages needed to run the TensorFlow image at DockerHub. With this data available, a scientist developing a TensorFlow algorithm today would know exactly what software packages she needs to run again that algorithm in the future, replicating the exact same execution environment.

3.2 The Annotation Process

To annotate the Docker images we need first to obtain them from the Docker Hub. In February 2018, this search returned 1,363,510 Docker repositories and 4,608,443 images composed of 4,593,602 community images and 14,841 official images. The total size of these images is 53.47 PB. Our machines do not have the computing power nor the network requirements needed to perform the analysis to that amount of data. Therefore, we only analyzed the 160,000 most downloaded Docker images from Docker Hub. To perform that analysis we used seven virtual machines with following specifications: 2 CPU (2.20GHz) and 3 GB of memory from Digital Ocean.

Once we discovered the name of the images to analyze, we obtain from the Docker Hub the Username, repository name, image description, whether the image is an automated build or not, when the image was updated for the last time, number of pulls and the number of stars of that image as well. We also obtain all versions of each image (which is called "tag" by the Docker Hub). For instance, the Docker repository "google/cadvisor" has 59 different images of the original software, and each image has different packages. By each of the tag, we obtain its name when was updated for the last time and its size.

After that, we use the tool from the Clair project[3] to detect the packages and vulnerabilities within each downloaded image. Clair is an open-source tool from CoreOS designed to identify known vulnerabilities in Docker images. Clair has been primarily used to scan images in CoreOS is private container registry, Quay.io[4], but it can also analyze Docker images. Clair downloads all layers from an image as a file system, mounts it and analyzes it, determining the layer's operating system, and the packages added and removed from the layer. Finally, the result of the analysis is the following: a list of the installed packages in the image, the layers associated with this image and the relationship between them. To store the data from the analysis Clair uses a relational database. However, this database is not available on the Web. Thus it does not comply with any of the five stars for data publishing [2].

3.3 The Docker Ontology

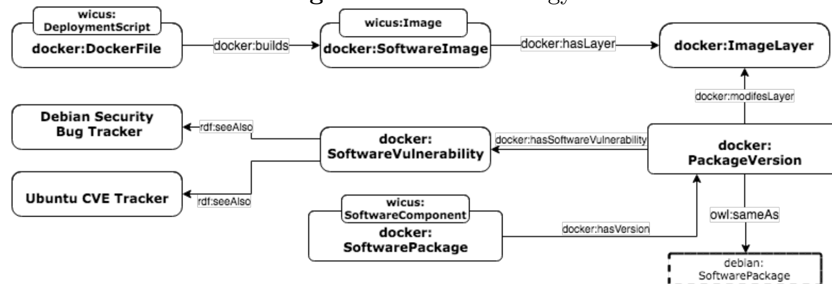
To publish all these data from the previous step in the form of a knowledge graph and link it to the Debian package RDF store we used Morph [9], a Rela-

[3] https://coreos.com/clair/
[4] http://status.quay.io

tional Database to RDF (RDB2RDF [5]) engine. Since the RDF2RDF process needs an ontology, we developed a lightweight ontology to shape the relations between the different concepts we want to publish. The ontology first imports abstract classes from well-known ontologies and relations from the Docker Ontology [8] for some of the Docker concepts and the WICUS ontology [10] for the software experiment reproducibility concepts. A scientific workflow requires a stack of software components, and the researchers must know how to deploy this software stack to achieve an equivalent environment. Thus, we import some abstract classes, and relations from WICUS ontology: (1) SoftwareStack describes the software components that must be installed and their dependencies, (2) DeploymentPlan, DeploymentStep, ConfigurationInfo and ConfigurationParameter classes describe the steps to deploy and configure the software.

Our final ontology is depicted in Figure 2.

Figure 2. Docker ontology.



The main classes in ontology are SoftwareImage (image from which a container is deployed), ImageLayer (a line within the Docker file that install the software within the container), the software packages installed at the ImageLayer (including security vulnerabilities and version packages) and the DockerFile class. DockerPedia resources are identified by the URI `http://dockerpedia.inf.utfsm.cl/resource` while the vocabulary is identified by the URI `http://dockerpedia.inf.utfsm.cl/ontology`. In Listing 1.2 we show the RDF data about the Google's "cadvisor" within Docker.

Listing 1.2. Cadvisor Docker repository from Google

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix docker: <http://dockerpedia.inf.utfsm.cl/vocab#> .
@docker_data: <http://dockerpedia.inf.utfsm.cl/resource/> .
@prefix dcterms: <http://purl.org/dc/terms/> .

docker_data:Repository/447135
    dcterms:description "Analyzes resource usage and performance characteristics of
        running containers." ;
    docker:hasImage docker_data:Image/1139726,
    docker_data:Image/1139926,
    docker_data:Image/1151288 ;
    docker:id 447135 ;
    docker:is_automated "f" ;
    docker:last_updated "2018-01-29T00:31:11.971064-03"^^xmls:dateTime ;
    docker:pull_count 615791788 ;
    docker:user "google" ;
    a docker:Repository ;
    <http://www.w3.org/2000/01/rdf-schema#label> "cadvisor" .
```

4 Related Work

Previously [11] analyzed the security of Docker images from a higher level point of view. However, this work has not published the data from the image analysis (that data mainly includes security vulnerability issues). Other work such as [6], have pointed the difficulty of reproducing scientific contributions due to their high dependence with the computational environment. The authors propose the use of Docker Containers to allow reviewers and future researchers to reproduce the experiments within the same environment. However, they do not provide any means to perform it, and they only express desiderata. Similarly, a different approach in which software ontologies are used is to allow software experiment reproducibility [10]. In that work, the authors present a set of ontologies that model software and hardware components to allow the experiment reproducibility. In [3] the author describes a set of good practices to use Docker for reproducing experiments. One of them is to archive `tarball` snapshots pointing that Docker can rollback layers that have been added to an image, but not revert to the earlier state of a particular layer. We allow the user to revert to that state by providing exact descriptions of each layer. Also, the project Timbus [1] address several issues including the computation environment conservation by means of an annotator. For this purpose, they proposed an extractor to extract and annotate the Software and Hardware components. Nevertheless, the extractor approach of Timbus Project is not adequate to be used in Containers since it increases the complexity of the container. Another related project is Common Workflow Language (CWL, https://www.commonwl.org/), CWL is a specification to describe analysis workflows and tools, as a result, this descriptions are portable and scalable over a variety of software and hardware environments including Docker Containers. However, CWL does not deal with the tasks of describing the resources that define the environment. The tool MyBinder (https://mybinder.org/) allows to generate Docker images from GitHub repositories containing the dependency files from that GitHub repository, however the tool does not describe them. Finally [13] describes how to use RDF to represent Docker files.

5 Conclusions

Throughout this paper, we have presented a resource that allows users to analyze Docker images before they run them into their hosts. We have gathered all (4.5 million) Docker images stored at DockerHub (5TB of data) and analyzed 150,000 of them. The analysis resulted in a dataset with more than 140 million triples, storing data about Docker images, software packages, and their vulnerabilities, links to the Debian package resource description, and the vulnerabilities information pages.

First, we propose a method to conduct logical and physical conservation of the computational environment of an experiment using Docker containers. Second, we achieve logical conservation without spending the considerable amount

of effort annotating the software components and their dependencies. Finally, we rely the physical conservation on DockerHub and their lightweight Docker images.

Acknowledgments The authors has been supported by the Fondecyt Project 11170714.

References

1. J. Barateiro, D. Draws, M. A. Neuman, and S. Strodl. Digital preservation challenges on software life cycle. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 487–490. IEEE, 2012.
2. T. Berners-Lee. Is your linked open data 5 star. *Repéré à https://www. w3. org/DesignIssues/LinkedData. html*, 2010.
3. C. Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, Jan. 2015.
4. A. Dappert, S. Peyrard, C. C. Chou, and J. Delve. Describing and preserving digital object environments. *New Review of Information Networking*, 18(2):106–173, 2013.
5. S. Das, S. Sundara, and R. Cyganiak. R2rml: Rdb to rdf mapping language. w3c rdb2rdf working group, 2012.
6. P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame. The impact of docker containers on the performance of genomic pipelines. *PeerJ*, 3:e1273, 2015.
7. D. Garijo, S. Kinnings, L. Xie, L. Xie, Y. Zhang, P. E. Bourne, and Y. Gil. Quantifying reproducibility in computational biology: the case of the tuberculosis drugome. *PloS one*, 8(11):e80278, 2013.
8. D. Huo, J. Nabrzyski, and C. Vardeman. Smart container: an ontology towards conceptualizing docker. In *International Semantic Web Conference (Posters & Demos)*, 2015.
9. F. Priyatna, O. Corcho, and J. Sequeda. Formalisation and experiences of R2RML-based SPARQL to SQL query translation using MORPH. In *Proceedings of the 23rd international conference on World wide web*, pages 479–490. ACM, 2014.
10. I. Santana-Perez, R. F. da Silva, M. Rynge, E. Deelman, M. S. Pérez-Hernández, and O. Corcho. Reproducibility of execution environments in computational science using semantics and clouds. *Future Generation Computer Systems*, 67:354–367, 2017.
11. R. Shu, X. Gu, and W. Enck. A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 269–280, New York, NY, USA, 2017. ACM.
12. V. C. Stodden. Reproducible research: Addressing the need for data and code sharing in computational science. *Computing in science & engineering*, 12(5):8–12, 2010.
13. R. Tommasini, B. De Meester, P. Heyvaert, R. Verborgh, E. Mannens, and E. Della Valle. Representing dockerfiles in RDF. In *ISWC2017, the 16e International Semantic Web Conference*, volume 1931, pages 1–4, 2017.