An integrated system for building and running reproducible research

Written by Mario Taddei, mariotaddei@rocketmail.com
supervised by Olivier Dalle, olivier.dalle@inria.fr
and Davide Sangiorgi, davide.sangiorgi@gmail.com

University of Nice-Sophia Antipolis
Master II IFI UBINET 2014/2015

Alma Mater Studiorum
Scuola di scienze
Corso di Laurea Magistrale in Informatica
Anno Accademico 2014/2015
Sessione II

This report is about the 6 months at Inria-Sophia Antipolis with the SCALE team.

Index

Introduction

Context

Our generation of computational scientists is living in an exciting time: not only do we get to pioneer important algorithms and computations, we also get to set standards on how computational research should be conducted and published. From Euclid's reasoning and Galileo's experiments, it took hundreds of years for the theoretical and experimental branches of science to develop standards for publication and peer review. Computational science, rightly regarded as the third branch, can walk the same road much faster. The success and credibility of science are anchored in the willingness of scientists to expose their ideas and results to independent testing and replication by other scientists. This requires the complete and open exchange of data, procedures and materials. The idea of a "replication by other scientists" in reference to computations is more commonly known as "reproducible research".

In this context the journal "EAI Endorsed Transactions on Performance & Modeling, Simulation, Experimentation and Complex Systems" had the exciting and original idea to make the scientist able to submit simultaneously the article and the computation materials (software, data, etc..) which has been used to produce the contents of the article. The goal of this procedure is to allow the scientific community to verify the content of the paper, reproducing it in the platform independently from the OS chosen, confirm or invalidate it and especially allow its reuse to reproduce new results.

This procedure is therefore not helpful if there is no minimum methodological support. In fact, the raw data sets and the software are difficult to exploit without the logic that guided their use or their production. This led us to think that in addition to the data sets and the software, an additional element must be provided: the workflow that relies all of them.

Work

The aim of the work is to provide an integrated system for scientific paper submission which not only allows the submission of the paper but also the experiment that produced its content, with the workflow, data sets and dependencies. This submission phase is only one step into multiple steps through which the paper should pass before its publication or its rejection: assigning to an editor for its evaluation by peer reviewers, assigning the reviewers, managing rejection/acceptance by reviewers, notification of their decisions, managing conflicts between reviewers, sending reminder in case of delayed reviews, etc. The platform has been thought to manage all these steps, including the possibility to host and run the experiment directly online independently from the OS and the software chosen by the scientists, such that the review procedure becomes faster and easier. We can identify in this last goal, the biggest challenge of the work.

As Ian Gent explained in [2], this goal is reachable if the experiment is hosted in a virtual machine. On one side the scientist does not have particular constraints on the favourite operative system and preferred software to use, but on the other side he has to submit also the workflow to provide the VM to correctly run his own experiment. For this purpose, we adopted the Vagrant software for the provisioning and management of the VMs that we will explain later in detail.

The importance of reproducibility

"An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete ... set of instructions [and data] which generated the figures." - David Donoho paraphrasing Claerbout's approach.

This quotation expresses exactly the importance of the reproducibility. As journalists report facts inside newspapers, scientists report their results inside scientific articles. But, how often do you deal with fake news on a newspaper? Or on a tg news? Or on a blog? It happens almost everyday to me. This because behind that being there are production costs and salaries for people. So, fake news bring more audience and more audience bring more money. The chain is easy and almost every time brings to the same result.

Behind a scientific conference or journal there are identically production costs, salaries to pay and an audience, so the probability that on a scientific article there are fake results is not 0, maybe is not even close to 0. Just think to the case happened on August 2012 when a paper with the title "*Independent, Negative, Canonically Turing Arrows of Equations and Problems in Applied Formal PDE*" written by Marcie Rathke of the University of Southern North Dakota at Hoople was submitted to the Advances in Pure Mathematics journal and it was accepted. The paper was without any sense, because generated with the software MathGen developed by Nate Eldredge, and moreover the professor Marcie Rathke was an invented person.

When we are reading an article we are never completely sure of the work that is behind those results. We don't know if the authors really got them or they are just cheating. But if we can have access to this work and reproduce those results, automatically the job done becomes trusted.

In theory, when we have open source, open data and open access we can reproduce the experiment, but as reported in [3] the main reasons on lack of reproducibility are 2:

- the authors are not careful enough and the code remains hard to understand,
- the project is not well documented.

So, in practice we need that the reproducibility comes easily. This concept of "easy reproducibility" cannot be defined rigorously, but we can say that the property is respected when a reviewer does not spend too much time to reproduce and trust the work of his peer.

So, we can recognize in the easy reproducibility the main goal of our web platform.

State of the art

Reproducibility is a concept that is born with the science itself, but the first that stressed the term was the chemist Robert Boyle in England in the 17th century (Wikipedia). In computer science we can find “reproducibility tools”, for example for managing the workflow, for solving the dependencies or for the automation of a project, since the 70s. The Make project, that automatically builds programs from source code by reading makefiles, is an example still currently widely used.

Nowadays, we have many technologies for workflow tracking and research environments that help the research to be reproducible. For example:

- VisTrails [6] is a tool for workflow and provenance management that provides support for simulations, data exploration and visualization. Similar solutions are Pegasus and Taverna [7,8]**.**
- Sumatra [9] helps the scientist to manage and track the project describing it in a file with all the parameters and input data needed.
- CDE [10] is a lightweight application virtualization for Linux that permits to move the project through different Linux OSes, such as Debian, Ubuntu, Fedora, OpenSUSE and CentOS.

Regarding the dissemination platform, we can find:

- ResearchCompendia.org [11] that allows users to share the experiment associated with the paper so that is available for everyone and Thedatahub.org and nanoHUB.org are similar,
- RunMyCode.org and recomputation.org [3] are closer to our goal. In fact they already give the possibility to users to submit the experiment with the paper and run it on the platform.

In the last case there are 2 main differences with our system:

1) is not provided an automatic way to run an experiment on the environment chosen by the authors,
2) the experiment are ready to run on the platform after that they are published, so the review phase has been made before in the traditional way while we want our platform useful mainly during the review phase.

Description

The platform

We are discussing about computational experiments. Often it can be hard to reproduce experiments in other scientific disciplines, for example in physics. Imagine to reproduce the experiment of the Faraday cage. It is feasible, but we need particular equipment, like a thunder lightning generator. Or other experiments, where a particular temperature or a particular environment is required.

In computational science everything is much more "easier" because runs inside a single or a set of machines. But to make a set of chips, cables and electronic components work properly to get the wanted results, we need to find and put together a lot of different equipments, that in this case is a combination of software and hardware: from the operative system to the compiler to all the needed libraries and dependencies that make your experiment run. And take care of the problem of software versions… How many times you have seen a software working with a version and crashing with the next one? Fortunately, we can easily make a photograph of the working machine and run it everywhere as a Virtual Machine. With a VM you can have the hardware you want and install inside all the desired software and when it's done, get the results and destroy it. The world of lazy boys, that in a few times they can pass from creating an entity to destroy it. A god game.

Our platform does not solve the problems of bad written or poorly documented code, but more than others solves the problems of "it works on my machine" and "too much time to reproduce it". To see if an author is cheating the reviewer has to go to read the source code, as it is done without our platform. But if the reviewer needs only to reproduce the experiment, he has just to click a button on the HTML page of his browser and inside the platform a VM is created to host the execution of the experiment and to collect the results. So, the reviewer does not need particular hardware or software, because everything is managed by the web service. He just needs a browser and an internet access.

As explained before, this is a goal already reached by other platforms, but with the difference that the execution procedure comes not automatically.

The platform that we want is using VMs to run experiments. The state of a VM can be easily preserved, so that once that a VM is trusted, an experiment can run on it and the reviewer can accept it or not.

To achieve this the VM has to be trusted. Trusting a VM means review the packages that will be installed inside and their installation workflow. We need to do it because the dependencies that the experiment uses, and the dependencies of the dependencies, till the Assembler primitives, they are all part of the experiment.

Vagrant

In this paragraph we will introduce briefly the Vagrant software, but for more information the reader can consult the documentation on vagrantup.com.

Vagrant is a software for managing VMs. It consists in a wrapper for VirtualBox and VMware with a per-project Ruby file, called Vagrantfile, where the user writes the instructions to build the VM and sets his own environment. It can create both Linux and Windows machines (still work in progress for Mac OS) and it supports shell scripting, Chef and Puppet.

We use it in combination with VirtualBox, but it works also with VMware.

This is an example of the Vagrant functioning:

- A shell script bootstrap.sh

```
#!/usr/bin/env bash

apt-get update
apt-get install -y apache2
if ! [ -l /var/www ]; then
     rm -rf /var/www
     ln -fs /vagrant /var/www
fi
```

- The Vagrantfile

```
Vagrant.configure(“2”) so |config|
     config.vm.box = “hashicorp/precise32”
     config.vm.provision :shell, path: “bootstrap.sh”
end
```

The first file “bootstrap.sh” is a shell script that installs Apache on an Ubuntu OS.

The second, is the Vagrantfile that just selects the box we want to use, in this case Ubuntu Precise 32, and runs the script “bootstrap.sh” at the startup of the VM.

Assuming that we imported the box “hashicorp/precise32” in the Vagrant boxes we just need to run the command

```
$ vagrant up
```

and the VM will run inside VirtualBox installing the wanted software.

Why Vagrant?

Being a tool for managing Virtual Machines, the main reason regards exactly the utilization of VMs. As we said before, experiments have to be run inside self-content VMs and Vagrant represents a solution pretty simple to learn and to use. Another solution that we could take into account is represented by Docker. Differently from Vagrant, Docker is a Virtual Environment manager. A Docker container is long way faster and more lightweight than a Virtual Machine, but the application running inside is not full isolated since different Docker containers share the same Kernel. Moreover, Docker is an extension of Lxc (Linux Container), and it is usable, without any adaptation layer, only in Linux platforms, while Vagrant can accept also Unix and Windows boxes.
So, to wrap up, the main reasons are:

- full isolation,
- bigger OSes' choose.

In our platform, VMs are not living long lives, just a bit more than the time to run the experiment. They are just created and provisioned with the necessary software, then, once that the experiment finishes, the output is collected in a shared folder and they are destroyed.
Vagrant performs the role of managing this workflow. In a Vagrantfile the user can specify the software to install inside the VM, as explained above, and the instructions to run the experiment.
We think that learning Vagrant can be a good compromise because we can have easy reproducibility without removing too much freedom to the user. He can choose the preferred OS and software. So, as the user has to prepare his machine for running the experiment, he can prepare the similar VM with the same scripts he used.

Existing solutions

Nowadays, when an article with results got from a computational experiment is published, it can be reproduced and reviewed mainly in 4 ways:

- **Old school:** the experiment is just described with the readme file of the project and is exchanged directly with the reviewer. Reproducibility and review time depend on how much the project is well described and the source code written with care. It is not guaranteed that the experiment will be available after the publication (Open Access).
- **Tools support:** the experiment makes use of one or more tools (Make, Maven, Virtualenv, CDE, VisTrails…) that led to a review phase easier and faster.
- **Dissemination platforms:** the experiment is published on web services that act as research project repositories, where the user can find all the material concerning the experiment. The review phase can be done like in the previous 2

points if the platform does not have a review management support. In this case the Open Access property is guaranteed, in fact the experiment remains available online for future peer reviews.

- **Dissemination platforms with run support:** in this case it is possible to run the experiment online such that the reviewer does not have to prepare the project environment, but the execution not always is self-contained and never is automatic. In fact, there is a phase where the scientist collaborates with sysadmins to make the online execution feasible. This is mostly done after that a paper is published, this means that the online run support was not available during the review phase, but it is only usable in future peer reviews.

Our solution is build on top of the last method. It is, in fact, a dissemination platform, but the design admits the experiments run automatically without more interventions from the author or the admins. Thanks to this, the experiment will be ready since the first moment for the initial reviews and it will remain available for future peers.

Contribution

Artifact

An artifact consists in 3 parts:

1. Box: a compressed .box file containing the OS, where the experiments will run, and some metadata. It will be passed to the provider, VirtualBox in our case. A box for Vagrant must be created respecting some rules listed in the documentation (http://docs.vagrantup.com/v2/boxes/base.html). There are also websites, like www.vagrantbox.es, where it is possible to find boxes ready to use.
2. Vagrantfile: the Vagrant per-project Ruby file that represents the workflow for providing the VM.
3. Packages: it is a software package, with its dependencies, or more than one attached with the workflow to install it/them in the OS (e.g. a Shell script).

The VM generated from the artifact has to be self-contained. For this reason all the packages needs to be inside the artifact, since the VM cannot download anything from the internet. This because an artifact has already been trusted before by a reviewer, so, if we change its state it becomes not trusted anymore.
An example of artifact can be the box Ubuntu Trusty 64-bit, with a Vagrantfile and the set of software consists in C++ and NS-3. There are different best practices to cache the packages so that later they can be installed in the VM. An easy way to create an artifact like this is to run locally the selected box, in this case Ubuntu, install C++, in the build-essential apt package and cache it with all the dependencies in a tar.gz file. The apt software provides the option -d to download the package and the dependencies inside the /var/cache/apt/archives folder. Attached to this tar.gz the user has to define a workflow, for example a bash script that install the cached apt-packages and the NS-3 software inside the VM.

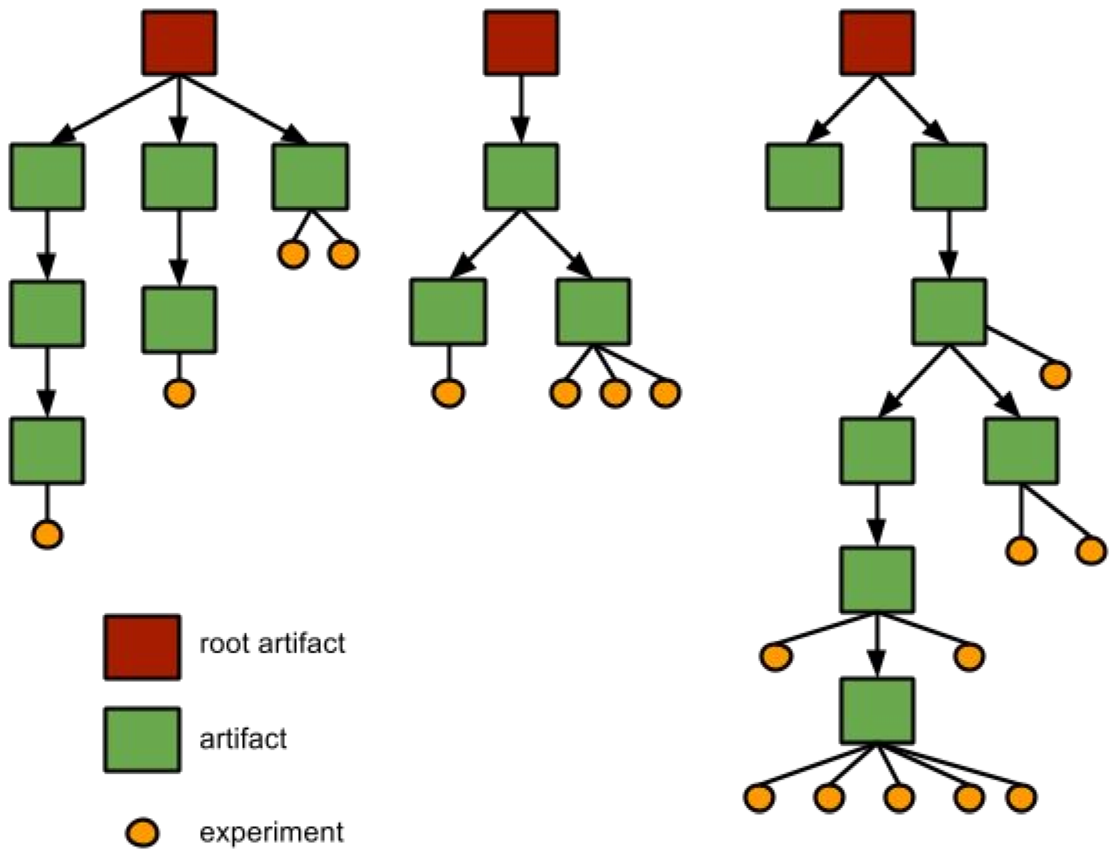
Definitions

- Artifacts are expressed with capital letters: `A,B,C` ...
- `s(A)`: the software of the artifact `A` or set of software packages of `A`.
- `s(B)>s(A)`: `B` contains the software of `A` or in the set of packages of `B`, there are the same packages of `A` and at least one more `s(B)=S(A)+p` where `p` is a random package.
- `A->B`: `A` is extended with `B` or `B` extends `A`, if `s(B)>s(A)`.

The space of artifacts

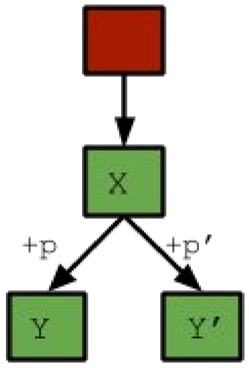
In general, an Artifact is an extension of another one, but inside the set of all the artifacts we have a subset containing elements that are a point of start for the others, they are called *Root Artifacts.*

A root artifact is not extending any other artifact.
The space of artifacts is a set of special Directed Acyclic Graphs (DAGs), where a DAG has only one root.



Why a set of DAGs and not a forest? Because an artifact can be generated by the union of two artifacts, as explained below.
Assume that we have an artifact `X` and we extend it twice creating 2 different artifacts:

- `Y` where `s(Y)=s(X)+p`
- `Y'` where `s(Y')=s(X)+p'`

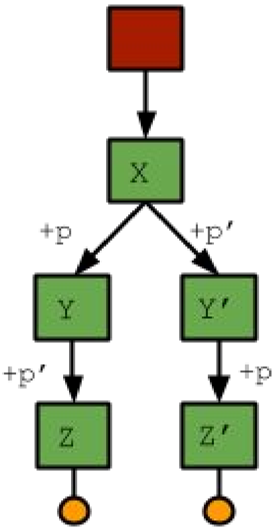


Then we extend `Y` with `p’` obtaining the artifact `Z` where

- `s(Z)=s(Y)+p’`

and `Y’` with p obtaining `Z’` where

- `s(Z’)=s(Y’)+p`



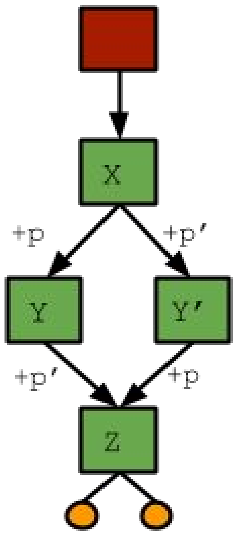
so

- `s(Z)=s(Y)+p’=s(X)+p+p’`
- `s(Z’)=s(Y’)+p=s(X)+p’+p`

then, assuming `p+p’` commutative

➢ `s(Z)=s(Z’)`

`Z` and `Z’` are identical artifacts and we can just list one of them.

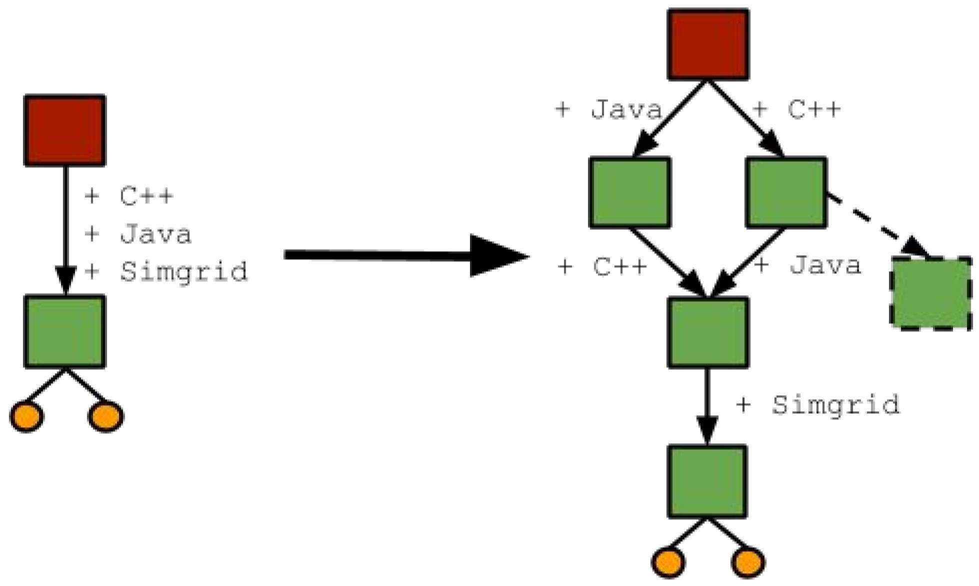


The reviewer's responsibilities

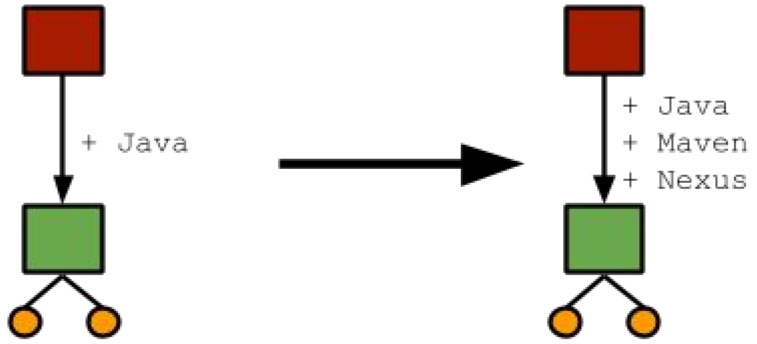
The review phase can be a process complicated with several subphases like assignation of the reviewers, reassignation in case of delay, merge of the reviews …
In our platform we introduced one more step: the review of the artifact. Reviewing an artifact involves several things:

1. to check that the software that will be installed in the VM are well set up,
2. to check that the VM is self-contained,
3. to understand if the artifact contains too much software and needs to be divided in more than one artifact.
4. to understand if the artifact contains not enough software and it needs something more.

We will clarify the last two points with examples.
About the 3rd, imagine a situation where on the platform there is a root artifact with Ubuntu and a user wants to extend it with an artifact containing a simulator, like Simgrid, that requests both Java and C++ installed on the machine. A best practice, is not to create directly an artifact with the simulator and all the dependencies inside, but to make more steps, creating more than one artifact. For example, in this case, we may have an artifact with C++, one with Java, one with C++ and Java and one with C++, Java and Simgrid, such that in the future it can be possible to extend with another artifact the DAG through the C++ node without having the Java installation. We can deduce that there are 2 trade-off, one, between high reusability with higher number of artifacts and another one between low amount of reviews with low amount of packages.



About the point 4, when I need, for example, an artifact with a Java installation, it is a best practice to install inside also Maven, a software that solves Java dependencies, and Sonatype Nexus, a caching system for Java dependencies, such that the creation of following self-contained artifacts comes easier for scientists that know these software and will make use of the artifact in the future.



Web service's overview

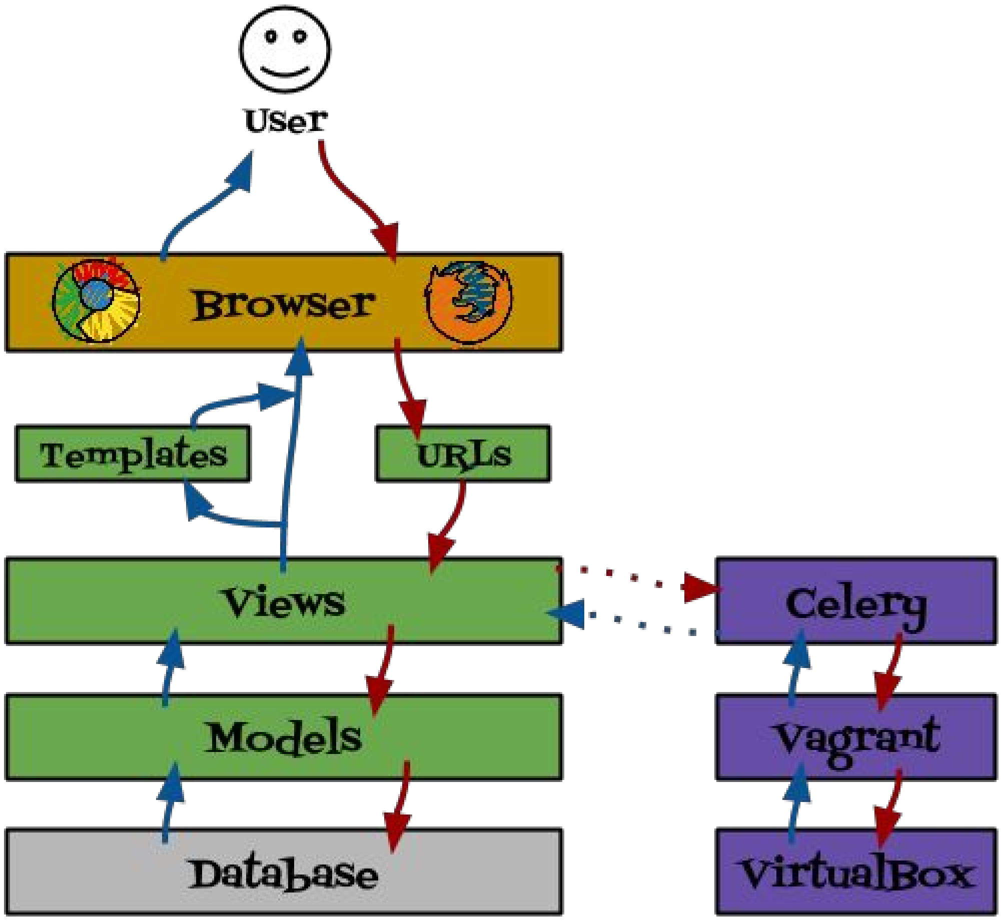
The web service has been primarily designed to be a repository for publications with a system that permits to run automatically the experiments related. On this platform, the experiments are run inside VMs, created when the user wants to execute it and destroyed when the experiment terminates its computations.
There are 4 main entities:

- Users: except the admin, there are not special users. A user can act both as a publisher or a reviewer.
- Artifacts: there is a section dedicated to artifacts with a list of all the available items. An artifact's page shows a description, with the list of packages installed inside, the list of experiments runnable on it and the previous and next artifacts in the DAG.
- Papers: the section of papers is designed to host all the material related to the publication. In fact, the user can decide to upload just the .pdf of the article or also the latex file with images that generated it. From this section, it is possible to move to the experiments' sections related to the paper, if any. The section contains also the merged reviews and it shows the state of the paper: accepted/pending/rejected and reproducible/unreproducible.
- Experiments: an experiment is not so different from an artifact, it contains the files needed to run it and the Vagrantfile. This last one depends on the Artifact where the experiment will run on. So, the scientist needs to know which artifact to choose before uploading the experiment. The Vagrantfile attached is a copy of the Vagrantfile of the artifact plus the source code lines needed to run the

experiment. From the experiment section it is possible to run the experiment and check the state: running/completed/failed.

The platform consists in a RESTful web service. It is developped in Django, a Python framework, and it manages VirtualBox VMs through the Vagrant software. Vagrant is run asynchronously with Celery, a Python library for asynchronous tasks.

The WS's design follows and extends the Django MVC as shown in the following picture.



The Django MVC design is represented in green. The user accesses the web service through URLs from his browser. The core of the platform is represented by Django views, functions directly connected to URLs. Database's entities are abstracted by Django models that are used inside the views. In purple we can find the asynchronous procedures that through the Celery library can run VirtualBox VMs managed by Vagrant. The platform renders its views to the user's browser through Django templates, HTML pages with a special syntax to insert dynamic content.

How to create an artifact

This section is dedicated to explain briefly to the reader with some examples what are the best practices the scientist should adopt to prepare easily an artifact from his working environment. The aim of the section is to show to the reader that the compromises to accept are not that difficult to deal with.
As we said previously, an artifact is composed by:

- an Operative System in format .box,
- a Vagrantfile,
- one or more software packages with their dependencies

and it will generate a Virtual Machine where an experiment will run on.
We remember to the reader that the VM needs to be self-contained.

How to install Java 1.7 in Ubuntu 14.04

Imagine a situation where there is a root artifact in the system with a vanilla Ubuntu box and a user needs to extend it with Java 1.7 (as we explained above, is a best practice to install inside also Maven and Sonatype Nexus). Assuming that the user wants to install the OpenJDK version on Java and he is using the same OS that he wants to extend, he would normally install the desired package on his environment writing on the command line:

```
$ sudo apt-get install openjdk-7-jdk
```

The apt software downloads the package with all the required dependencies and it installs Java 1.7 on the machine.
In order to create an artifact self-contained we have to cache all this packages. One way to cache apt packages is to copy them from the folder /var/cache/apt/archives/. The few commands to do it are listed here:

```
$ sudo apt-get clean
$ sudo apt-get update
$ sudo apt-get -d install -y openjdk-7-jdk
$ tar -cvzf apt-requirements.tar.gz -C
  /var/cache/apt/archives/ ./
```

Where:

- -d for downloading only the package,
- -y for proceeding without typing ‘yes’ and

- the last command to create a tar.gz with all the packages cached.

A way to install the packages inside the VM is the following:

```
$ mkdir apt-requirements
$ cd apt-requirements
$ tar -xvzf ../apt-requirements.tar.gz
$ sudo dpkg -i *.deb
```

Where *dpkg* is a primitive of *apt* and in the last command it installs all the *.deb* packages in the folder, or rather all the packages we untared from the *tar.gz*.
Maven can be install with the same method.
Instead, when I want to install a package manually, like Sonatype Nexus, or the latest version of Maven, or Java Oracle, I need to list all the commands contained in the instructions in a script or directly in the Vagrantfile.

How to maintain Ubuntu 14.04 self-contained

When we want to extend an artifact with apt packages we need to prevent apt to update the old ones. In apt there is a way to *mark hold* the packages that you don't want to update as in the following command:

```
$ sudo apt-mark hold $(dpkg --get-selections | grep -v
  deinstall | awk '{print $1}')
```

This command is marking hold all the packages in the Operative System so that the artifact remain trusted and we are sure that only the new packages have to be reviewed.
Unholding the packages is possible with:

```
$ sudo apt-mark unhold package_name
```

How to install pip-requirements

Pip is a package management system used to install and manage software packages written in Python (Wikipedia). If I have to extend an artifact with some Python Pip packages one of the fastest way is using the Pip caching system called Wheel.
Normally a user would run the command:

```
$ sudo pip install -r requirements.txt
```

Where *requirements.txt* is supposed to be a file with a list of dependencies of the project and it is also possible to specify the version of a package.
With the Wheel caching system the commands to cache Pip packages are the following:

```
$ mkdir pip-wheel
$ pip wheel --wheel-dir=./pip-wheel -r requirements.txt
$ tar -cvzf pip-wheel.tar.gz pip-wheel requirements.txt
```

Where the packages are cached inside a folder and tared inside a tar.gz.
To install them inside the self-contained VM, the commands will be:

```
$ tar -xvzf pip-wheel.tar.gz
$ sudo pip install --no-index --find-links=./pip-wheel
  -r requirements.txt
```

Where *--no-index* make the command not caring if the user specified or not the version of the package in the *requirements.txt* file. In fact, we don't need the version because the only packages that will be installed are the ones cached inside the Wheel folder.

How to prepare an experiment

When a scientist submits an experiment he has to respect some simple rules that make it runnable inside a VM.

Self-content

As explained above a VM generated by an artifact needs to be self-contained, or the artifact cannot remain trusted. In a few words, the experiment cannot have access to the network or get information from a shared folder.
Sometimes, an experiment can be designed to download at runtime the dependencies needed, in this case, another artifact with the requested packages must be created to support the execution.

Vagrantfile

Vagrant supports Shell, Chef and Puppet provisioning, but the workflow of the experiment can be expressed from the scientist in the way that he prefers: bash, python, perl… When the user chooses his favourite language the Vagrantfile results to be just a chain of ordered calls to external scripts. This is useful for porting all the experiments written without knowing that they will be run in a Vagrant environment.

Output

Another rule to respect is that the experiment needs a folder named *output*. When an experiment ends up in the completed state, the platform will show to the user the content of this folder. The user remains free to choose the structure of the project that he prefers, in fact, if he is used to collect the results in another way, the porting consists in just writing a script that copies them inside the output folder.

Conclusions

We described the design of a platform that helps to reproduce computational experiments running them automatically online, just clicking a button, without such big constraints on the choice of Operative System and software used.
This comes with some costs:

- the user needs to learn Vagrant, a software that manages and provides VMs;
- the review phase has one more step, in fact the artifact that will generate the VM where the experiment will run on has to be reviewed and trusted as the experiment and the paper. But once that an artifact is trusted, it is forever, and all the experiments that need this artifact have just to be uploaded and run.

The platform has the primary goal to help the reviewer during the experiment review, but it can act also as a repository where a user can explore the article published, read the source code and reproduce the results without losing time on preparing an environment for running it.
As long as the web service will be available, the experiments can be reproduced. In fact, they all have the property of Long Term Reproducibility (LTR).

Next challenges

The platform is currently a prototype that needs several improvements before going in production.

Cloud approach

First of all, the platform needs a cloud approach. The fact that Virtual Machines run in the same physical machine is not an efficient and scalable solution. To be able to run in parallel several experiments, the web service needs to be run on a cloud environment. In this way, it will be also possible to host and run experiments that use more than one VM. Because of Vagrant, that in the same Vagrantfile can create and manage more than one VM, it is already possible to execute experiments with multiple VMs, but without a cloud middleware and a scheduler it could be difficult to get the results of heavy experiments.

Mac OS case

According to Apple's licensing policies, it is possible to have only the virtualization of Apple Mac OS X 10.9 (Mavericks) client or server, Mac OS X 10.8 (Mountain Lion) client or server, Mac OS X 10.7 (Lion) client or server, 10.6 (Snow Leopard) server and 10.5 (Leopard) server.
The End User License Agreement (EULA) for Apple Mac OS X legally and explicitly binds the installation and running of the operating system to Apple-labeled computers only. Mac OS X 10.5 Leopard Server, 10.6 Snow Leopard Server, 10.7 Lion client or

server, 10.8 Mountain Lion client or server and 10.9 Mavericks client or server are fully supported on VMware Fusion while running on supported Apple hardware.
Unfortunately these policies make the reproducibility in Mac OS VMs impossible without having Apple physical machines. We hope in a scientific release from the company.

Vagrant and VirtualBox support

The platforms depends strongly from Vagrant and VirtualBox. If they change some policies or stop to sustain used versions we have to follow their standards with a portings.

Next optional challenges

The platform can be improved also with some optional extensions, not necessary to the functioning of the web service, but that would make it better.

Not self-contained experiments

When an experiment is designed to download dependencies at runtime (think to a Java experiment that in the workflow has maven calls), the experiment will end in failure state because the property of VM self-contained is broken. The design of the platform can obviate to this problem in 2 ways:

- the scientist modify the experiment uploading the needed packages as it is done in the artifacts;
- the scientist extends the used artifact with the needed packages.

But a better way has to be designed, maybe relaxing the constraint of self-content when the dependencies' source is trusted, such that we are sure that the requested package is always available and not changed during the time.
An example can be the Maven and the Python Pip repositories. In fact, on these platforms it is possible to specify the versions of the wanted software and the availability is well insured.

Artifacts' DAGs view

At the moment, the user does not have an idea of the dimension of the DAG of the artifact that he is using, or the set of the DAGs of all the artifacts in the platform. In the future, it would be nice to show to the user a complete view with all the artifacts and their links with the possibility to get instantly the main informations like the software packages for provisioning the VM, the number of experiments that run on top of that and some details about the author.

Article production automatized

The platform support to reproducibility is dedicated only to experiments, but it can be extended to the entire article. The latex source code, in fact, can generate the pdf of the article after that the experiment is in the complete state, such that the output (data and figures) can be directly integrated inside the paper.

Bibliography

[1] Dalle, Olivier. "On reproducibility and traceability of simulations." *Simulation Conference (WSC), Proceedings of the 2012 Winter*. IEEE, 2012.

[2] Gent, Ian P. "The recomputation manifesto." *arXiv preprint arXiv:1304.3674* (2013).

[3] Stodden, V., Hurlin, C., & Pérignon, C. (2012, October). RunMyCode. org: a novel dissemination and collaboration platform for executing published computational results. In *E-Science (e-Science), 2012 IEEE 8th International Conference on* (pp. 1-8). IEEE.

[4] Schwab, Matthias, Martin Karrenbach, and Jon Claerbout. "Making scientific computations reproducible." *Computing in Science & Engineering* 2.6 (2000): 61-67.

[5] Stodden, Victoria. "What computational scientists need to know about intellectual property law: A primer." *Implementing Reproducible Research* (2014): 325.

[6] Freire, Juliana, et al. "Reproducibility using vistrails." *Implementing Reproducible Research* (2014): 33.

[7] Deelman, Ewa, et al. "Pegasus: A framework for mapping complex scientific workflows onto distributed systems." *Scientific Programming* 13.3 (2005): 219-237.

[8] Wolstencroft, Katherine, et al. "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud." *Nucleic acids research* (2013): gkt328.

[9] Davison, Andrew P., et al. "Sumatra: A Toolkit for Reproducible Research." *Implementing Reproducible Research* (2014): 57.

[10] Guo, Philip J., and Dawson R. Engler. "CDE: Using System Call Interposition to Automatically Create Portable Software Packages." *USENIX Annual Technical Conference*. 2011.

[11] Stodden, Victoria, Sheila Miguez, and Jennifer Seiler. "Researchcompendia. org: Cyberinfrastructure for reproducibility and collaboration in computational science." *Computing in Science & Engineering* 17.1 (2015): 12-19.