

# EXECO tutorial

Grid'5000 school, Lyon, June 2014

Matthieu Imbert   Laurent Pouilloux

INRIA/LIP ENS-Lyon

05/06/2014

- 1 introduction
- 2 execo, core module
- 3 execo\_g5k, Grid'5000 interface
- 4 execo\_engine, experiment engine
- 5 examples
- 6 conclusion

1 introduction

2 execo, core module

3 execo\_g5k, Grid'5000 interface

4 execo\_engine, experiment engine

5 examples

6 conclusion

# Context overview

- ▶ Conduct a computer sciences experiment, or perform a system or admin task (e.g. install / configure / start / stop some software)
- ▶ On local or remote hosts
- ▶ Control complex workflows of system operations

# Some possible solutions

## Interactive local or ssh sessions

- ▶ painful
- ▶ error prone
- ▶ not reproducible

## Shell scripting

- ▶ limited syntax
- ▶ poor unix process control, especially through ssh
- ▶ difficult asynchronous management

# The EXECO solution

A python API for controlling **local or remote unix processes**, for scripting workflows of operations on distributed systems.

- ▶ easy, fast, and intuitive to develop. You directly write your script the same way you think about it.
- ▶ fine grained control, e.g. easy to get simultaneously stdout, stderr, exit code.
- ▶ asynchronous, e.g. start process A, start process B, wait B, kill A
- ▶ optimized, scalable: transparently scales to the order of thousands of parallel remote processes
- ▶ powerful log system:
  - ▶ convenient defaults giving the right amount of output
  - ▶ live or post-mortem analysis of complex distributed workflows
- ▶ efficient file transfers

# EXECO characteristics

## unix processes

Based on forking or ssh-like tools for executing processes. Can use taktuk for scaling up.

## file transfers

Can use parallel scp, taktuk, or an efficient chained broadcast.

## Grid'5000 interaction

A python module offers an API to interact with Grid'5000 services

- 1 introduction
- 2 execo, core module**
- 3 execo\_g5k, Grid'5000 interface
- 4 execo\_engine, experiment engine
- 5 examples
- 6 conclusion



- ▶ EXECO kernel:
  - ▶ a thread handles the lifecycle and outputs of all handled operating system processes
  - ▶ multiplexed I/O: poll on linux, select on osx → very efficient
- ▶ allows client code to control process asynchronously: start a process, wait for it, kill it
- ▶ lifecycle and I/O of processes handled with callbacks. Default callbacks:
  - ▶ collect stdout / stderr
  - ▶ handle process death: exit code, duration
- ▶ default callbacks sufficient most of the time. But may supply custom callbacks if needed

This is one of the main strengths of EXECO which distinguishes it from other libraries, and which allows easily coding workflows that would be impossible or difficult to code with other libraries.

# Process

Hi level class hierarchy for local processes `Process` or remote processes `SshProcess`

- ▶ control: `start`, `wait`, `kill`
- ▶ state: `error`, `exit_code`
- ▶ I/O: `stdout`, `stderr`
- ▶ etc.

# SshProcess

**SshProcess** inherits from **Process**. It targets a **Host**:

- ▶ hostname or address
- ▶ user (*optional*)
- ▶ port (*optional*)
- ▶ keyfile (*optional*)

All remote connections also use default **ConnectionParams** or user-supplied one:

- ▶ ssh / scp executables
- ▶ ssh / scp options
- ▶ address rewriting hook (*eg. handy for going through an ssh alias*)
- ▶ etc.

# Logging

- ▶ standard python logger
- ▶ default setup ideal for most situations:
  - ▶ quiet
  - ▶ except: warning (with detailed infos) for all processes with an error (start error, non zero exit code)
  - ▶ can explicitly tell some processes to not log warnings on such conditions
- ▶ eases post-mortem analysis of problems in long experience logs

# Process states

- ▶ carefully designed process state system
- ▶ most of the time: **Process.ok**
- ▶ means:  $process\ running \vee (process\ finished \wedge \neg error \wedge \neg timeout \wedge exit\ code = 0)$
- ▶ can explicitly tell some processes to be considered “ok” even if  $exit\ code \neq 0$
- ▶ can access each detailed state attributes as well as process start date or end date

# Actions

- ▶ set of parallel remote processes to a list of hosts
- ▶ same as process:
  - ▶ control: `start`, `wait`, `kill`
  - ▶ state: `ok`, `error`, `exit_code`, `start_date`, `end_date`
- ▶ access to individual processes (`stdout`, `stderr`)
- ▶ as for remote processes: connect to list of `Host`, using default or specific `ConnectionParams`

# Taktuk transparent usage

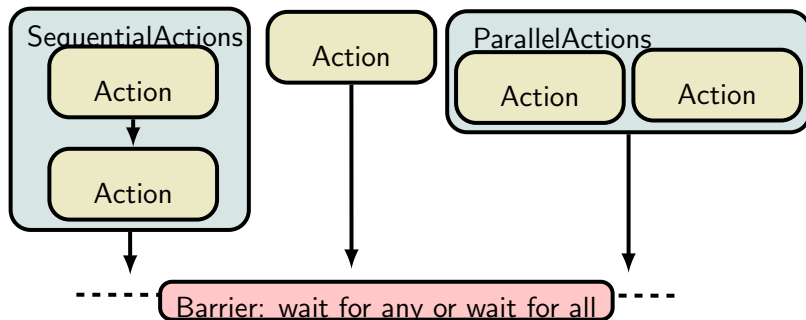
- ▶ taktuk can be used instead of using parallel ssh
- ▶ same Action interface → same code can switch to taktuk when facing scalability issues
- ▶ can use a “taktuk frontend”: a host to connect to with ssh, on which to run taktuk → allow for example running taktuk inside g5k, controlled by outside

A factory can be used to centrally control what kind of action to instantiate: taktuk or classic parallel ssh

- ▶ parallel file copies to / from nodes
- ▶ there are versions using taktuk (but with limitations)
- ▶ **ChainPut**: efficient chained broadcast for copying large data to several remote hosts.



# Action workflows



- ▶ workflows with `ParallelActions`, `SequentialActions`
- ▶ `wait_any_actions`, `wait_multiple_actions`

# Substitutions

A very convenient syntax to express variations in the command line executed in parallel on the set of remote hosts of an Action.

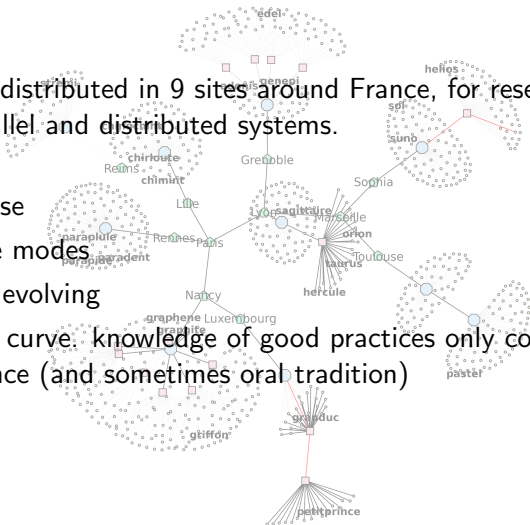
- ▶ all occurrences of the literal string `{{{host}}}` are substituted by the address of the connected **Host**.
- ▶ all occurrences of `{{<expression>}}` are substituted in the following way: `<expression>` must be a python expression, which will be evaluated in the context (globals and locals) of the Action declaration, and which must return a sequence. `{{<expression>}}` is replaced for all individual remote host by `<expression>[index % len(<expression>)]`.

- 1 introduction
- 2 execo, core module
- 3 execo\_g5k, Grid'5000 interface**
- 4 execo\_engine, experiment engine
- 5 examples
- 6 conclusion

An infrastructure distributed in 9 sites around France, for research in large-scale parallel and distributed systems.

Powerful, but

- ▶ complex to use
- ▶ lots of failure modes
- ▶ continuously evolving
- ▶ long learning curve: knowledge of good practices only comes with experience (and sometimes oral tradition)



- ▶ oar / oargrid
- ▶ kadeploy
- ▶ Grid'5000 API
- ▶ planning, charter, funk

## Submission and deletion

- ▶ `OarSubmission(resources, walltime, job_type, ...)`
- ▶ `oarsub([OarSubmission(...), frontend])`
- ▶ `oardel([(job_id1, frontend1), (job_id2, frontend2)])`

## Job information

- ▶ `get_oar_job_info, get_oar_job_nodes`
- ▶ `wait_oar_job_start, get_current_oar_jobs`

## Network information

- ▶ `get_oar_job_subnets`
- ▶ `get_oar_job_kavlan`

# oargrid

## Submission and deletion

`oargridsub`, `oargriddel`

## Job information

`get_current_oargrid_jobs`, `get_oargrid_job_info`,  
`get_oargrid_job_oar_jobs`, `get_oargrid_job_nodes`

## Flow control

`wait_oargrid_job_start`

## deploy

- ▶ possible to not redeploy already deployed nodes
- ▶ able to retry deployments several times if not enough resources
- ▶ user-provided callback for deciding if enough resources (even with complex topology)
- ▶ supports custom checks to detect if node deployed



# Grid'5000 API

## Specific functions

`get_g5k_sites,`      `get_g5k_clusters,`      `get_g5k_hosts,`  
`get_site_clusters,`   `get_cluster_hosts,`   `get_cluster_site,`  
`get_host_cluster,` `get_host_site,` `group_hosts`

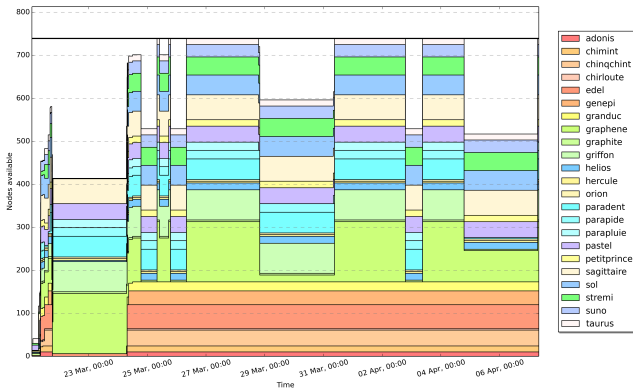
## Gathering resources information

`get_host_attributes,`                      `get_cluster_attributes,`  
`get_site_attributes,` `get_resource_attributes`

## planning, charter

- ▶ Retrieve the schedule planning of hosts / vlans, from a Grid'5000 site
- ▶ finds time slots allowing scheduling of jobs with a given need of resources and wall-time
- ▶ 3 search modes (searching on a time range, 1 month by default)
  - ▶ **date**: find how much resources are available at a given date for a given wall-time
  - ▶ **free**: find the first time-slot the given resources are available for a given wall-time
  - ▶ **max**: find the time-slot where the maximum number of resources are available for a given wall-time
- ▶ charter: for a given day, get the remaining time available per cluster for submitting jobs during work hours.

- ▶ Based on modules planning and charter, this is a command line tool allowing manually finding time-slots with sufficient resources for reserving nodes.
- ▶ already installed on Grid'5000
- ▶ alternative to disco



- 1 introduction
- 2 execo, core module
- 3 execo\_g5k, Grid'5000 interface
- 4 execo\_engine, experiment engine**
- 5 examples
- 6 conclusion

# module `execo_engine`

## class `Engine`

- ▶ a class hierarchy implementing basic experiment life cycle
- ▶ intended to allow providing specific custom experiment engine that can be sub-classed and specialized (e.g. `vm5k`, `g5k_cluster_engine`)

## Parameter sweeping

- ▶ to explore the combinations of several parameters: a syntax allows expressing these parameters and generate the list of all parameters combinations (the Cartesian product)
- ▶ class `ParamSweeper` provides a check-pointed, thread-safe, process-safe iterator over the parameter combinations, allowing to track the progress of an experiment.

# Example of a generic experiment engine

## `g5k_cluster_engine`:

- ▶ a reusable EXECO engine, automatizing the workflow of submitting jobs in parallel to Grid'5000 clusters / sites
- ▶ well suited for bag-of-task kind of jobs, where the cluster is one of the experiment parameter, e.g. benching flops, benching storage, network, etc.

- 1 introduction
- 2 execo, core module
- 3 execo\_g5k, Grid'5000 interface
- 4 execo\_engine, experiment engine
- 5 examples**
- 6 conclusion

# openstack deployment on Grid'5000

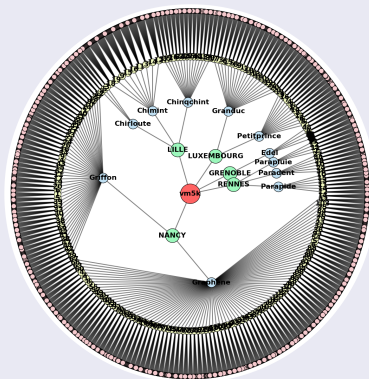
courtesy of Guillaume Verger

- ▶ a single EXECO script, automatizing the whole deployment of openstack, as described here:  
[https://www.grid5000.fr/mediawiki/index.php/Deploying\\_OpenStack\\_using\\_KaVLAN](https://www.grid5000.fr/mediawiki/index.php/Deploying_OpenStack_using_KaVLAN)
- ▶ adds several options for controlling the deployment topology
- ▶ supports deploying DIET on top of openstack



## basic usage

a single script, automatizing the deployment of a large number of virtual machines on Grid'5000, providing several options to control physical hosts and virtual machines



## advanced usage

a reusable EXECO engine that performs a user defined experiment on these virtual machines, e.g. live migration, multi-core performance.

## Building a model for VM live migration

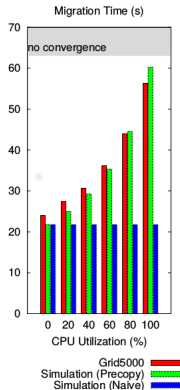
► **Question:**

## Impact of memory update intensity, network bandwidth, collocated VMs

► **Methodology:**

- ▶ Generating parameter combinations (bw, number of VMs, memory update)
- ▶ Reserving 2 PMs and a private network
- ▶ Launch VMs and the memory update program
- ▶ Migrate VMs from a PM to the other one (10 times)

	VM 2G	VM 4G	VM 8G
G5K	55.4 (1.9)	96.7 (2.0)	137.5 (0.5)
SimGrid	55.8	93.0	140.1



T. Hirofuchi, A. Lèbre, L. Pouilloux: "Adding a Live Migration Model Into SimGrid, One More Step Toward the Simulation of Infrastructure-as-a-Service Concerns" *5th IEEE International Conference on Cloud Computing Technology and Science 2013*

# Conclusion

EXECO strong points:

- ▶ pragmatic approach: **rapid experiment development**
- ▶ **efficient, user-friendly** (e.g. logs, asynchronous process control, transparent local or remote unix processes, efficient file broadcasting, scalable, thanks to taktuk, etc.)
- ▶ flexible: no experiment model imposed
- ▶ brings the power of a general purpose language into the world of experiment prototyping and conducting
- ▶ may also use it for admin tasks, unit testing

Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lèbre, Takahiro Hirofuchi "Using the EXECO toolbox to perform automatic and reproducible cloud experiments" *1st International Workshop on UsiNg and building CIOud Testbeds UNICO, collocated with IEEE CloudCom 2013* 2013