Using the EXECO toolbox to perform automatic and reproducible cloud experiments

Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lebre, Takahiro Hirofuchi

Using the EXECO toolbox to perform automatic and reproducible cloud experiments

Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lèbre, Takahiro Hirofuchi

Using the EXECO toolkit to perform automatic and reproducible cloud experiments

Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas
Université de Lyon, LIP Laboratory
UMR CNRS - ENS Lyon - INRIA - UCB Lyon 5668
Lyon, France
matthieu.imbert@inria.fr, laurent.pouilloux@inria.fr, jonathan.rouzaud-cornabas@ens-lyon.fr

Adrien Lèbre, Takahiro Hirofuchi
École des Mines
Nantes, France
adrien.lebre@inria.fr, t.hirofuchi@aist.go.jp

***Abstract*—This paper describes EXECO, a library that provides easy and efficient control of local or remote, standalone or parallel, processes execution, as well as tools designed for scripting distributed computing experiments on any computing platform. After discussing the EXECO internals, we illustrate its interest by presenting two experiments dealing with virtualization technologies on the Grid'5000 testbed.**

I. INTRODUCTION

Like in other experimental sciences, the characterization of all parameters as well as the recording of all operations is mandatory for Computer Sciences. This is the only way to ensure accuracy, reproducibility and verifiability of computing experiments. Such a "laboratory notebook" is particularly critical for cloud testbeds as conducting experiments on such platforms involves the execution and management of several local and remote unix processes, the handling of hardware and software failures and the much more complex characterization of distributed experimental conditions. As the number and complexity of all these tasks grows, automation of the experiments becomes mandatory.

Cloud testbeds, as well as many modern computing platforms, are distributed environments involving multiple programming languages, development paradigms, hardware, and software. At the lowest level, the only common factor is the unix process, local or remote. An important difficulty in performing automated experiments is to finely control the execution and life-cycle of these distributed processes.

In this paper, we present EXECO[1], a generic toolkit for conducting and controlling large-scale experiments autonomously. EXECO has been designed for easy and efficient management of many distributed unix processes while providing dedicated tools for experiment scripting.

To illustrate the benefits of EXECO, we present experiments involving virtual machines on Grid'5000. Two experiments campaigns that have been recently performed are discussed: the first one studies the impact on performance of collocating virtual machines while the second one focuses on the virtual machines migration performance. Although each one investigates a particular concern of virtual machine management, both include more than one hundred experiments that have been automatically operated by EXECO.

Last but not the least, it is noteworthy that although a dedicated module has been developed to take advantage of EXECO on Grid'5000 [1], it can also be extended to other testbeds by implementing an appropriate wrapper that interacts with the API of the targeted testbed.

The remainder of this paper is structured as follows. In Section II, we give an overview of the Grid'5000 instrument. Section III introduces the EXECO toolkit and the extension that are specific to Grid'5000. The two experiment campaigns are discussed in Section IV. Finally, Section V presents related work and Section VI concludes the article.

II. GRID5000

The Grid'5000 testbed has been designed to support experiment-driven research in parallel and distributed systems. By leveraging the Grid'5000 platform, users may perform experiments on all layers of the software stack of distributed infrastructures, including high-performance computing, grids, peer-to-peer, and cloud computing architectures [1], [2]. Mainly located in France, Grid'5000 is composed of 26 clusters, 1,100 nodes, and 7,400 CPU cores, with various generations of technology, on 9 physical sites interconnected by a dedicated 10 Gbps backbone network.

The core steps identified to run an experiment on Grid'5000 are (1) finding and reserving suitable resources for the experiment and (2) deploying the experiment apparatus on the resources. Finding suitable resources can be approached in two ways: either users browse a description of the available resources and then make a reservation, or they describe their needs to the system that is in charge of locating appropriate resources. We believe both approaches should be supported, and therefore a machine-readable description of Grid'5000 is available through the reference API. It can be browsed by using a web interface or by running a program over the API. At the same time, the resource scheduler on each site is fed with the resource properties so that a user can ask for resources describing the required properties *e.g.*, 25 nodes connected to the same switch with at least 8 cores and 32 GB of memory. Once matching resources are found, they can be reserved either

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action and the INRIA large-scale initiative Hemera, with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr). Takahiro Hirofuchi is a visiting researcher from National Institute of Advanced Science and Technology (AIST) of Japan in the context of the French ANR project SONGS (11-INFRA-13).

[1]http://execo.gforge.inria.fr/

for exclusive access at a given time or for exclusive access when they become available. In the latter case, a script is given at reservation time, as in classical batch scheduling.

Several tools have been developed to facilitate experiments. Grid'5000 users select and reserve resources with the OAR batch scheduler [3] and they can install their own system image on the nodes (without any virtualization layer) using Kadeploy [4]. Experiments requiring network isolation can use KaVLAN to reconfigure switches and isolate nodes from the rest of the testbed. Several monitoring tools (resource usage on nodes with Ganglia, energy consumption) are also available. All tools can be accessed by a REST API to ease the automation of experiments using scripts.

Different approaches for deploying the experimental apparatus are also supported. At the infrastructure level, users either utilize the preconfigured environment on the nodes, called the production environment and configure it according to their needs, or they can directly install their own environment (*i.e.*, a disk image to be copied on the node). Whatever approach used for the first two steps described here, access to resources (sites and nodes) is done through *ssh*.

III. The Execo toolkit

Execo is a Python API. It is not a directly executable experiment engine which takes a declarative experiment description as input, but rather a rapid experiment development toolkit which offers modular building blocks. Programming experiments in a generic language (Python) offers greater flexibility, allowing the use of all kinds of language constructs, such as tests, loops, exception blocks *e.g.*, for releasing resources, even in case of failures, etc.

The Execo toolkit is divided into three modules: `execo`, the core of Execo, `execo_g5k`, the interface to Grid'5000 services and virtual machines management services, `execo_engine`, the tools for experiments scripting. These modules are separated, so that Execo usage is modular: modules `execo` and `execo_engine` can be used outside Grid'5000. Some code which only needs the functionality of remote process control can use only module `execo`. These three modules have the minimal dependencies required, in order to allow an easy installation on any compute node or virtual machine.

A. Module `execo`: core API for unix process management

The Execo core offers abstractions of local or remote, standalone / sequential / parallel unix processes. It allows the transparent, asynchronous, concurrent, fine-grained, and efficient control of several unix processes as follows:

- *transparent*: local or remote processes are handled similarly (with ssh or ssh-like connector).
- *asynchronous*: to start some processes, then do something else, then later take back control of the processes by waiting for their termination or by killing them.
- *concurrent*: to control groups of parallel processes on groups of remote hosts.
- *fine-grained*: for each process or group of parallel processes, all attributes of the processes are available: start date, end date, state, return code, stdout, stderr.
- *efficient*: all I/O to local or remote processes are handled with a single thread doing asynchronous I/O (with `select` or `poll` system calls). This allows very efficient handling of more than 1000 parallel ssh connections. To scale further, Execo can transparently use TakTuk [5] which builds a tree of parallel ssh connections, and which can support thousands of parallel remote ssh processes.

As one of the main goals is *rapid* experiment development, Execo is designed to allow the expression of complex experiment scenarios in a very straightforward way. One consequence is that for most cases, experiment conducting scripts are centralized rather than distributed. Our experience so far is that Execo performance and scalability are sufficient for centrally controlling the processes of all experiments we performed. To illustrate this, figure 1 shows the *total duration* of Execo parallel remote connections as a function of the number of parallel connections. The *total duration* is defined as the time between the start of all the remote connections, each running the simple command `uname -a`, and the close of the last connection. One curve shows the durations of real parallel ssh connection processes (Execo `Remote`), while the other curve show the durations of remote connections through TakTuk (Execo `TaktukRemote`), taking advantage of TakTuk connection tree. Both curves thus show the aggregated effect of several factors such as the number of concurrently running unix processes, the number of concurrent TCP connections, the overhead of Execo asynchronous I/O layer, etc. This figure shows the performance which can be expected from Execo. Having an Execo abstraction on top of parallel ssh or TakTuk is a real gain: usually, when developing the Execo code for an experiment, it's simpler to prototype and debug everything with a limited number of remote hosts, using parallel ssh, and when there's a need to scale up and run the real experiment with many hosts, one can switch almost transparently to use TakTuk.

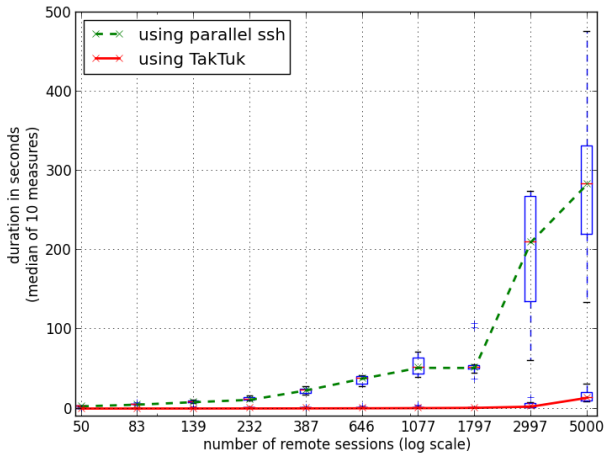


Figure 1: Total durations of Execo parallel remote sessions

All these services are offered as classes in the `Process` and `Action` class hierarchies. A `Process` is a single local or remote process. An `Action` models groups of concurrent processes. An `Action` instance takes as parameters a command to run, and a list of hosts on which to run this command. Process command lines can be the subject of *substitutions*, a syntax allowing each command line sent to a group of hosts to be computed as a function of the host. Some `Action` classes also allow composing sub-`Action` concurrently or sequentially, allowing the building of more high level abstractions of complex distributed behaviors. All these `Process` and `Action` classes support registering some handlers whose events are triggered when receiving output on process stdout / stderr or on process life-cycle events. Some `Action` classes also implement file transfers with parallel scp, with TakTuk, or with chained TCP connections (an efficient strategy for sending big files to many hosts [6], such as when broadcasting operating system or virtual machine disk images).

An important feature of EXECO is the log system, which by default emits just as much log as is needed to avoid cluttering the log files and at the same time allow postmortem debugging of failed executions. In default mode, only failed process are logged with all their attributes and excerpts of their stdout / stderr.

B. Module `execo_g5k`: cloud services on Grid'5000

It offers an interface to the Grid'5000 cloud services. It allows to find details about the hardware resources on the platform from the Grid'5000 reference API, to manage jobs (listing, submission, deletion, waiting a job to start) on a site with OAR [3] and on several sites with OARGrid, to deploy operating system images with Kadeploy3 [4]. These features allow a fine-grained control on the experimental conditions.

Recently, the Grid'5000 platform has evolved to add KVM-based virtualization capabilities [2]. A prototype framework to deploy a large number of hosts and virtual machines using `libvirt` has been created and used to run large scale experiments involving around 5000 virtual machines [7]. We then added these functionalities to the EXECO API to automate hosts deployment and configuration, and simplify virtual machine management (disks creation, installation, distribution, start and destroy, migration). We use a Debian Wheezy with `qemu-kvm` (1.1.2) and an updated version of `libvirt` (1.0.6). Thanks to the usage of TakTuk, we are able to centrally control the hosts and the virtual machines (in current experiments, we managed up to 200 physical hosts, and 5000 VMs. We plan to increase these numbers).

C. Module `execo_engine`: experimental automation

The EXECO `Engine` is a class hierarchy that controls the control flow of experiments. The base class does the minimum: setting up or reusing an experiment directory, taking care of log files. User implemented sub-classes can implement full experiments, or for more advanced users, can implement reusable experiment engines which can then be further refined or customized in sub-classes.

For experiments involving the systematic exploration of various parameter (or factors) values, the `ParamSweeper` offers an automatic iterator over the cartesian product of all parameters. It defines a syntax for describing parameters and the set of values among which the experiment should iterate. It is then able to generate the full experimental plan exploring all the parameters combinations. It can track the progress of the experiment, check-pointing this progress to disk, so that an experiment can be interrupted and later restarted.

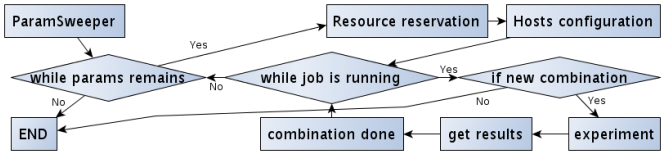


Figure 2: The `Engine` workflow used for systematic experimentation

IV. EXPERIMENTS

While we do not have objective metrics to highlight EXECO ease of use for preparing and conducting experiments, we observed the usage of EXECO among 10 persons (Ph.D, post-docs, engineers) and in most cases, users found it has shortened their development, the debugging time and more generally the global time to complete their experiments. Before using EXECO, users where struggling doing some experiments, where most of their actions where performed interactively, by copy-pasting and adapting commands in several interactive shells. This is far from reproducible, and it was very difficult to interactively conduct experiments when, for example, resources are available only during nights or week-ends. After switching to EXECO, they were able to automate complex experiment workflows and conduct large scale experiment campaigns.

To illustrate this point, we chose to describe two scenarios that require to complete a large number of experiments under different parameters. From the software engineering point of view, the EXECO `engine` class hierarchy allowed an expert engineer to develop the common experiment engine class, dedicated for a class of virtual machines experiments, taking care of various complex setup operations, and offering research scientists a framework in which they only had to subclass the VM experiment engine and override the specific methods for performing the two experiments described. The custom experiment engine class follows the workflow shown in figure 2. It automatically explores a parameters range for an experiment defined by a sequence of actions. The base workflow consists in : the creation of a `ParamSweeper` to have an object to iterate over; a loop while there are parameters combinations to perform ; within this loop, a resource reservation, a deployment and a loop that gets a new combination and executes the sequence of actions until the job ends or all combinations have been treated. Thanks to this algorithm, new jobs are spawned as long as there are remaining parameter combinations to explore. We have then created two engines derived from this general one to study two different scenarios: the impact on performance of collocating VMs; and the effect of memory load on VMs live migration time.

A. Impact of collocating VMs on performance

The first experiment aims to evaluate the impact on performance of collocating multiple VMs on the same physical

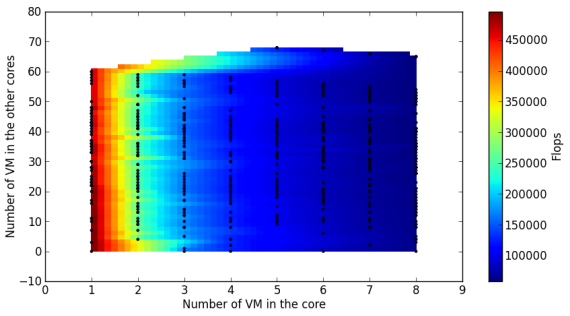


Figure 3: Impact on performance of VM collocations

machine (PM) in order to extract and validate a multi-core processor performance model for the SimGrid [8] toolkit [2]. To be able to construct and validate such model, we need to run a large set of experiments covering the different configuration of VMs consolidation on a single PM.

Our approach is incremental, first, we study performance interference for a set of VMs with a single VCPU. Moreover, we limit ourselves to a hardware configuration, *i.e.* a PM with 2 Intel Xeon E5-2630 (2x6cores with a shared L3 15MB cache). Finally, we use a compute-intensive benchmark [3] (a part of Linpack). We run the benchmark in all the running VMs for 5 minutes for each combination. We limit ourselves to a number of VMs per core sets between 0 and 8. This is due to the fact that we want to run VMs with a least 256Mb of RAM and each core has 2GB of RAM. Accordingly, we have 9.916×10^{149} possible combinations of VM configurations to run. Doing a parameter sweeps on all the combinations will be too long. Accordingly, we randomly pick a subset of configurations from this search space.

The experimental plan is to pick N different combinations from the search space (with $N = 100$ for the first batch of experiments) and to execute for each combintaions the following workflow. *1)* We create the VM disk images. *2)* We configure through `libvirt` each VM, pin them to a core and start them. *3)* When all the VMs have been started, we install the benchmark *i.e.*, put the source and compile them. *4)* We run the benchmark and redirect its output to a file. *5)* We kill all the benchmarks after 5 minutes *6)* We retrieve the output files in a specific directory for each conbination *7)* We destroy all the VMs. Doing so without EXECO would have required to enter a lot of repetitive and error-prone commands (at least 15 for each VM in each configuration). Doing it with a shell script will be difficult due to the lack of proper parallelization. With EXECO, the management of VMs is hidden and we can focus on the experimental plan. The whole engine, with parallelization, is only 120 lines of Python.

Figure 3 shows the first results of this experiment. The x-axis shows the number of VMs running on the same core. The y-axis presents the number of VMs running on other cores. The color displays the number of FLOPS the VM gets (higher is better). As one can see, the correlation between the speed (in FLOPS) of a VM and the number of VMs sharing the same core is strong (x-axis). With our compute-intensive benchmark, we can see that a VM that runs concurrently with other VMs, has a number of FLOPS more or less equal to the FLOPS of running alone on the core divided by the number of other VMs on the same core. Accordingly, we can almost make a very basic performance model $FLOPS_{VM} = \frac{FLOPS_{CORE}}{|VM_{CORE}|}$ with $FLOPS_{VM}$ the number of FLOPS a VM can have, $FLOPS_{CORE}$ the number of FLOPS a core can provide, VM_{CORE} the list of running VMs on a core $CORE$. But this model is not accurate, as we can see a weak correlation between the FLOPS a VM gets and the number of VMs running on other core (y-axis). This correlation can be due to micro-architectural components sharing such as cache, I/O and memory bus. Therefore, we need to run new experiments to characterize the impact on performance of the sharing of micro-architectural components to refine our models. Thanks to EXECO, taking into account these new requirements is straightforward. We just need to modify our configuration generator to create VM combinations that are aware of the micro-architecture. Without EXECO, we could be required to redo even more complex experiments or rewrite a large part of shell scripts. In the future, we will also run the same experimentation with other type of benchmarks (memory-intensive, I/O-intensive workloads) and on other platforms with different micro-architectures. By doing so, we will be able to validate (and refine) the model we will have constructed from the first batch of experiments.

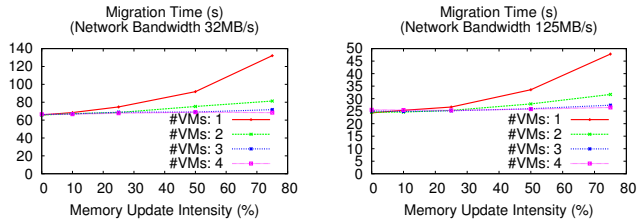
B. Experiments on VM live migration performance

As the second use case of EXECO, we performed long-duration experiments measuring live migration performance under various conditions. Live migration is a mechanism to relocate the execution of a VM to another PM without stopping the guest operating system. We are developing the virtualization extension of SimGrid, allowing users to correctly simulate live migration behaviors. Live migration performance of a VM *i.e.*, migration time and generated network traffic, is significantly impacted by the activities of other VMs and workloads in a simulated system. The experiments are intended to clarify characteristics of live migrations under various conditions. Results give us essential information to determine the design criteria of the VM model in SimGrid.

The experiments tested 120 parameter settings, each of which is a combination of 5 types of parameters *i.e.*, the memory sizes of a VM (2GB, 4GB and 8GB), the available network bandwidths (32MB/s and 125MB/s), the numbers of co-located VMs (1, 2, 3, and 4), and the memory update speeds of the VM (0, 10, 25, 50, and 75% of the available network bandwidth). The different available network bandwidths for a live migration are tested to see how a live migration will be impacted by other network traffic sharing the same network link. The different numbers of co-located VMs are also tested. A live migration will be affected by the activities of the other VMs running on the PM of a migrating VM. To observe the worst case, all VMs are pinned to the first CPU core of a PM. The different memory update speeds are defined against the current available bandwidth. The precopy algorithm of the live migration mechanism [9] recursively transfers updated

[2]This work is a part of a larger project to extend SimGrid for next generation system from HPC to Clouds: ANR INFRA SONGS http://infra-songs.gforge.inria.fr/

[3]The binary is available online: http://graal.ens-lyon.fr/~jrouzaud/files/kflops.tgz



(a) Network Bandwidth 32 MB/s (b) Network Bandwidth 125 MB/s

Figure 4: Live migration of a 2 GB VM

memory pages to the destination until the live migration ends. The memory update speed of a migrating VM will significantly impact the repetition times of memory transfers. In theory, to end a live migration, the memory update speed must be slower than the available network bandwidth.

The EXECO script of the experiments is just 200 lines of Python. It performs the following workflow: *1)* Reserve 2 PMs and a private network. *2)* Deploy experiment environments to the PMs. *3)* Pick up a new combination of parameters. *4)* Generate the settings of VMs, such as disk image files and libvirt XML files. *5)* Launch VMs on a PM. Launch a memory update program on each VM. *6)* Invoke a live migration of a VM to the other PM. Measure each migration time. Repeat a migration 10 times. *7)* Destroy VMs. Return to the 3rd step.

The workflow repeats the 3rd-7th steps until all combinations are done. At the end, we obtain 1200 migration durations *i.e.*, 10 times per each combination. Figure 4 shows a subset of obtained results. At a glance, without other co-located VMs, the memory updates of the VM caused the exponential increase of live migration times. However, as the number of co-located VMs increased, the impact of the memory update intensity became less serious. Although the larger network bandwidth shortened live migration times, the shape of upward curves of migration times was the same as that of the narrower bandwidth. These results gave us important hints to design the VM migration model in SimGrid. The simulation system will require the accurate model of memory update intensity and CPU resource contention of VMs to correctly simulate migration cost. Details will be reported in our upcoming paper.

The EXECO script greatly contributed to reducing the cost of experiments. The overall experiments for all combinations required approximately 2 days. Obviously, it is difficult to manually perform this size of experiments. Once launching the script, we did not need to pay any attention to the experiments. Moreover, we could launch the script on other clusters simultaneously.

The script successfully dealt with an error during the experiments. The Grid5000 charter asks fair use of resource; users should not keep their nodes for a long period of time. During the experiments, we reserved nodes for 6 hours. When a 6-hours reservation was expired, the script detected an error of an experiment, automatically reserved new PMs, deployed experimental environments, and restarted remaining experiments. This error handling mechanism will work also for other hardware failures, and strongly support long-duration, large-scale experiments.

V. RELATED WORK

A. Parallel remote processes control

There are many parallel ssh tools, such as *TakTuk/Kanif*[4], *pssh*[5], *clusterssh*[6], *sshpt*[7], *cssh*[8], *dsh*[9], etc. These tools usually come from the server administration community and take the form of command line tools, which are limited to being scripted in shell scripts, and which interface is limited to their standard output and error and their return code.

There are also some more advanced administration tools such as *capistrano*[10] or *ansible*[11] which allow organizing remote commands in sets of reusable workflows, but these tools offer a limited API through a fixed formalism for describing the system administration tasks workflows.

A few libraries exist for controlling parallel remote ssh connections, such as *python-fabric*[12] or ruby *Net::SSH::Multi*[13], but they lack the asynchronous control offered by EXECO: with these libraries, the client code can only start some ssh processes and wait for their termination, whereas in EXECO you can start some processes, then do something else *e.g.*, start other processes, then take back control of the processes to wait for them or kill them. EXECO makes this possible because it handles unix processes (I/O, lifecycle) in a separate thread. Actually, to our knowledge, only EXECO treats individual or groups of parallel local or remote processes as API objects whose I/O and lifecycle can independently be controlled.

B. Experiment engines

Many experiment engines exist, with various characteristics and goals.

OMF [10] is a control, measurement and management framework for networking testbed. It is tightly integrated with its testbed platform and services: it encompasses the resource management or node deployment services, as well as router or VLAN management. It is thus much more heavyweight than EXECO because the whole experiment platform needs to be setup with OMF, whereas in the case of EXECO, you can just install it on some nodes. For example, it is easy to use EXECO in the context of an experiment running on a commercial cloud, whereas it is impossible for OMF. Experiment workflows are described in a custom domain specific language (DSL). It consists in starting some programs at given points in time or on given measurement events. Compared to EXECO and Python, this DSL is much more limited.

Plush (PlanetLab User Shell) [11] and its successor *Gush (GENI User Shell)* [12] are frameworks to control the deployment and execution of applications on the PlanetLab and GENI testbeds. They are moderately integrated with their platforms since using them on another testbed requires an

[4]http://taktuk.gforge.inria.fr/kanif/
[5]http://code.google.com/p/parallel-ssh/
[6]http://sourceforge.net/apps/mediawiki/clusterssh/
[7]http://code.google.com/p/sshpt/
[8]http://cssh.sourceforge.net/
[9]http://www.netfort.gr.jp/~dancer/software/dsh.html.en
[10]https://github.com/capistrano/capistrano
[11]http://www.ansibleworks.com/
[12]http://docs.fabfile.org/en/1.6/
[13]http://net-ssh.rubyforge.org/multi/v1/api/

adaptation. They are more heavyweight than EXECO (more dependencies, configuration), but only provide interactive or xml-rpc interfaces, but no direct API. The commands provided act at a coarse grain *e.g.*, you can run a shell command on all connected nodes or on the nodes matching a given regular expression, but you can not select the nodes programmatically, nor control the lifecycle of the shell command run. Plush and Gush rely on a declarative language (XML, rather than general purpose programming language) for application specification, and they then take care of resource discovery and acquisition, then application deployment and maintenance. The experiment workflow is thus much more constrained than with EXECO.

FCI (Federated Computing Interface) [13] is an API for controlling resources federated from different cloud providers, for experimentally driven research. It takes care of federated resources reservation and configuration, as well as federated resources metrology. Experiment scenarios are expressed in a declarative DSL, which is limited compared to EXECO and Python. Experiment workflows are thus much more constrained than with EXECO. FCI offers no remote process execution API.

cigri[14] manages the execution of multi-parametric experiments. It takes a job description, with the command to run, the resources needed, and the explicit list of parameters to explore. It then reserves the resources (taking care of limiting its resources usage), starts the jobs, checkpoints the progress. Compared to EXECO, its workflow is fixed and it only deals with the parameter sweeping, everything else is left to the client provided executable. Also, cigri is tightly integrated with the platform on which it is hosted (it relies on using the OAR batch scheduler).

Expo [14] is a DSL (on top of Ruby) for the management of experiments. It has many similarities with EXECO (actually some EXECO ideas came from previous versions of Expo).

Overall, all these tools are limited to some platforms, while EXECO can be used on any platform *e.g.*, not only clouds, since its basic block is the unix process. It is designed to be a toolkit, and its remote process control core component, albeit being low level, allows automating almost everything. For example, we could imagine to automate the deployment of a full OMF platform with EXECO, or its testing (in a continuous integration approach).

VI. CONCLUSION

Performing reproducible experiments or running tests on cloud computing platforms is a complex task which needs appropriate tools to be automated.

In this paper, we presented the EXECO framework, a library aiming at relieving researchers of the burden of dealing with low-level details such as resources management, failures, when conducting distributed computing experiments. Leveraging a modular design and advanced tools to interact with remote resources, EXECO provides a simple API that allows the concise development of very complex experiment scenarios. To our knowledge, no other library or experiment engine offers the kind of asynchronous control that EXECO offers. We illustrated the power of EXECO by discussing two scenarios that include up to 120 experiments performed 10 times each (leading to 1200 experiments that have been autonomously performed and controlled by the EXECO engine when resources were available). As future work, we plan to extend the parameter sweeping capabilities of EXECO and to extend it to work on other platforms than Grid'5000.

[14] http://cigri.imag.fr/

REFERENCES

[1] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab *et al.*, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[2] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to grid'5000," INRIA, Rapport de recherche RR-8026, Jul. 2012. [Online]. Available: http://hal.inria.fr/hal-00720910

[3] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard, "A batch scheduler with high level components," in *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, vol. 2. IEEE, 2005, pp. 776–783.

[4] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum, "Kadeploy3: Efficient and Scalable Operating System Provisioning," *USENIX ;login:*, vol. 38, no. 1, pp. 38–44, Feb. 2013.

[5] B. Claudel, G. Huard, and O. Richard, "Taktuk, adaptive deployment of remote executions," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*. ACM, 2009, pp. 91–100.

[6] Y. Georgiou, J. Leduc, B. Videau, J. Peyrard, and O. Richard, "A Tool for Environment Deployment in Clusters and light Grids," in *Second Workshop on System Management Tools for Large-Scale Parallel Systems (SMTPS'06)*, Rhodes Island, Greece, apr 2006.

[7] F. Quesnel, A. Lèbre, J. Pastor, M. Südholt, and D. Balouek, "Advanced validation of the dvms approach to fully distributed vm scheduling," in *ISPA' 13: The 11th IEEE International Symposium on Parallel and Distributed Processing with Applications*, Melbourne, Australie, Jul. 2013. [Online]. Available: http://hal.inria.fr/hal-00817369

[8] H. Casanova, A. Legrand, and M. Quinson, "Simgrid: a generic framework for large-scale distributed experiments," in *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, ser. UKSIM '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 126–131. [Online]. Available: http://dx.doi.org/10.1109/UKSIM.2008.28

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, 2005, pp. 273–286.

[10] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar, "Omf: a control and management framework for networking testbeds," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 54–59, 2010.

[11] J. R. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat, "Remote control: Distributed application configuration, management, and visualization with plush." in *LISA*, vol. 7, 2007, pp. 1–19.

[12] J. Albrecht and D. Y. Huang, "Managing distributed applications using gush," in *Testbeds and Research Infrastructures. Development of Networks and Communities*. Springer, 2011, pp. 401–411.

[13] C. Tranoris, "Adopting the dsm paradigm: defining federation scenarios through resource brokers for experimentally driven research," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*. IEEE, 2011, pp. 1140–1147.

[14] B. Videau, C. Touati, and O. Richard, "Toward an experiment engine for lightweight grids," in *MetroGrid workshop : Metrology for Grid Networks*. ACM publishing, Oct. 2007.