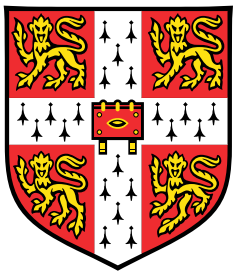
Writing your PhD thesis in LaTeX2e

Using the CUED template



Krishna Kumar

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

King's College

November 2018

I would like to dedicate this thesis to my loving parents ...

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Krishna Kumar
November 2018

Acknowledgements

And I would like to acknowledge ...

Abstract

This is where you write your abstract ...

Table of contents

List of figures

List of tables

Chapter 1

Introducción

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 re-purposing the experimental methods and conditions [1].

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. Conservation refers to the fact of obtaining the same result from an experiment in a different environment [**?**]. 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 [2].

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 focused on conserving the environment of an experiment such as the work in [3] or the Timbus project[1] [4] that focuses on business processes and the underlying software and hardware infrastructure.

The authors in [3] 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

[1] http://www.timbusproject.net/

of the resources in the environment are described using semantic vocabularies to allow a researcher to reproduce an equivalent setting. They 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 workflow computational environment (relying on the chosen infrastructure). However, logical and physical conservation are important to achieve experiment reproducibility.

The reproducibility problem not only happens in the scientific community. IT-companies also face similar problems when they want to distribute any software product into several hosts. To solve it, companies use 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.

In order to reproduce computational scientific experiments by other researchers, it is mandatory to first allow scientists to share these experiments and second to allow to execute them again using the same (or a very similar) computational environment. In this ways a scientist will have guarantees that the experiment she is executing is running using the same environment from which it was created. Thus, it is needed a procedure to guarantee both requirements: to preserve both logical and physical environments to re execute data workflows with reproducibility guarantees.

To allow this reproducibility is necessary thus to provide a way to first describe the scientific experiments and the computational environment in which that experiment was executed. Logical conservation allows to describe the computational environment in which experiment was executed and the physical conservation allows the researchers to rerun an experiment without having to deal with the physical distribution of the experiment and not having to deal with dependencies problems.

Herein we propose a solution to improve the physical and logical conservation solution by using containers. 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. Using Docker, the users can distribute these computational

[2]https://www.docker.com/

environments through software images using a public repository called DockerHub. Thus, we aim to address the physical conservation. In order to achieve logical conservation, we built an annotator system for the Docker Images that describe the workflow management system, as well as their dependencies by developing an annotator system for the Docker images before. In this way, we aim to address the physical conservation. To validate our solution we reproduce 4 different computational experiments. These experiments span different systems, languages and configurations, showing that our approach is generic and can be applied to any computational experiment. We run these experiments, we describe them logically and next we reproduce them based on the logical descriptions we obtained before. To validate the approach we compare the outputs from the experiments.

1.0.1 El desafio de la reproducibilidad

El problema de la reproducibilidad entre distintos tipos de practicas cientificas varia entre una y otra. Por ejemplo, la biología los recursos son muestras de celulas u otro

La utilización de equipamiento también es una limitación

En el caso de experimentos in-silico estas limitaciones no son tan fuertes debido a que los sistemas digitales pueden compartirse y replicar.

Los experimentos computaciones descansan en artefactos digitales para resolver sus

Sistemas computacionales se basan sistemas digitales que son almacenados

En este trabajo, se busca soluciona el este problema, para ello se

mas

En ee

1.1 Experimentos científicos computacionales

La reproducibilidad de los resultados en experimentos es una piedra angular del método científico. Es por ello, que la comunidad científica ha incentivado a los investigadores a publicar sus contribuciones en un forma verificable y entendible [5, 1].

Los términos de reproducibilidad y repetibilidad son utilizados como sinónimos. En este trabajo, utilizaremos las definiciones propuestas por [3], replicabilidad será definida como recreación estricta del experimento original y reproducibilidad es menos restrictivo e implica que puedan existen algunos cambios.

Particularmente, en las áreas de la ciencia donde se ejecutan experimentos in-silico, o sea que realizan a través de computadores o simulaciones, la reproducibilidad requiere que los investigadores compartan el código y los datos de los experimentos realizados con el fin

de que tanto los resultados como el método pueden ser analizados en una forma similar al trabajo original descrito en la publicación asociada a dicho experimento. Para lograr ese objetivo, el código debe estar disponible y los datos deben encontrarse en un formato leíble [6].

1.1.1 Computational Scientific Experiments

Workflows in Science

Loas

conservación fisica y lógica

1.1.2 Scientific Conservation and Reproducibility

1.2 Estructura de tesis

Chapter 2

Estado del arte

2.1 Conservación de procedimiento científicos

Distintas áreas de la ciencia han adoptado técnicas y herramientas para conservar el procedimiento. Por ejemplo, investigadores en bio-informática ha incorporado los workflows para distintos análisis: Doblamiento de proteínas [7], secuencias de DNA y RNA [8, 9] y la detección de ondas gravitacionales [10]. Los workflows científicos son métodos que permiten representar un conjunto de pasos computacionales. Estos pasos pueden ser la obtención de los datos de entrada, transformaciones o generación de los resultados. La representación de los workflow se construye en un lenguaje abstracto para simplificar la complejidad. El conjunto de pasos se pueden representar como gráfos sin ciclos y dirigidos, donde cada paso computacional es representado por un nodo y las dependencias entre los pasos son representado por los arcos. El uso de sistemas de manejo de workflows científicos *Scientific Workflow management Systems (WMS)* permiten diseñar abstractamente, ejecutar y compartir el procedimiento científicos.

Dado que los workflows formalmente describen la secuencia de tareas computacionales y administración de datos, es fácil encontrar el camino de los datos producidos. Un científico podría ver el workflow y los datos, seguir los pasos y llegar al mismo resultado. En otras palabras, la representación del workflow facilita la creación y administración de la computación y además construye una base en la cual los resultados pueden ser validados y compartidos.

Sin embargo, múltiples estudios han mostrado las dificultades reproducir los resultados de los experimentos.

2.1.1 Sistemas de administración de workflows

Cómo se menciono anteriormente, los workflows científicos permiten a los usuarios expresar fácilmente tareas computacionales de varios pasos, por ejemplo, recuperar datos de un instrumento o una base de datos, reformatear los datos y ejecutar un análisis. Un workflow científico describe las dependencias entre las tareas y la mayoría de los casos se describe como un gráfico acíclico dirigido (DAG), donde los nodos son tareas y los bordes denotan las dependencias de las tareas. Una propiedad que define un workflow científico es que gestiona el flujo de datos. Es por ello, que las tareas en un workflow científico varían ampliamente según las necesidades del autor, los tipos pueden ser tanto como tareas cortas en serie o tareas paralelas muy grandes (e.g, utilizando Message Passing Interface - MPI) rodeadas de un gran número de pequeñas tareas en serie utilizadas para el pre y post procesamiento. La interpretación y ejecución de los workflows son manejados por un sistema de manejo de workflows (WMS) que administra la ejecución de la aplicación en la infraestructura. Un WMS puede ser considerado como una capa intermedia necesaria para la abstracción y orquestación de prodecimiento científico. A continuación se describe algunos de los WMS más populares.

Actualmente, existen múltiples WMS que han sido generado por diversas comunidades.

Galaxy: Galaxy (Goecks et al., 2010) is a web-based WMS that aims to bring computational data analysis capabilities to non-expert users in the biological sciences domain. The main goals of the Galaxy framework are accessibility to biological computational capabilities and reproducibility of the analysis result by tracking the information related to every step on the process.

Taverna: Taverna is a Web Service-based WMS, as all the components of the workflow must be implemented as web services (either locally or using an available remote service). Taverna is able to integrate Soaplab26, REST (Fielding, 2000) and WSDL (Christensen et al., 2001) web services. It offers a wide range of services for different processing capabilities, such as local Java services, statistical R processor services, XPath scripts, or spreadsheet import services.

Pegasus: Pegasus (Deelman et al., 2005) is a WMS able to manage workflows comprised of mil- lions of tasks, recording data about the execution and intermediate results. In Pegasus, workflows are described as abstract workflows, which do not contain resource informa- tion, or the physical locations of data and executables.

WINGS: WINGS (Gil et al., 2011) may not be considered as a proper WMS by itself, as it does not provide workflow enactment and execution features. However it is widely

known for workflow design. WINGS can be seen as a top-level and domain-oriented design tool whose workflows can be later enacted in different workflow engines, such as Pegasus or Apache OODT29.

dispel4py: dispel4py (Filguiera et al., 2014) is a Python (Rossum, 1995) library for describing workflows. It describes abstract workflows for data-intensive applications, which are later translated and enacted in distributed platforms (e.g. Apache Storm, MPI clusters, etc.).

2.2 Conservación de equipamiento

Comúnmente el equipamiento en otras disciplinas no es un problema a resolver dado que los recursos utilizados son son conocidos, no-variables y estándares. Por ejemplo, la utilización de probetas, microscopios u otros elementos físicos. Consecuentemente, los investigadores pueden nombrarlos e identificarlos de forma manual en los procedimientos de sus cuadernos de laboratorio. Lo que permite que otro investigador conozca cuáles fueron las herramientas utilizadas en el experimento. Aunque existen excepciones como en la ciencia de biología, donde ciertos recursos son materiales que son utilizados en los procedimientos. En estos casos, los investigadores deben describir los materiales incluyendo información como marcas, composición y otros. En otras ciencias como la astronomía se utilizan recursos de alta tecnología, donde también es necesario documentar las características de hardware y configuraciones utilizadas en el proceso experimental. En las ciencias de la computación sucede un caso similar, dado que los recursos computacionales son una componente requerida en la ejecución del sistema. Es por ello, que esta comunidad no puede ser la excepción respecto a la descripción de los recursos. Por lo tanto, los autores deben poder documentar computadores, clusters, servicios web, componentes de software, etc., en el contexto de sus experimentos.

Diversos trabajos han estudiado el estado actual de la reproducibilidad en las ciencias de computación, en [11] se estudia el factor de decaimiento de un conjunto workflows científicos almacenado en myExperiment [1] que fueron diseñados para el WMS Taverna [12] del área de biología. Para ello, los autores utilizaron cuatro conjuntos de paquetes de workflows y clasifican el decaimiento de los workflows en cuatro categorías: recursos de terceros volatiles, datos de ejemplos faltantes, ambiente de ejecución faltante y descripciones insuficientes sobre los workflow. El estudio muestra que casi el 80% de workflows fallan al ser reproducidos, con un 12% de esos fallos debido al ambiente de ejecución faltante y 50%

[1] https://www.myexperiment.org/home

recursos de terceros volatiles. Estas dos últimas categorías están asociadas a la conservación del ambiente computacional del experimento. En [13], los autores describen un procedimiento para preservar el software, argumentando que el software es frágil a los cambios de ambiente ya sea hardware, sistema operativo, versiones de las dependencias y configuración. Los autores afirman que el software no puede ser preservado con la metodología anterior de sólo mantener su código binario ejecutable. Por ello, introducen el concepto de adecuación de la preservación, una métrica para medir si la preservación de conjunto de funcionalidades de componente de software luego de un proceso reproducción.

De la misma manera, editoriales se ha enfocado en intentar resolver los desafíos en la publicación de trabajos científicos. Por ejemplo, Elsevier formó el *Executable Paper Grand Challenge* para abordar la dificultad de reproducibilidad de los resultados en las ciencias de la computación, ellos argumentan que los bloques vitales y necesarios de información para replicar los resultados -por ejemplo, software, código, grandes conjuntos de datos- no suelen estar disponibles en el contexto de una publicación académica. Y *Executable Paper Grand Challenge* creó una oportunidad para que los científicos diseñen soluciones que capturen esta información y proporcionen una plataforma para que estos datos puedan ser verificados y manipulados. En 2011, [14] se argumenta que el documento de investigación en su estado actual ya no es suficiente para reproducir, validar o revisar completamente los resultados y conclusiones experimentales de un documento. Esto impide el progreso científico. Para remediar estas preocupaciones, presentan Paper Mâché, un nuevo sistema para crear documentos de investigación dinámicos y ejecutables. La principal novedad de Paper Mâché es el uso de máquinas virtuales, que permite a los lectores y revisores ver e interactuar fácilmente con un documento y poder reproducir los principales resultados experimentales. En la misma línea, CernVM [15] propuso la utilización de máquinas virtuales para resolver problemas de reproducibilidad en la ciencia. CernVM es un sistema para el uso de máquina virtuales capaz de ejecutar aplicaciones físicas de los experimentos relacionados al *Large Hadron Collider* (LHC) en el *European Organization for Nuclear Research* (CERN). Su objetivo es proporcionar un entorno completo y portátil para desarrollar y ejecutar el análisis de datos LHC en cualquier ordenador del usuario final (portátil, de sobremesa), así como en la red, independientemente de las plataformas de sistemas operativos (Linux, Windows, MacOS). La motivación del uso de técnicas de virtualización que permite separar los recursos de computación desde la infraestructura subyacente.

Algunos autores han expuesto la necesidad de capturar y preservar el entorno de ejecución de un ejecución del experimento, proporcionando herramientas para analizar y empaquetar los recursos involucrados en él. ReproZip [16] busca captar el conocimiento sobre una infraestructura e intentar reproducirla en un nuevo entorno. Esta herramienta lee los compo-

nentes de infraestructura involucrados en la ejecución (archivos, variables de entorno, etc.) y almacena esta información en una base de datos MongoDB [2]. A continuación se recogen y empaquetan los elementos descritos. Luego, el sistema debe desempaquetar los elementos en otra máquina para reproducir el experimento. Sin embargo, este tipo de enfoque que empaqueta los componentes físicos de una infraestructura determinada presenta limitación en la práctica, debido que los paquetes deben ser ejecutado en una máquina destino similar. Otro ejemplo es TOSCA (Topology and Orchestration Specification for Cloud Applications), TOSCA es un ejemplo de las soluciones que han definido sintaxis para describir la ejecución de los ambientes computacionales. TOSCA es un lenguaje de código abierto utilizado para describir las relaciones y dependencias entre servicios y aplicaciones que residen en una plataforma de computación. TOSCA puede describir un servicio de computación en nube y sus componentes y documentar la forma en que están organizados y el proceso de orquestación necesario para utilizar o modificar dichos componentes y servicios. Esto proporciona a los administradores una forma común de gestionar aplicaciones y servicios en la nube, de modo que esas aplicaciones y servicios puedan ser portátiles a través de las diferentes plataformas de los proveedores de cloud computing. Otro esfuerzo importante relacionado a nuestro trabajo incluye la descripción de los ambientes computaciones utilizando ontologías es TIMBUS. El proyecto se focaliza en preservar procesos de negocios y su infraestructura computacional. Para ello, propusieron un extractor para extraer y anotar los componentes de Software y Hardware, éstas anotaciones son almacenadas según un conjunto de ontologías con el objetivo de gestionar la preservación y reejecución de los procesos de negocio. Sin embargo, el enfoque extractor del Proyecto Timbus no es adecuado para ser utilizado en cualquier sistema de virtualizada ya que aumenta la complejidad del ambiente y generando ruido. Además, exige ejecutar el ambiente computacional lo cual conlleva altos costos computacionales, disminución en la escalabilidad y creación de brechas de seguridad.

En el enfoque de describir los recursos computacionales, los autores en [3] identificaron dos enfoques para conservar el ambiente computacional de un experimento científico: la conservación física, donde los objetos de investigación dentro del experimento se conservan en un entorno virtual; y la conservación lógica, donde las principales capacidades de los recursos en el entorno se describen utilizando vocabularios semánticos para permitir al investigador reproducir un entorno equivalente. Para ello, definieron un proceso para documentar la aplicación de flujo de trabajo y su sistema de gestión relacionado, así como sus dependencias. Además, los autores propusieron *The Workflow Infrastructure Conservation Using Semantics ontology* (WICUS). WICUS es una red de ontologías OWL2 (Web Ontology Language) que implementan la conceptualización de los principales dominios de una infraestructura

[2]https://www.mongodb.com/es

computacional. Como: Hardware, Software, Workflow y Recursos Informáticos. Los autores argumentan este trabajo debido que a que el workflow científico requiere un conjunto de componentes de software, y los investigadores deben saber cómo desplegar esta pila de software para lograr un entorno equivalente. Sin embargo, este proceso se realiza de forma de manual, dejando mucho trabajo a los científicos. Además, los autores afirman que la conservación de los ambientes computacionales comúnmente se logra utilizando un enfoque físico dado que la conservación física permite compartir fácilmente un ambiente computacional con otros investigadores y ellos pueden reproducir el experimento utilizando en el mismo ambiente. Sin embargo, los esfuerzos necesarios para mantener la infraestructura son altos y no hay garantías que no sufran un proceso de decaimiento [17]. Consecuentemente, la mayoría de los trabajos dejan fuera del ámbito de aplicación la conservación física del entorno informático del workflow (basándose en la infraestructura elegida). Pese a que la conservación lógica y física son deseadas para lograr la reproducibilidad del experimento.

En diversos trabajos [18–21] se ha propuesto la utilización de Docker como un reemplazo al uso de máquinas virtuales como ambiente computacionales científicos, los trabajos argumentan que Docker presenta beneficios de portabilidad, documentación precisa de la instalación y configuración, manejo de control de versiones de las imágenes y fácil adopción por desarroladores. Un ejemplo de uso de Docker para la reproducibilidad es BioContainers [18]. BioContainers es un framework de código abierto y orientado a la comunidad que proporciona entornos ejecutables independientes de la plataforma para el software de bioinformática. BioContainers permite a los laboratorios instalar fácilmente software de bioinformática, mantener múltiples versiones del mismo software y combinar herramientas de análisis. BioContainers se basa en los populares proyectos de código abierto Docker [3], Singularity [4] y rkt [5], que permiten que el software sea instalado y ejecutado bajo un entorno aislado y controlado. Sin embargo, en [20, 22] los autores exponen que Docker no controla que paquetes instalados en las imágenes y no existe una descripción completa de los componentes de la imagen y consecuente del contenedor. Por lo tanto, las imágenes Docker funcionan como una caja negra, lo que significa que los usuarios saben cuál es el paquete que se ejecuta dentro del contenedor pero no conocen las versiones o los otros paquetes necesarios para ejecutarlo.

Respecto a la descripción de los componentes de una imagen Docker, en [23] analizó más de 300.000 imágenes de Docker almacenadas en el repositorio oficial de Docker. Los autores encontraron que en promedio las imágenes que contiene el Docker Hub (sistema de almacenamiento de imágenes) son más de 180 vulnerabilidades, siendo la raíz de tal cantidad

[3] https://www.docker.com/
[4] https://www.sylabs.io/singularity/
[5] https://coreos.com/rkt/

de vulnerabilidades el hecho de que muchas imágenes no han sido actualizadas en varios días y que muchas de estas vulnerabilidades se propagan de imágenes de padres a hijos. Los autores encontraron correlaciones entre las imágenes más influyentes y los paquetes vulnerables mejor clasificados, lo que sugiere que la fuente de esa cantidad de vulnerabilidades era probablemente el resultado de la propagación de un pequeño número de imágenes populares (debido a la falta de actualización de las imágenes principales). Los autores utilizaron el software Clair [6] de la empresa CoreOS[7]. En términos de ingeniería ontológica, los autores en [24] presentan la ontología Smart Container que extiende DOLCE [25] y modela Docker en términos de sus interacciones para desplegar imágenes. Otro trabajo relacionado, [26] describe cómo usar RDF para representar archivos de construcción de Docker.

2.3 Docker

Durante los años el uso de tecnologías de virtualización ha aumentado rápidamente, esta tecnología permite dividir el sistema de un computador en múltiples ambientes virtuales. Los importantes beneficios que entrega esta tecnología ha hecho que la virtualización sea muy utilizada [**?**]. Uno de los usos comunes para esta tecnología es la virtualización de servidores en *datacenters*. Con la virtualización de servidores, un administrador de sistemas puede crear una o más instancias virtuales o máquinas virtuales (VMs) en un servidor y hoy en día se utiliza comúnmente en *datacenters* y también en plataformas de *cloud* como Amazon EC2, RackSpace, Dreamhost y otros [**?**]. El crecimiento el uso de la virtualización ha hecho necesario la búsqueda de una solución que permita tener un ambiente escalable y seguro. Un gran numero de soluciones han nacido en el mercado y se pueden clasificar en dos tipos: *container-based virtualization* y *Virtualización basada en hipervisores*.

Container-based virtualization es una virtualización liviana a nivel de software usando el kernel de host para correr múltiples ambientes. Estos ambientes son nombrados con *containers* (contenedores). Hoy en día Linux-VServer, OpenVZ, libcontainer y Linux Container (LXC) son las principales implementaciones para utilizar contenedores. En la figura 2.1 se puede observar que la arquitectura de *container-based virtualization* que trabaja con un sistema operativo compartido con el host por lo tanto no es necesario cargar *n* veces el sistema operativo por *n* containers. Para el sistema operativo *host* el *container* es un proceso más que corre encima del kernel. Gracias al kernel y otras herramientas proveen un ambiente aislado con los recursos limitados necesarios para ejecutar las aplicaciones en el *container* [**?**].

[6]https://github.com/coreos/clair
[7]https://coreos.com/

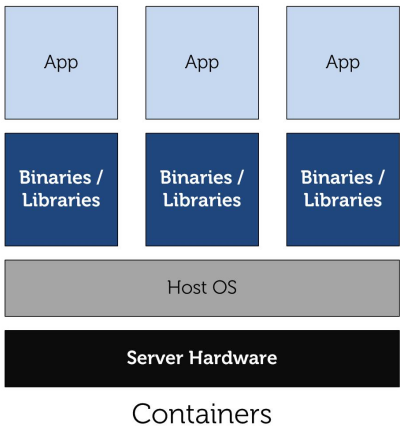


Fig. 2.1 Container-based virtualization.

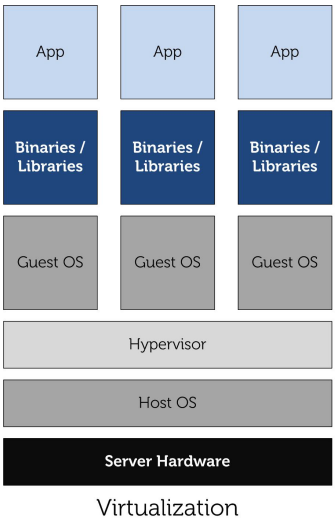


Fig. 2.2 Hypervisor-based virtualization.

Por otra parte *hypervisor-based* establece una máquina virtual (*virtual machine*) encima del sistema operativo, cada máquina virtual (VM) no solamente incluye la aplicación y las dependencias sino además incluye el sistema operativo completo con otro kernel separado.

Las diferencias entre estos dos tipos de virtualización presenta ventajas interesantes para *container-based virtualization* como una mayor densidad de ambientes virtuales en un host debido a que no debe cargar el sistema operativo y puede compartir los binarios y bibliotecas con otros containers en el mismo host. Segundo se ha demostrado que la *container virtualization* es capaz de ser más liviana y eficiente [**? ? ?**]. A partir de esto nace la motivación de estudiar más a fondo soluciones que utilicen los containers y la seguridad de estos.

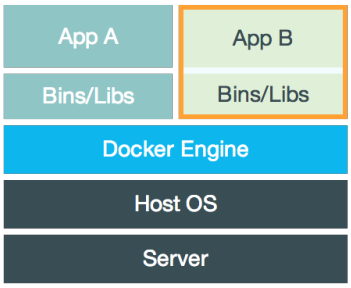


Fig. 2.3 Docker

Docker es un proyecto open-source que utiliza la tecnología de los containers (libcontainer) para “construir, migrar y correr aplicaciones distribuidas". Actualmente utilizado por Yelp, Spotify, entre otros [**?** **?**] Docker es una solución que simplifica el uso de los *containers* que han estado presente durante más de una década. Primero provee una interfaz simple y segura para crear y controlar *containers* [**?**], segundo permite a los desarrolladores empaquetar y correr sus aplicaciones de manera sencilla y además se integra con herramientas terceras que permiten administración y despliegue como Puppet, Ansible y Vagrant. Además que existen diversas herramientas de orquestación como Mesos [**?**], Shipyard [**?**], Kubernetes [**?**], RancherOS [**?**] y Docker Swarm [**?**]. Docker puede separarse en dos grandes componentes:

Dentro de Docker Engine existen componentes:

- Docker Engine
- Docker Images
- Docker Containers
- Docker Registries

Docker Engine es una herramienta liviana y portable para manejar *container-based virtualization* utilizando la arquitectura de la figura 2.3. Los containers corren encima del servicio de Docker que se encarga de ejecutar y manejar los *containers*. Docker Client, provee una interfaz para interactuar con los containers con los usuarios a través de RESTful APIs[**?**].

Docker utiliza una arquitectura de cliente-servidor, Docker Client habla con Docker Daemon y este construye, maneja y corre los *containers*. Docker Client y Docker Daemon pueden correr en el mismo host o se puede conectar el cliente desde un host remoto. El cliente y el Daemon se comunican en forma de sockets o RESTful API [**?**].

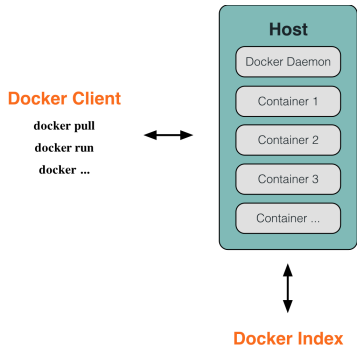


Fig. 2.4 Arquitectura

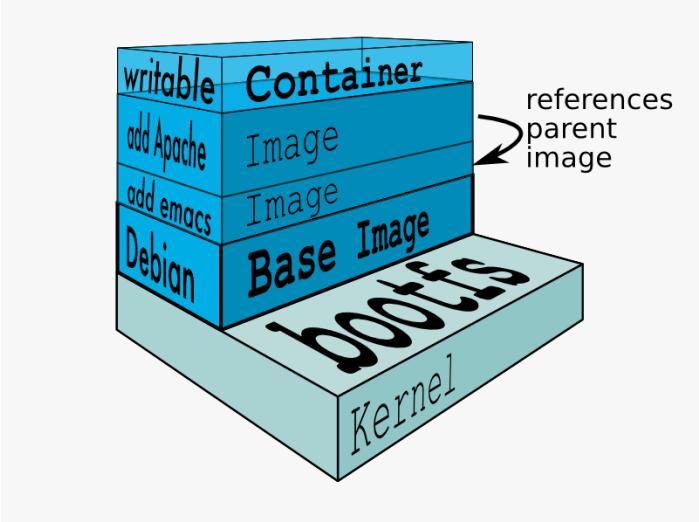


Fig. 2.5 Representación *union file system*

Una imagen de Docker (*Docker Image*) es una plantilla de solo lectura (*read-only template*). Por ejemplo, una imagen puede contener un sistema operativo de Ubuntu con Apache y una aplicación web instalada o simplemente el sistema operativo. Las imágenes son usadas para construir *Docker containers*. Cuando el usuario crea cambios en el *container*, este cambio no se realiza en la imagen, sino que Docker añade una capa adicional con los cambios de la imagen[**?**]. Por ejemplo, si el usuario utiliza una imagen base de Debian, luego añade el paquete emacs y luego añade el paquete apache, el estado de capas estaría representado por la figura 2.5 [**?**]. Esto permite tener un proceso de distribución de imágenes más eficiente dado que solo es necesario distribuir las actualizaciones [**?**]. El*filesystem* descrito anteriormente se denomina *Union File System (UnionFS)*.

Para la definición de ordenes Docker utiliza un archivo para construir las imágenes, este archivo es Dockerfile. Cada Dockerfile es un script compuesto de comandos y argumentos listados sucesivamente para realizar de forma automática acciones en la imagen de base para

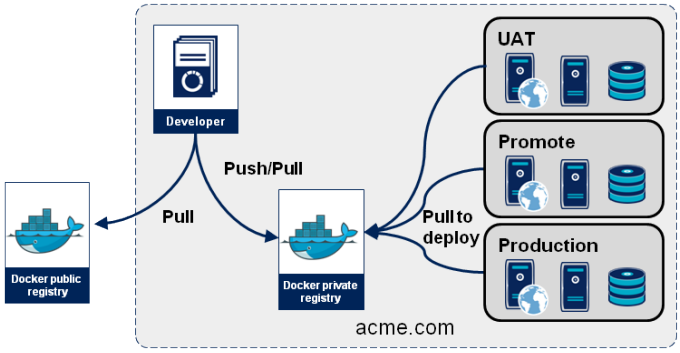


Fig. 2.6 Representación *Dinámica de un Docker registry*

crear una nueva imagen. Esto permite simplificar el despliegue de aplicaciones desde el inicio hasta el final.

Docker *Registries* son repositorios de imágenes, estos pueden ser públicos y privados, vía a estos los usuarios pueden compartir las imágenes personalizadas o bases. Docker cuenta con un repositorio publico llamado Docker Hub, Docker Hub es un repositorio central donde se pueden compartir y obtener imágenes, estas imágenes pueden ser verificadas tanto en la autenticidad y integridad a través un firmado y verificación de datos [**?**]. Los *registries* son parte fundamental de eco sistema de Docker como una herramienta de desarrollo y despliegue de aplicaciones. Un caso de uso de un eco sistema típico que utiliza Docker está descrito por los siguientes eventos y la figura 2.6:

- Devs (desarrolladores) utiliza una imagen lista para usar, ya sea por ejemplo Java, PHP o Rails, está puede ser obtenida de repositorios públicos o privados.
- Paralelamente, ops (operaciones) pueden contribuir añadiendo patrones de seguridad o de despliegue.
- Devs y ops envían y traen sus cambios a través de Docker con comandos como commit, push, pull, tag hasta llegar a una versión de producción
- Ops traen los cambios a los servidores de producción, staging o quality assurance

Un Docker *container* consiste de un ambiente virtual con: sistema operativo, con archivos de usuarios y metadata, cada *container* es construido a partir de una imagen como se mencionó anteriormente. Esa imagen indica a Docker lo que contiene el *container* y asocia un proceso inicial, el cual es un proceso que debe correr cuando el *container* es iniciado. [**?**]. Para describir los pasos incluidos en la creación de un *container*, se ejemplificará con la creación de uno.

```
docker run -i -t ubuntu /bin/bash
```

- *Docker Client* le informa al Docker que debe correr un *container*.
- El comando será el *init 0* del container en este caso en */bin/bash*
- **Traer la imagen:** Docker verifica la existencia de la imagen Ubuntu y sino existe en el host, entonces Docker descarga de un *registry* ya sea privado o publico. Si la imagen existe entonces crea el container.
- **Asignar un *filesystem* y montar una capa *read-write*:** El *container* es creado en el *filesystem* y se añade una capa en modo *read-write* a la imagen.
- **Crear la red y conectar con el *bridge interface*:** Crea la interfaz de red que permite que el Docker *container* pueda hablar con el host a través del bridge (docker0).
- **Asignar un IP:** Asigna una dirección IP del pool al *container*
- **Capturar el *output*, *input* y errores**.

Docker utiliza dos funcionalidades de Linux *namespaces* y *cgroups* para crear ambientes virtuales para los *containers*. *cgroups* o *control groups* proveen un mecanismo de contabilidad y limites de recursos que pueden utilizar los *containers*[**?**]. Los *namespaces* proveen del aislamiento que es llamado *container*. Cuando sea crea un *container*, Docker crea un conjunto de *namespaces* para el *container*. Los *namespaces* utilizados por Docker: mount (mnt) para el manejo del montaje, PID para el aislamiento de los procesos, net para el manejo de interfaces y IPC para acceder a recursos de IPC.

2.3.1 Aislamiento

A isolation

Docker logra el aislamiento de los procesos separando los procesos en *namespaces* y limitando los permisos y la visibilidad de los procesos a otros *containers*. *PID namespaces* (añadido en el kernel $\geq$ 2.6.3.2) es el mecanismo utilizado, logrando que un proceso que se encuentra en el container solo pueda ver procesos que se encuentra en ese container. Por lo tanto un atacante no puede observar procesos de otro container, lo que aisla al *container* este nivel [**? ?**].

Filesystem isolation

Docker usa *mount namespaces* o *filesystem namespaces* para aislar los *filesystems* asociados a los containers. De la misma forma que ocurre con los procesos, los eventos del *filesystem* que ocurre en el container afectan a ese container. Pero existen *Linux kernel file systems* que deben ser montados desde host para que el container pueda funcionar correctamente, esto permite a los containers acceso directo al host. Docker utiliza dos técnicas para proteger el host: primero montar los *file-systems* en modo lectura para evitar que puedan escribir en ellos y eliminar la opción de montar *file-systems* en modo escritura y lectura [**?**].

Device isolation

Docker utiliza *cgroups* que permite especificar que *device* puede ser utilizado con el container. Esto bloquea la posibilidad de crear y usar *device nodes* que puedan ser utilizados para atacar el host. Los *device nodes* que son creados para cada container por defecto: son: /dev/console, /dev/null, /dev/zero, /dev/full, /dev/tty*, /dev/urandom, /dev/random, /dev/fuse. [**?**]

IPC isolation

IPC (*inter-process communication* es un conjunto de objetos para el intercambio de data a través de los procesos, como semáforos, colas de mensajes, segmentos de memoria compartida. Los procesos corriendo en los containers utilizan *IPC namespaces* que permite la creación de un *IPC* separado y independiente para cada container, con esto se previene que procesos en un container interfieran con otros containers o el host.

Network isolation

Para cada container, Docker crea una red independiente usando *network namespaces*, compuesta de su propia IP, rutas, *network devices* Esto permite que el container pueda interactuar con otro host a través de su propia interfaz.

Por omisión, la conexión se realiza gracias al host que provee un *Virtual Ethernet bridge* en la máquina host, llamado docker0 que automáticamente realiza un *forward* de los paquetes entre las interfaces. Cuando Docker crea un nuevo container, esto establece una interfaz de red virtual con un nombre único que se conecta con el *bridge (docker0)* y con la interfaz *eth0* del container [**?**].

Limite de recursos

Cgroups controlan la cantidad de recursos como CPU, memoria, *disk I/O* que el container puede utilizar. A partir de esto se protege contra ataques de DoS.

Linux Capabilities

Los sistemas basado en Unix clasificando los procesos en dos categorías: *privileged processes* (superuser o root) y *unprivileged processes*. El kernel salta la verificación de todos los permisos para los procesos privilegiados. Pero desde el kernel 2.2 los privilegios de los usuarios root o superuser se dividen en *capabilities* las cuales el kernel puede activar o desactivar [**?**].
Docker containers corren en un kernel compartido con el host, en los containers es innecesario activar todas las *capabilities* dado que las aplicaciones no las necesitan [**?**].

Docker is a technology that allows virtualizing a minimal version of an Operating System. Therefore users can run applications within it. Throughout this section, we introduce how Docker and its registry (Docker Hub) work, starting with how Docker images are created and stored in Docker Hub.

2.3.2 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.

2.3.3 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.

2.4 Clair

Chapter 3

Objetivos de trabajo

El principal objetivo de este trabajo es complementar enfoques existentes para la reproducibilidad científica en el área de las ciencias de la computación. Para ello, se propone un nuevo enfoque para conservar el ambiente de ejecución del experimento científico. Hemos identificado problemas abiertos, en orden de definir los objetivos de trabajo. Luego, estos objetivos se ven formalizados por un conjunto de hipótesis. Y además, se define un conjunto de hechos que se asumen para restringir el campo de aplicación de la propuesta.

3.1 Open Research Problems

- Problema 1: La infraestructura computacional utilizada por un workflow científico se encuentra predefinida. Por lo tanto, no existe una definición de los recursos de la infraestructura para ejecutar el experimento. Consecuentemente, pueden existir dificultades para lograr la reproducción del experimento.
- Problema 2: La conservación física de los ambientes computaciones permite mantener y compartir fácilmente el ambiente con la comunidad. Sin embargo, se ha descartado debido a tres problemas: 1. Alta utilización de almacenamiento por parte de las máquinas virtuales, 2. El acceso a los datos almacenados está sujetos a políticas de la organización y 3. Existe un proceso de decaimiento en tiempo. Sin embargo, no se ha estudiado el uso de containers para solucionar el problema.
- Problema 3: Los enfoques actuales anotan los pasos de construcción de los ambientes computacionales de forma indirecta, por lo tanto, recaen en el científico

3.2 Hipótesis

En función a los problemas abiertos detectados, se definen las siguientes hipótesis:

- Un proceso automático puede describir los requerimientos de ambiente computacional y codificarlo en un formato compartible utilizando modelos semánticos.
- La descripción de containers utilizando modelos semánticos permite la reproducción del ambiente de un experimento científico.

3.3 Objetivos

Para enfrentar los problemas abiertos se definen los siguientes objetivos. Estos objetivos permiten la verificación de la hipótesis y ser una guía para el desarrollo.

- Lograr conservación física y lógica de los ambientes computaciones de un experimento usando Containers
- Implementar un proceso automático capaz de leer la descripción del ambiente y especificar uno nuevo.
- Integrar un sistema que permita el despliegue de estos ambientes computacionales en proveedores de infraestructura e instalar el software apropiado basado al plan de despliegue.

3.3.1 Objetivos específicos

- Adaptar y mejorar modelos estándares que describen ambientes computacionales científicos para incluir virtualización basada en containers.
- Designar una framework para anotar los componentes de ambiente del experimento usando modelos semánticos.

3.4 Restricciones

- Ambientes Linux

Chapter 4

Representación de ambiente de ejecución

En este trabajo, nosotros argumentamos que las descripciones de los ambientes computacionales es necesaria para la reproducción del experimento. Además, la información debe ser la suficiente para comparar y detectar las diferencias entre el ambiente original y el reproducido.

Dada que las imágenes Docker son ambientes aislados y independientes, los componentes de software dentro del container están relacionado al experimento y no existen componentes relacionados a otros experimentos. Por esta razón, nosotros aseguramos que las anotaciones no presentarán ruido de otras herramientas o dependencias asociadas. Para realizar una anotación automática de los paquetes instalados, nosotros proponemos, e implementamos un sistema de anotación. El sistema requiere el nombre de una imagen existente en un repositorio de DockerHub y opcionalmente los datos del repositorio Git que almacena el archivo Dockerfile.

Los pasos asociados del sistema anotación:

1. Consultar a repositorio los metadatos de la imagen. Anotar los metadatos utilizando la ontología propuesta.

2. Descargar cada una de las capas asociadas a la imagen Docker.

3. Sistema anotador consulta a Clair, Clair monta cada una de capas de la imagen, detecta los componentes de software instalados utilizando el sistema de paquete.

4. Obtener la información de Clair, anotar los datos obtenidos por Clair según la ontología.

5. Guardar los datos en una base de datos RDF que permita consulta SPARQL.

4.1 Semantic models

En [3], los autores proponen *The Workflow Infrastructure Conservation Using Semantics ontology (WICUS)*. WICUS es ontología OWL2 (Web Ontology Language) que implementa la conceptualización de los principales dominios de la infraestructura computacional. Estas son: Hardware, Software, Workflow y recursos de computo

Los autores definen que los workflows científicos requieren un conjunto de componentes de software, y los investigadores deben conocer cómo desplegar estos componentes para lograr ambiente equivalente. Por lo tanto, nosotros utilizamos algunos clases y relaciones desde la ontología WICUS:

DeploymentPlan: Un plan de despliegue está compuesto de todos los pasos El plan de despliegue permite entender al investigador cuáles fueron los pasos requeridos para construir el ambiente computacional.

DeploymentStep: Cada paso de despliegue es representado por una línea de comando que realiza la instalación, descarga o configuración del ambiente. La información se obtiene desde el archivo DockerFile

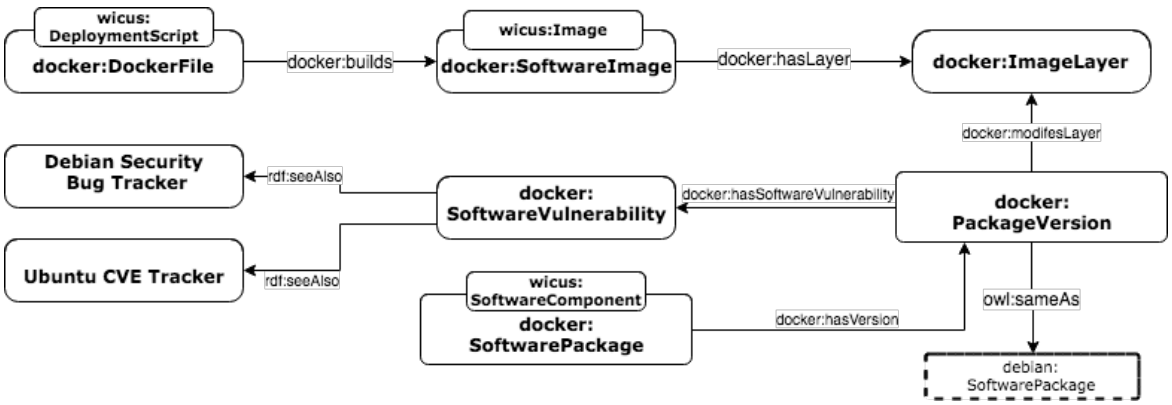
SoftwareComponent: Es un componente de software instalado sin una versión específica.

Definimos `SoftwarePackage` como una subclase de `SoftwareComponent` para definir los componentes instalados por el gestor de paquetes del sistema operativo Cada `wicus:SoftwareComponent` tiene una relación `dockerpedia:hasVersion` o `dockerpedia:isVersionOf` que muestra la versión del software instalado. El recurso `dockerpedia:PackageVersion` puede estar afectado por vulnerabilidades representadas por el recurso `dockerpedia:SoftwareVulnerability` y versiones que reparan la vulnerabilidad `dockerPedia:SecurityRevision`. Una imagen Docker se representa por `dockerpedia:SoftwareImage` y sus capas por el recurso `dockerpedia:ImageLayer`. Los paquetes instalados en la imagen están relacionado por `vocab:ContainsSoftware` con su imagen. Finalmente, el sistema operativo es representado por recurso `dockerpedia:OperatingSystem` y cada imagen y capa de la imagen están asociados al recurso.

Una versión completa se encuentra disponible en nuestros repositorios [1] y la versión resumida está representada en la figura 4.1.

[1] https://github.com/dockerpedia/ontology/

Fig. 4.1 Docker ontology.



4.2 Annotator

El servicio de anotación utiliza una interfaz REST para recibir la imagen Docker del experimento científico con la información del experimento. El sistema anotación realiza dos tipos de anotaciones: Componentes de software y pasos de construcción

4.2.1 Anotaciones de los pasos de construcción

Anotamos el plan de despliegue (Deployment Plan) usando dos métodos. El primero método obtiene los pasos desde el archivo Dockerfile entregado por el usuario, esto permite obtener los pasos y la ubicación de archivo en el repositorio. Usando este método se asegura la reconstrucción del ambiente debido a que cualquier archivo necesario por Dockerfile se encuentra en el repositorio Git. Sin embargo, un usuario puede construir una imagen sin compartir el archivo Dockerfile. []. reporta que el 30% de las imágenes Docker en DockerHub enlanza su archivo Dockerfile. Por lo tanto, nosotros extraemos la información utilizando el manifesto de la imagen Docker. Según [2], el manifiesto de la imagen provee la configuración y el conjunto de las capas de la imagen

Algunos atributos importantes son:

name: *string* nombre de la imagen

tag: *tag* version de la imagen

architecture: *string* arquitectura del servidor en cuál la imagen ha sido construido. Esta información actualmente no es utilizada por Docker.

[2]https://docs.docker.com/registry/spec/manifest-v2-2/

fsLayers: *array* lista de las capas que componente la imagen. La estructura contiene los siguiente campos:

blobSum: es un identificador utilizando una función de hash sha256 para cada capa de la imagen

history : *array* Es una lista de datos históricos no estructurados para la compatibilidad con la v1. Contiene el ID de la capa de imagen y el ID de las capas principales de la capa. El historial es una estructura que consta de los siguientes campos:

v1Compatibility : *string* V1Compatibilidad es la información de compatibilidad de V1 en bruto. Esto contiene el objeto JSON que describe la V1 de esta imagen. Una V1Compatibilidad es una estructura que consta de los siguientes campos:

Id: *string* ID de la capa utilizando hash sha256

Parent: *string* ID de la capa madre utilizando hash sha256

ContainerConfig: *string* El comando que construyó la capa

Apache Jena es utilizado para almacenar y interactuar con los datos. Apache es un framework Java gratuito y de código abierto para la construcción de aplicaciones de Web Semántica y Datos Enlazados. El sistema anotador construye los triples relacionado con el experimento, imagen, capas, componentes de software y vulnerabilidades y se envían usando con la API de Apache Jena que serializa los triples utilizando formatos populares como RDF/XML o Turtle. Esta decisión permite realiza cambios a otra herramienta en caso que Apache Jena no cumpla con los requerimientos de escalabilidad.

4.2.2 Anotaciones de componentes de software

Para lograr la anotación de los componentes de software se utiliza gestores de paquetes del sistema, un gestor de paquete es una colección de herramientas de software que automatizan la instalación, actualización, configuración y eliminación de componentes de software. Los gestores que paquetes se clasifican en dos tipos: gestor de paquetes del tipo sistema y general:

Gestor de paquetes de sistema: son aquellos vinculados al sistema operativo (e.g., apt de familia Debian, yum de familia RedHat).

Gestor de paquetes generales: son aquellos externos que normalmente son utilizados para instalar un componentes de software de tercero ó un lenguaje especifico (e.g., pip, conda, npm). Conda es un sistema paquete frecuentemente utilizado por investigadores al estar relacionado con Jupyter Notebook.

4.2.3 Desafios

La descripción de los componentes de software es fundamental para realizar cuantificar la similitud entre dos o más ambientes computacionales. Un enfoque común para detectar los componentes de software guardar o detectar los comandos que realizan la instalación de software. Por ejemplo, la figura 4.1 muestra los comandos para instalar los componentes de la imagen TensorFlow.

Listing 4.1 Ejemplo de instalación de dependencias para la imagen TensorFlow

```
apt-get install -y --no-install-recommends \
        build-essential \
        curl \
        libfreetype6-dev \
        libhdf5-serial-dev \
        libpng12-dev \
        libzmq3-dev \
        pkg-config \
        python \
        python-dev \
        rsync \
        software-properties-common \
        unzip
```

El enfoque detectaría los componentes: build-essential, curl, libfreetype6-dev, libhdf5-serial-dev, libpng12-dev, libzmq3-dev, pkg-config, python, python-dev, rsync, software-properties-common y unzip. Sin embargo, el enfoque no obtiene información sobre las versiones o las dependencias del software. Utilizando el enfoque propuesto, se puede terminar que el línea anterior instala 184 paquetes.

Considerando la popularidad de Jupyter Notebook y conda en la comunidad científica, nosotros extendemos Clair para detectar los paquetes instalados por Conda en la imagen. Nuestra propuesta incluye la extensión de Clair para detectar los componentes de software instalados por Conda

En la figura se muestra algunos de los paquetes instalado en la imagen de un workflow construido con Pegasus 4.2, en la figura 4.3 se muestra una vulnerabilidad de paquete glibc instalado en la imagen Pegasus.

Listing 4.2 Ejemplo de triples que muestran los componentes de software de la imagen Pegasus

```
<http://dockerpedia.inf.utfsm.cl/resource/SoftwareImage/
    dockerpedia-pegasus_workflow_images%3Apegasus-4.8.5>
        vocab:containsSoftware  <http://dockerpedia.inf.utfsm.cl
            /resource/PackageVersion/libxcb-1.11.1-1ubuntu1>, <
            http://dockerpedia.inf.utfsm.cl/resource/
            PackageVersion/cryptsetup-2%3A1.6.6-5ubuntu2.1>  , <
            http://dockerpedia.inf.utfsm.cl/resource/
            PackageVersion/e2fsprogs-1.42.13-1ubuntu1>  , <http://
            dockerpedia.inf.utfsm.cl/resource/PackageVersion/dbus
            -glib-0.106-1>  , <http://dockerpedia.inf.utfsm.cl/
            resource/PackageVersion/libxmu-2%3A1.1.2-2>
```

Listing 4.3 Ejemplo de triples de la vulnerabilidad de glibc

```
<http://dockerpedia.inf.utfsm.cl/resource/SoftwareVulnerability/
    CVE-2018-6485>
vocab:affectOS      <http://dockerpedia.inf.utfsm.cl/resource/
    OperatingSystem/ubuntu%3A16.04>  , <http://dockerpedia.inf.
    utfsm.cl/resource/OperatingSystem/debian%3A9>  ;
vocab:description



                                                              ;
vocab:link
                    ,
                        ;
vocab:severity              .
```

Chapter 5

Experimentación y evaluación

5.1 Experimentos computacionales

Para validar nuestra propuesta y su aplicación en el contexto de workflows científicos, se ha seleccionado un subconjunto de workflows y WMSs. El subconjunto ha sido seleccionado bajo los criterio de: nivel de utilización de WMS, el lenguaje utilizado en él, y disponibilidad de los materiales del workflow. Los materiales deben incluir los datos de ingresos que permitan reproducir de forma completa el workflow, documentación, resultados o anotaciones. Además nos debe permitan comparar los resultados de nuestro ambientes resultantes.

Para cada WMS, hemos construimos una imagen estándar. En consecuencia, un investigador puede importar esta imagen e instalar los componentes de software relacionados con el workflow. Esto se puede conseguir utilizando la instrucción FROM de Dockerfile. Por ejemplo, el workflow de SoyKB utiliza el workflow manager system: Pegasus. Por lo tanto, la imagen SoyKb importa la imagen Pegasus. Para entender los requisitos del workflow, nosotros inspeccionamos la documentación disponible y las anotaciones generadas por WICUS [3]. En caso que el WMSs no distribuya su software utilizando Docker, se construye las imágenes, los archivos necesarios como configuraciones y el DeploymentPlan (Dockerfile). Estos archivos están disponibles en nuestros repositorios para cada flujo de trabajo. [1]. Además, cada DeploymentFile incluye información sobre la imagen utilizando el estándar de la Open Container Initiative [2].

[1] https://github.com/dockerpedia
[2] https://www.opencontainers.org/

5.1.1 Pegasus

Pegasus [] es un WMS capaz de gestionar flujos de trabajo compuestos por millones de tareas, registrando datos sobre la ejecución y resultados intermedios. Cuando se producen errores, Pegasus intenta recuperarse cuando es posible reintentando tareas, reintentando todo el flujo de trabajo, proporcionando puntos de control a nivel de flujo de trabajo, remapeando partes del flujo de trabajo, probando fuentes de datos alternativas para la puesta a disposición de los datos y, cuando todo lo demás falla, proporcionando un flujo de trabajo de rescate que contiene sólo una descripción del trabajo que queda por hacer. Limpia el almacenamiento a medida que se ejecuta el flujo de trabajo, de modo que los flujos de trabajo de datos intensivos tengan suficiente espacio para ejecutarse en recursos limitados por el almacenamiento]. Pegasus lleva un registro de lo que se ha hecho (procedencia), incluyendo las ubicaciones de los datos utilizados y producidos, y qué software se utilizó con qué parámetros. Pegasus lee las descripciones del flujo de trabajo de los archivos DAX. El término DAX" es la abreviatura de "Directed Acyclic Graph in XML". DAX es un formato de archivo XML que tiene sintaxis para expresar trabajos, argumentos, archivos y dependencias. Para crear un DAX es necesario escribir código para un generador de DAX. Pegasus opcionalmente utiliza en HTCondor como administrador de tareas. Por lo tanto, construimos dos versiones para la imagen de Pegasus; una versión tiene ins talado el paquete condor y otra sin él. La justificación de decisión recae en permitir a los científicos utilizar una imagen simple si es necesario. El paquete Pegasus ha sido obtenido del repositorio oficial [3]. Los principales requisitos de pegasus son: Java, (la versión Java depende de la versión pegasus), python y perl. La figura **??** muestra las dependencias especificadas tanto por el sistema de paquetes o documentación.

Las imágenes se encuentran disponibles en DockerHub [4]

Soybean Knowledge Base

El workflow SoyKB (Soybean Knowledge Base) [27] permite realizar un proceso de resecuencia de germoplasma de la soya, ésta ha sido seleccionada para estudiar rasgos como aceites, proteina, resistencia de los nematodos del quiste de la soya y resistencia al estrés. Pegasus presenta un workflow que implementa polimorfismo de nucleótido único (SNP), la operación insertar/remover (indel) de la base del genoma de un organismo y una análisis utilizando el software GATK [5] La figura 5.2 muestra una representación gráfica del workflow,

[3] http://download.pegasus.isi.edu/wms/download/debian
[4] https://hub.docker.com/r/dockerpedia/pegasus_workflow_images/
[5] https://software.broadinstitute.org/gatk/

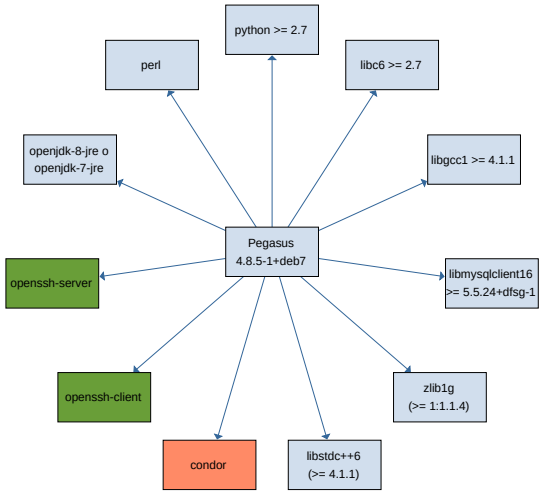


Fig. 5.1 Dependencias de Pegasus. En azul: dependencias necesarias y especificadas en el sistema de paquetes. En verde: necesarias pero no especificadas en el sistema de paquetes. En naranjas: dependencias recomendadas

donde se realiza un análisis en paralelo de las muestras para que sean alineadas con el genoma de referencia y se identifique los indels y SNPs. Luego fusiona y filtra los resultados.

Los componentes de software utilizados por SoyKB se clasifican en dos tipos: propio (en amarillo) y de terceros (en verde). Los componentes principales son: bwa, gatk y bwa y el software de tercero java-1.7.0

La evaluación de los resultados se realizó de forma manual al igual que en [3] debido a los pasos aleatorios del workflow. La metodología fue una verificación manual de la estructura de los resultados, el tamaño de los archivos, número de líneas y la inexistencia de errores.

Montage

Montage es un set de herramientas creadas por *NASA Infrared Processing and Analysis Center (IPAC)*, permitiendo generar bajo demanda mosaicos de imágenes astronómicas personalizadas, a partir de archivos de entrada que cumplen con el estándar del Sistema de Transporte Flexible de Imágenes (FITS) y que contienen datos de imágenes registrados en proyecciones que cumplen con los estándares del Sistema Mundial de Coordenadas (WCS).

La figura 5.4 entregada por Pegasus ilustra el workflow Montage. Cada uno de los nodos de la figura es un software binario que generan la imagen final.

Debido a que el software es un binario, no se encuentra empaquetado. Por lo tanto fue descargado de la fuente. La dirección de descarga se encuentran en el plan de despliegue [6].

[6]https://github.com/dockerpedia/montage/blob/master/Dockerfile

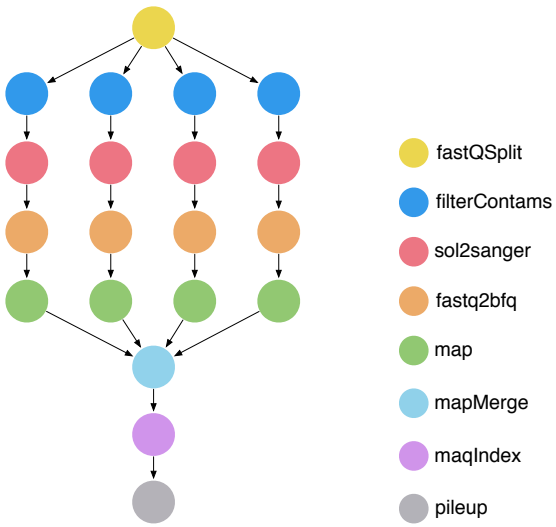


Fig. 5.2 Representación entregada por Pegasus del workflow SoyKB.

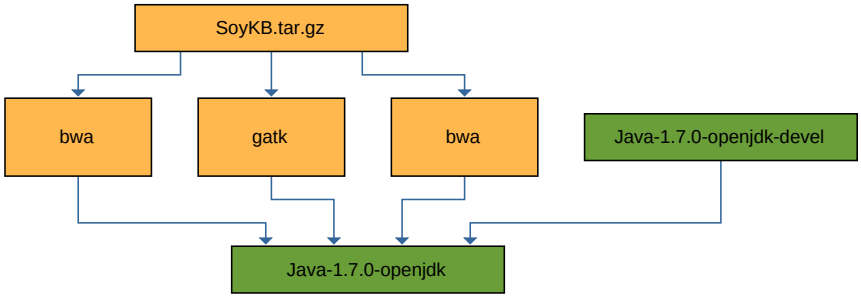


Fig. 5.3 Dependencias de SoyKb. En amarillo: software propio y en verde software de terceros

En el caso de Montage, se produce una imagen como salida, por ello utilizamos la herramienta de hash perceptual [7] para realizar la comparación entre la imagen reproducida (imagen del cielo de 0,1 grados) frente a la original. Como resultado, se obtiene un factor de similitud de 1,0 (más de 1,0) con un umbral de 0,85. Las figuras 5.5a y **??** muestran las imágenes resultantes y los archivos resultantes en formato FITS se encuentran en nuestro repositorio [8].

[7] http://phash.org
[8] https://github.com/dockerpedia/montage_results

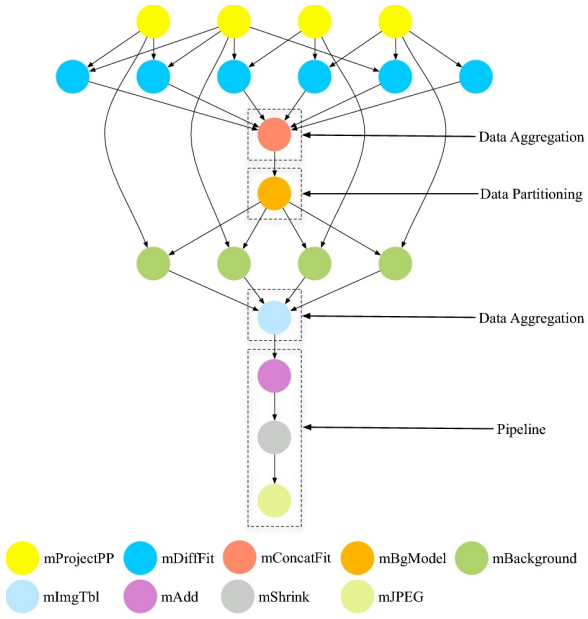


Fig. 5.4 Representación entregada por Pegasus del workflow Montage.

5.1.2 dispel4py

dispel4py [] es una biblioteca Python para describir workflow. Describe flujos de trabajo abstractos para aplicaciones intensivas, que luego se traducen y ejecutan en plataformas distribuidas (por ejemplo, Apache Storm, clusters MPI, etc.). El paquete dispel4py ha sido obtenido del repositorio oficial. Las imágenes están disponibles en DockerHub [9]. Para la instalación el paquete, utilizamos Conda, un gestor de paquetes, dependencias y entornos para cualquier lenguaje Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, FORTRAN y es ampliamente utilizado en entornos de *Jupyter notebook*. Para asegurar la versión de los paquetes que se instalarán, incluimos la lista completa de los paquetes instalados en el repositorio GitHub

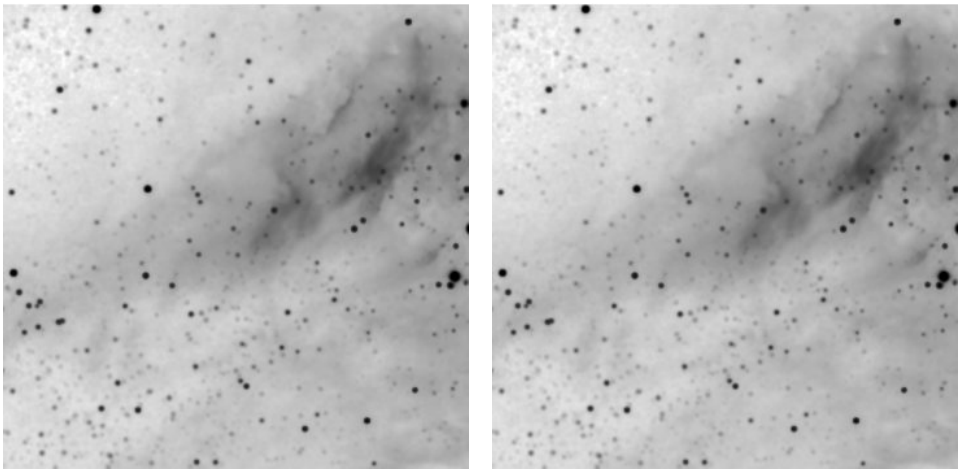
Los principales requisitos de dispel4py son: python2.7, git y python-setuptools

Las imágenes se encuentran disponibles en DockerHub [10]

Internal extinction

Internal Extinction of Galaxies Workflow calcula la extinción interna de galaxias desde el catalogo Amiga. Estos datos son obtenidos a partir de un servicio de Observatorio Virtual, una red de herramientas y servicios que implementan estándares publicados por la *International Virtual Observatory Alliance* (IVOA). El workflow calcula la propiedad, que representa la

[9] https://hub.docker.com/r/dockerpedia/internal_extinction/
[10] https://hub.docker.com/r/dockerpedia//



(a) Resultados originales obtenidos desde WICUS

(b) Resultados reproducidos por nuestra propuesta

Fig. 5.5 Los resultados obtenidos con el nuevo ambiente son iguales

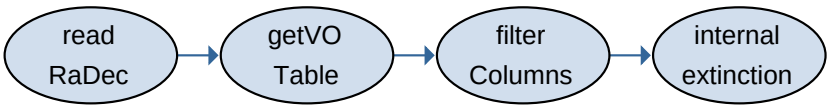


Fig. 5.6 Pasos necesarios para el workflow internal extinction

extinción de polvo dentro de las galaxias y que es un coeficiente de corrección necesario para calcular la luminosidad óptica de una galaxia.

El workflow primero lee el archivo de entregada que contiene la declinación y ascensión recta de 1051 galaxias. Luego, utiliza los valores para realizar consultas al observatorio virtual. Los valores resultantes de las consultas son filtrados seleccionando sólo los valores que correspondan al tipo morfológico (Mtype) y al rango de los ejes del isófito 25 $mag/arcsec^2$ (logr25) de las galaxias. Finalmente, se calcula su extinción interna. La figura 5.6 muestra los pasos del workflow

Nuestra investigación inicial sobre las dependencias para la ejecución de Internal extinction muestra que el software principal requeridos es el siguiente: `requests==2.20.0`, `python=2.7.15=h33da82c_4`, `numpy=1.15.0=py27h1b885b7_0` y `astropy==2.0.9`

Seismic Ambient Noise Cross-Correlation

Seismic Ambient Noise Cross-Correlation workflow (o xcorr workflow) es parte del proyecto *Virtual Earthquake and seismology Research Community e-science environment in Europe*

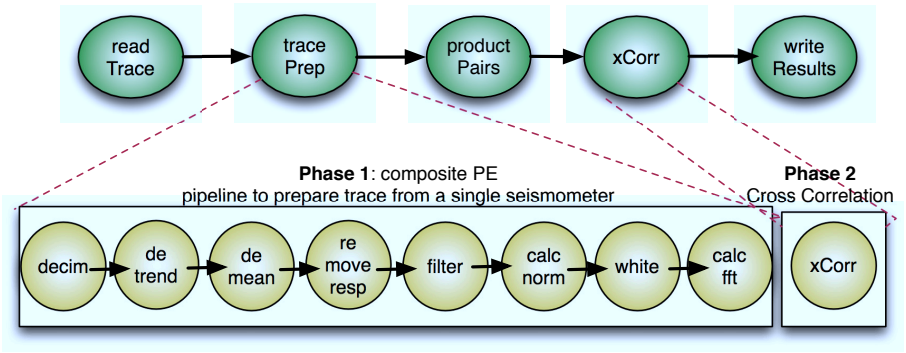


Fig. 5.7 Pasos necesarios para el workflow internal extinction

(VERCE). Los terremotos y las erupciones volcánicas en ciertos casos van precedidos o acompañados de cambios en las propiedades geofísicas de la Tierra, como la velocidad de las olas o la velocidad de los eventos. El desarrollo de métodos fiables de evaluación de riesgos para estas amenazas requiere un análisis en tiempo real de los datos sísmicos y un pronóstico verdaderamente prospectivo y pruebas para reducir los sesgos. El objetivo del workflow es la prevision de riesgo a partir del pre procesamiento y correlación de forma cruzada las propiedades anteriores xcorr presenta dos etapas, la primera etapa es un preproceso donde cada serie de tiempo continua de una estación sísmica dada (llamada traza), está sujeta a una serie de tratamientos. El procesamiento de cada traza es independiente de otras trazas, haciendo que esta fase sea paralela y la segunda etapa empareja todas las estaciones y calcula la correlación cruzada para cada par. La figura 5.7 muestra los pasos del workflow.

El software básico para la ejecución de Internal extinction es: `python=2.7.15=h33da82c_4`, `obspy=1.1.0=py27h39e3cac_2` y `numpy=1.15.0=py27h1b885b7_0`

5.1.3 WINGS

WINGS es un sistema de flujo de trabajo semántico que ayuda a los científicos en el diseño de experimentos computacionales. Un experimento computacional especifica cómo deben ser procesados los conjuntos de datos seleccionados por una serie de componentes de software en una configuración particular. Los principales requisitos de pegasus son: Docker y Tomcat. WINGS a diferencia de los otros sistemas de workflows utiliza imágenes Docker para su distribución. La imagen de sistema se encuentra disponible en [11]

Los principales requisitos de pegasus son: Java 1.8, Tomcat 8.5, Docker. La figura 5.1 muestra las dependencias especificadas tanto por el sistema de paquetes o documentación.

[11] https://hub.docker.com/r/dockerpedia/wings-docker/

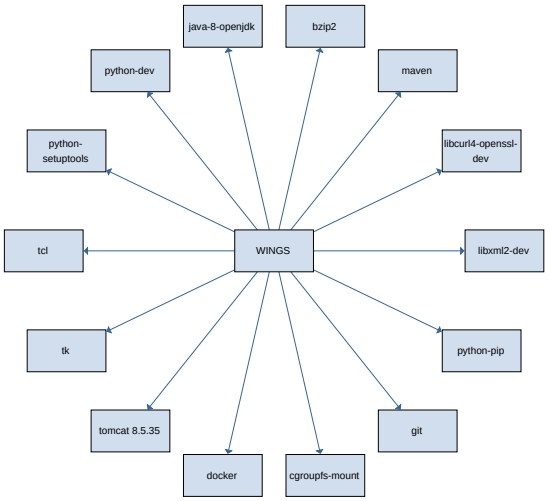


Fig. 5.8 Dependencias de WINGS

Las imágenes se encuentran disponibles en DockerHub [12]

MODFLOW-NWT

MODFLOW es el modelo hidrológico modular del *United Satatus Geological Survey* (USGS). MODFLOW se considera una norma internacional para simular y predecir las condiciones de las aguas subterráneas y las interacciones entre las aguas subterráneas y las aguas superficiales. El USGS MODFLOW-NWT es una formulación de Newton-Raphson para MODFLOW-2005 para mejorar la solución de problemas de flujo de aguas subterráneas no confinadas. MODFLOW-NWT es un programa independiente destinado a resolver problemas de secado y rehumectación de las no linealidades de la ecuación de flujo de agua subterránea no confinada.

La figura 5.9 muestra la estructura principal del worlklow, desarrollado con pipeline compuesto por tres pasos. El proceso inicia leyendo el modelo a utilizar. Luego, el archivo de zona se usa para especificar arreglos de zonas que van usarse y finalmente se genera una visualización que muestra la cantidad de millones de galones por día en zona. Las figuras 5.10a y 5.10b muestran los resultados generados por el ambiente original y reproducido respectivamente.

[12]https://hub.docker.com/r/dockerpedia/wings/

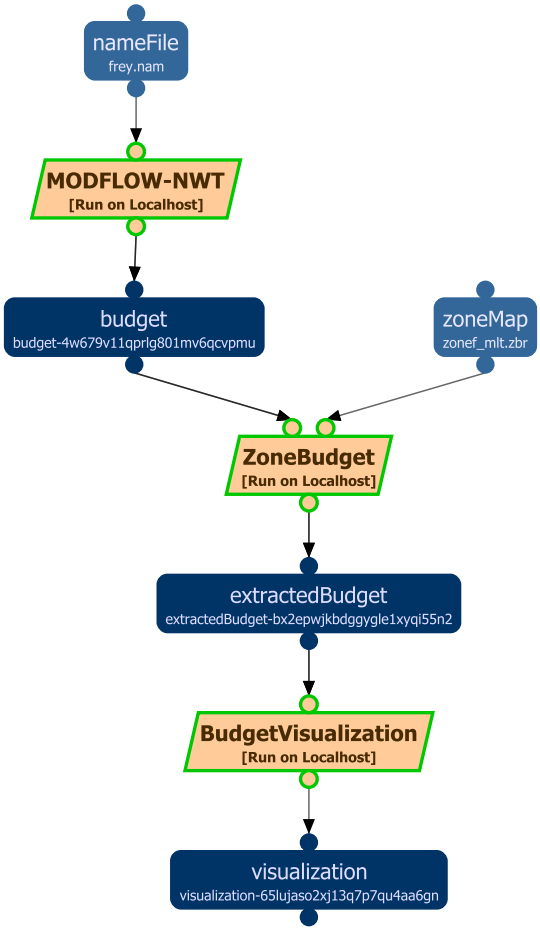


Fig. 5.9 Figure caption.

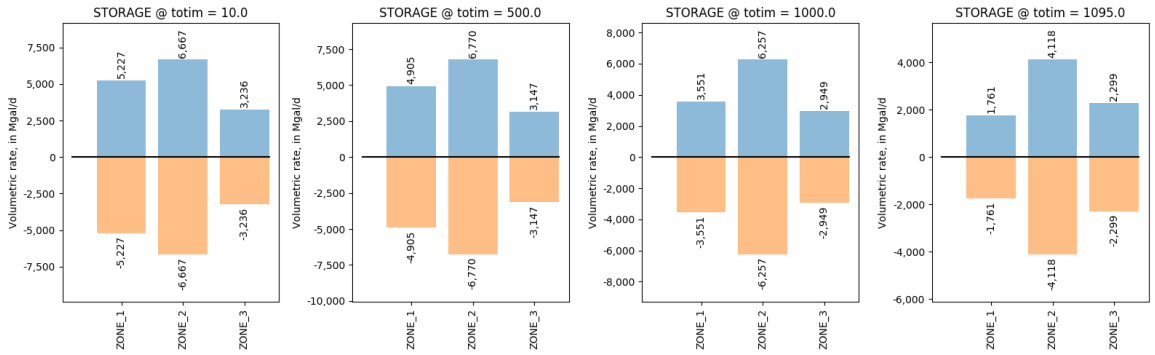
5.2 Conservación física

Dado que publicamos las imágenes con su correspondiente Dockerfile, cualquier usuario puede inspeccionarlas y mejorarlas. Las imágenes puede ser encontradas en DockerHub [13] y [14].

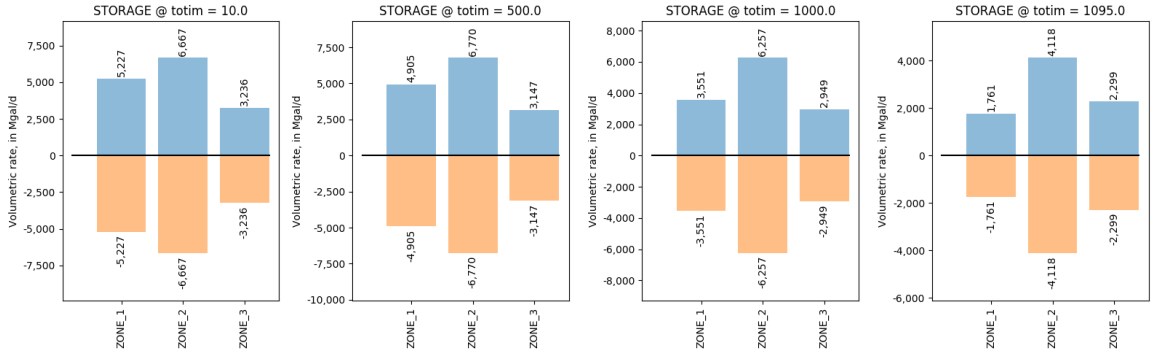
Para evaluar la conservación física utilizamos tres proveedores diferentes: DigitalOcean, Google Cloud y un local. La figura de 5.1 presenta una descripción de las características de los ambientes

El ambiente debe tener instalado Docker, la información del manifiesto de la imagen de Docker describe la versión de Docker necesaria. Sin embargo, Docker asegura idempontencia para los ambientes CentOS7, Debian 10/9/8/7.7, Fedora 26/27/28, Ubuntu 14.04/16.06/18.04,

[13] https://hub.docker.com/u/dockerpedia/
[14] https://github.com/dockerpedia



(a) Resultados originales entregados por el Information Sciences Institute



(b) Resultados reproducidos por nuestra propuesta

Fig. 5.10 Los resultados obtenidos con el nuevo ambiente son iguales

Resource	Digital Ocean	Google Compute	Local
RAM (GB)	8	8	4
Disk (GB)	100	100	70
CPU (GHz)			
CPU (Arch)	64	64	64
OS	Centos 7	Debian 9	Fedora 27

Table 5.1 Image appliances characteristics.

Windows 10, macOS El Capitan 10.11 o nuevas versiones. El proceso de instalación puede ser encontrado en la documentación oficial https://docs.docker.com/install/

Cada imagen de Docker tiene archivo README con las instrucciones para correr el experimento. Las figuras 5.1 5.2 y 5.3 muestran los pasos necesarios para correr el experimento computacional.

Para correr el experimento y descargar la imagen:

Listing 5.1 Descargar y correr la imagen disponible en DockerHub mosorio/pegasus_workflow_images:soykb

```
docker run -d --rm -it --name soybean \
 mosorio/pegasus_workflow_images:soykb
```

Luego, el usuario debe entrar al container. El usuario puede confirmar que se encuentra dentro del container por el cambio de símbolo de la terminal (prompt).

Listing 5.2 Entrar al ambiente computacional utilizando bash

```
root@docker-instance:~# docker exec \
-ti -u workflow:workflow soybean bash
workflow@a0f861e6fbc4:~
```

Finalmente, correr el workflow.

Listing 5.3 Run the workflow

```
workflow@a0f861e6fbc4:~/soykb \
./workflow-generator --exec-env distributed
```

Para evaluar si las imágenes Docker son livianas y almacenables, nosotros construimos dos imágenes, una utilizando Docker y otra utilizando máquinas virtuales. La imagen de Docker se construye bajo nuestro enfoque y la imagen de la máquina virtual basada en el trabajo de [3]. Luego comparamos el uso de disco de ambas.

5.3 Conservación lógica

A través de las anotaciones realizadas, buscamos describir los ambientes computacionales en forma automática, comparar las diferencias entre dos ambientes y construir un ambiente computacional similar que permita reproducir la ejecución del workflow.

Para ello, anotamos los flujos de trabajo con nuestra herramienta, las anotaciones están agrupadas por: pasos de construcción y componentes de software.

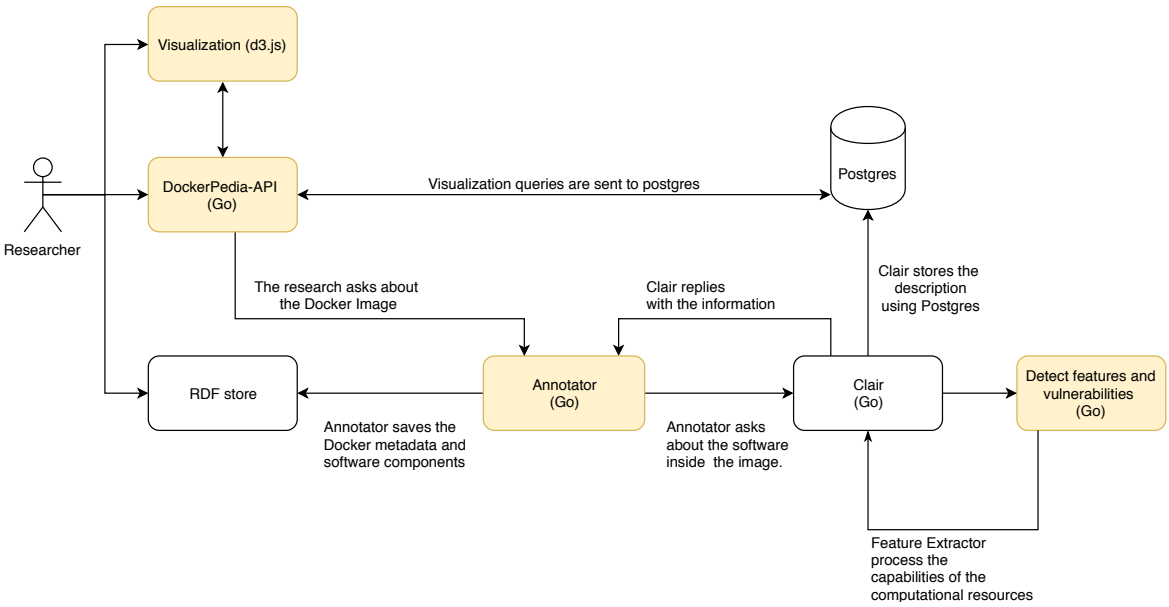


Fig. 5.11 Figure caption.

Para evaluar la calidad de las anotaciones utilizamos dos experimentos.

- Reproducir el ambiente utilizando los pasos de construcción representados por DeploymentPlan
- Detectar las similitudes y diferencias entre dos ambientes computacionales usando distintas versiones de la distribución Linux.

Para obtener las anotaciones, proponemos usar Clair y construir un sistema de anotador. La figura 5.11 muestra los pasos principales del sistema.

1. El investigador pregunta sobre la información de una imagen al sistema anotador. El sistema anotador se encuentra escrito en GoLang y disponible en nuestro repositorio.
2. El sistema anotador pregunta a Clair sobre software y sus vulnerabilidades de la imagen. Nosotros utilizamos nuestra propia versión de Clair que puede detectar componentes de software instalado por Conda.
3. Para obtener los pasos de construcción, etiquetas, arquitectura y más información. El anotador obtiene el manifiesto de la imagen desde DockerHub
4. El sistema anotador guarda la información usando RDF y ontología propuesta.

Para reproducir el entorno, el usuario debe clonar el repositorio y construir la imagen. La URL del repositorio y el cambio asociado (representado por un git commit) se pueden obtener por dos métodos: consultando nuestras anotaciones o usando un comando Docker. El listado 5.4 muestra las etiquetas de la imagen de flujo de trabajo SoyKB.

Listing 5.4 Inspect image annotations

```
root@docker-instance:~# docker inspect \
    --format='{{json .Config.Labels}}' \
    dockerpedia/soykb:latest
```

La figura 5.5 muestra algunas de las etiquetas de la imagen basado en *Open Container Initiative*

Listing 5.5 Inspect image annotations

```
{
"maintainer": "Maximiliano Osorio <mosorio@inf.utfsm.cl
    >",
"org.label-schema.build-date": "2018-11-10T21:11:28Z",
"org.label-schema.name": "Soybean Knowledge Base",
"org.label-schema.schema-version": "1.0",
"org.label-schema.url": "http://www.soykb.org/",
"org.label-schema.vcs-ref": "15955b0",
"org.label-schema.vcs-url": "https://github.com/
    dockerpedia/soykb",
"org.label-schema.vendor": "DockerPedia",
"org.label-schema.version": "1.0"
}
```

Para evaluar si los entornos son similares, comparamos ambas imágenes utilizando el lenguaje de consulta SPARQL 1.1. El experimento fue un caso real en el que una imagen podía ejecutar el flujo de trabajo y la otra no. La figura 5.7 muestra la consulta para identificar los diferentes componentes de software entre dos imágenes.

Listing 5.6 ¿Cuáles son los diferentes componentes entre dos imágenes?

```
SELECT * WHERE {
 pegasus_workflow_images%3Alatest
  vocab:containsSoftware ?p .
```

```
  MINUS{
  pegasus_workflow_images%3Apegasus-4.8.5
   vocab:containsSoftware ?p
  }
 }
```

Listing 5.7 ¿Cuáles son los componentes que comparten entre dos imágenes?

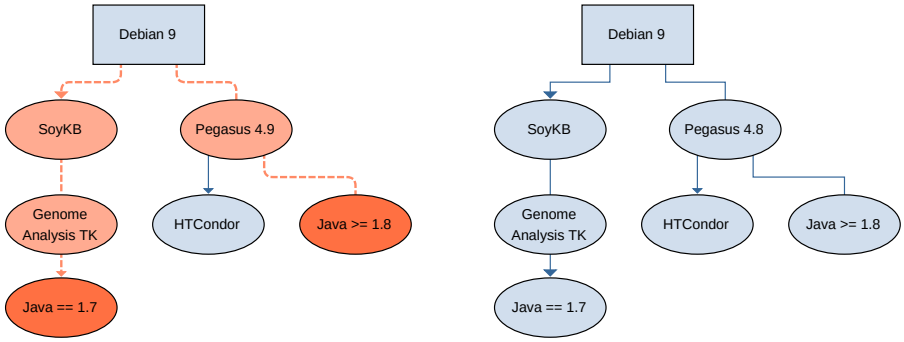
```
SELECT * WHERE {
  pegasus_workflow_images%3Alatest
       vocab:containsSoftware ?p .
  pegasus_workflow_images%3Apegasus-4.8.5
       vocab:containsSoftware ?p
     }
```

5.4 Resultados y discusión

Ejecutamos las imágenes sobre sus plataformas correspondientes. Aún mas, los componentes de software de la plataforma y la versión Docker son diferentes. Sin embargo, Docker garantiza que el software siempre funcionará de la misma manera, independientemente de dónde se instale. Todas las ejecuciones se compararon con la original en una imagen VM predefinida, donde ya existía un entorno de ejecución. Los resultados muestran que los ambientes de ejecución de contenedores son capaces de ejecutar completamente sus workflow relacionados. Para comprobar que no sólo los flujos de trabajo se ejecutan con éxito, sino también que los resultados son correctos y equivalentes, comprobamos los datos de salida producidos.

Por otra parte, los resultados experimentales muestran que nuestra propuesta puede detectar automáticamente los componentes del software, las vulnerabilidades relacionadas, los pasos de construcción y los metadatos específicos de los experimentos científicos en forma de imágenes Docker. Además, los resultados muestran que es posible extender Clair para anotar a otros gestores de paquetes.

Las anotaciones generadas por nuestro enfoque permiten comparar los componentes de software entre dos o más entornos. Esta función se puede utilizar como herramienta de depuración cuando un entorno reproducido no funciona. Por ejemplo, en agosto de 2018, construimos la imagen del workflow SoyKB, y pudimos ejecutar el flujo de trabajo con éxito.



(a) Dependencies graph Pegasus 4.9 and SoyKB

(b) Dependencies graph Pegasus 4.8 and SoyKB

Fig. 5.12 The orange nodes are the differences between the images

Sin embargo, reconstruimos una nueva imagen en noviembre con el mismo DeploymentPlan y no se pudo ejecutar el workflow con éxito.

Describimos los componentes de software dentro de ambas imágenes. Y encontramos las siguientes diferencias:

- Agosto: Pegasus 4.8 y Java 1.7
- Noviembre: Pegasus 4.9 y Java 1.8

Así, analizamos el código y la documentación de SoyKB y las dependencias de Pegasus 4.9. Como resultado, obtuvimos las gráficas de dependencia mostradas en la figura 5.12a. Los gráficos de dependencia muestran que el paquete Pegasus 4.9 y SoyKB no son compatibles debido a sus requerimientos de versión Java.

Así, construimos una nueva imagen instalando Pegasus 4.8, y obtuvimos los gráficos de dependencia que se muestran en la figura 5.12b. Aquí, el gráfico no tiene un conflicto y se pudo ejecutar el workflow con éxito.

La nueva imagen fue nombrada como: `pegasus_workflow_images:4.8.5`

La razón principal de los trabajos anteriores para evitar la conservación física era la gran demanda de almacenamiento de máquinas virtuales. Afirmamos que las imágenes Docker son ligeras.

Los resultados de la comparación del uso del disco muestran que hay una disminución del 64,2% para la imagen Pegasus y del 41,5% para la dispel4py. La tabla 5.2 muestra la diferencia para el sistema de flujo de trabajo Pegasus y dispel4py.

Enfoque	Pegasus (MB)	dispel4py (MB)
Virtualization	1929	3509
Container	690	2500

Table 5.2 Uso de disco de las imágenes pegasus y dispel4py utilizando máquinas virtuales y containers

Chapter 6

Evaluación

6.1 Main 1

Dos evaluaciones de reproducción: - utilizando el repositorio github con toda la información - utilizando el manifiesto de la imagen

Chapter 7

Conclusiones y trabajo futuro

7.1 Main 1

m at, molestie in quam. Aenean rhoncus vehicula hendrerit.

References

[1] V. C. Stodden, "Reproducible research: Addressing the need for data and code sharing in computational science," *Computing in science & engineering*, vol. 12, no. 5, pp. 8–12, 2010.

[2] 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*, vol. 8, no. 11, p. e80278, 2013.

[3] 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*, vol. 67, pp. 354–367, 2017.

[4] A. Dappert, S. Peyrard, C. C. Chou, and J. Delve, "Describing and preserving digital object environments," *New Review of Information Networking*, vol. 18, no. 2, pp. 106–173, 2013.

[5] D. James, N. Wilkins-Diehr, V. Stodden, D. Colbry, C. Rosales, M. Fahey, J. Shi, R. F. Silva, K. Lee, R. Roskies *et al.*, "Standing together for reproducibility in large-scale computing: Report on reproducibility@ xsede," *arXiv preprint arXiv:1412.5557*, 2014.

[6] V. Stodden, F. Leisch, and R. D. Peng, *Implementing reproducible research*. CRC Press, 2014.

[7] T. Craddock, P. Lord, C. Harwood, and A. Wipat, "E-science tools for the genomic scale characterisation of bacterial secreted proteins," in *All hands meeting*, 2006, pp. 788–795.

[8] D. Blankenberg, G. V. Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor, "Galaxy: a web-based genome analysis tool for experimentalists," *Current protocols in molecular biology*, pp. 19–10, 2010.

[9] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor *et al.*, "Galaxy: a platform for interactive large-scale genome analysis," *Genome research*, vol. 15, no. 10, pp. 1451–1455, 2005.

[10] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientific workflows onto the grid," in *Grid Computing*. Springer, 2004, pp. 11–20.

[11] J. Zhao, J. M. Gómez-Pérez, K. Belhajjame, G. Klyne, E. García-Cuesta, A. Garrido, K. M. Hettne, M. Roos, D. D. Roure, and C. A. Goble, "Why workflows break - understanding and combating decay in taverna workflows," in *8th IEEE International Conference on E-Science, e-Science 2012, Chicago, IL, USA, October 8-12, 2012*. IEEE Computer Society, 2012, pp. 1–9. [Online]. Available: https://doi.org/10.1109/eScience.2012.6404482

[12] T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004. [Online]. Available: https://doi.org/10.1093/bioinformatics/bth361

[13] B. Matthews, E. Conway, J. Woodcock, C. Jones, J. Bicarregui, and A. Shaon, "Towards a methodology for software preservation," in *Proceedings of the 6th International Conference on Digital Preservation, iPRES 2009, San Francisco, CA, USA, October 5-6, 2009*, 2009. [Online]. Available: http://hdl.handle.net/11353/10.294040

[14] G. R. Brammer, R. W. Crosby, S. Matthews, and T. L. Williams, "Paper mâché: Creating dynamic reproducible science," in *Proceedings of the International Conference on Computational Science, ICCS 2011, Nanyang Technological University, Singapore, 1-3 June, 2011*, ser. Procedia Computer Science, M. Sato, S. Matsuoka, P. M. A. Sloot, G. D. van Albada, and J. J. Dongarra, Eds., vol. 4. Elsevier, 2011, pp. 658–667. [Online]. Available: https://doi.org/10.1016/j.procs.2011.04.069

[15] P. Buncic, C. A. Sanchez, J. Blomer, L. Franco, A. Harutyunian, P. Mato, and Y. Yao, "Cernvm–a virtual software appliance for lhc applications," in *Journal of Physics: Conference Series*, vol. 219, no. 4. IOP Publishing, 2010, p. 042003.

[16] F. S. Chirigati, D. E. Shasha, and J. Freire, "Reprozip: Using provenance to support computational reproducibility," in *5th Workshop on the Theory and Practice of Provenance, TaPP'13, Lombard, IL, USA, April 2-3, 2013*, A. Meliou and V. Tannen, Eds. USENIX Association, 2013. [Online]. Available: https://www.usenix.org/conference/tapp13/technical-sessions/presentation/chirigati

[17] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, and R. K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Comp. Syst.*, vol. 46, pp. 17–35, 2015. [Online]. Available: https://doi.org/10.1016/j.future.2014.10.008

[18] F. da Veiga Leprevost, B. A. Grüning, S. Aflitos, H. L. Röst, J. Uszkoreit, H. Barsnes, M. Vaudel, P. Moreno, L. Gatto, J. Weber, M. Bai, R. C. Jimenez, T. Sachsenberg, J. Pfeuffer, R. V. Alvarez, J. Griss, A. I. Nesvizhskii, and Y. Pérez-Riverol, "Biocontainers: an open-source and community-driven framework for software standardization," *Bioinformatics*, vol. 33, no. 16, pp. 2580–2582, 2017. [Online]. Available: https://doi.org/10.1093/bioinformatics/btx192

[19] B. K. Beaulieu-Jones and C. S. Greene, "Reproducibility of computational workflows is automated using continuous analysis," *Nature biotechnology*, vol. 35, no. 4, p. 342, 2017.

[20] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2723872.2723882

[21] M. E. Aranguren and M. D. Wilkinson, "Enhanced reproducibility of sadi web service workflows with galaxy and docker," *GigaScience*, vol. 4, no. 1, p. 59, 2015.

[22] M. Osorio, C. B. Aranda, and H. Vargas, "Reproducibility of computational environments for scientific experiments using container-based virtualization," in *Proceedings of the Second Workshop on Enabling Open Semantic Science co-located with 17th International Semantic Web Conference, SemSci@ISWC 2018, Monterey, California, USA, October 8-12th, 2018.*, ser. CEUR Workshop Proceedings, D. Garijo, N. Villanueva-Rosales, T. Kuhn, T. Kauppinen, and M. Dumontier, Eds., vol. 2184. CEUR-WS.org, 2018, pp. 43–51. [Online]. Available: http://ceur-ws.org/Vol-2184/paper-05.pdf

[23] 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*, ser. CODASPY '17. New York, NY, USA: ACM, 2017, pp. 269–280. [Online]. Available: http://doi.acm.org/10.1145/3029806.3029832

[24] D. Huo, J. Nabrzyski, and C. Vardeman, "Smart container: an ontology towards conceptualizing docker." in *International Semantic Web Conference (Posters & Demos)*, 2015.

[25] A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider, "Sweetening ontologies with dolce," in *International Conference on Knowledge Engineering and Knowledge Management*. Springer, 2002, pp. 166–181.

[26] 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*, vol. 1931, 2017, pp. 1–4.

[27] T. Joshi, K. Patil, M. R. Fitzpatrick, L. D. Franklin, Q. Yao, J. R. Cook, Z. Wang, M. Libault, L. Brechenmacher, B. Valliyodan *et al.*, "Soybean knowledge base (soykb): a web resource for soybean translational genomics," in *BMC genomics*, vol. 13, no. 1. BioMed Central, 2012, p. S15.

Appendix A

Installing the CUED class file

LaTeX.cls files can be accessed system-wide when they are placed in the <texmf>/tex/latex directory, where <texmf> is the root directory of the user's TeXinstallation. On systems that have a local texmf tree (<texmflocal>), which may be named "texmf-local" or "localtexmf", it may be advisable to install packages in <texmflocal>, rather than <texmf> as the contents of the former, unlike that of the latter, are preserved after the LaTeXsystem is reinstalled and/or upgraded.

It is recommended that the user create a subdirectory <texmf>/tex/latex/CUED for all CUED related LaTeXclass and package files. On some LaTeXsystems, the directory look-up tables will need to be refreshed after making additions or deletions to the system files. For TeXLive systems this is accomplished via executing "texhash" as root. MIKTeXusers can run "initexmf -u" to accomplish the same thing.

Users not willing or able to install the files system-wide can install them in their personal directories, but will then have to provide the path (full or relative) in addition to the filename when referring to them in LaTeX.