VIP ANR-09-COSI-03
Virtual Imaging Platform

D2.2.1: Robust and efficient job execution service

Rafael Ferreira da Silva, Tristan Glatard

Abstract

The VIP execution service enables the execution of simulation workflows on the European Grid Infrastructure, one of the largest distributed system in the world. The VIP execution service consists of the MOTEUR workflow engine and the GASW application wrapper, which was totally re-engineered and re-developed in VIP to provide a robust execution system. Basic robustness features are coupled to a self-healing method to provide autonomic task replication, which yields very good speed-up results. Rich monitoring and statistics are also available. Until now, the development and exploitation of the VIP execution service produced 3 papers in international conferences [1, 2, 3]. A journal paper about the self-healing method was also submitted. In addition, usage statistics demonstrate the maturity of the VIP execution service: since January 2011, 254 users have registered in the VIP portal, 6204 simulations were executed, 1 million tasks were processed by GASW and 379 normalized CPU years were consumed on EGI. It makes VIP the 2nd most active user in the biomed virtual organization of EGI. The VIP execution service can be accessed via the VIP portal at http://vip.creatis.insa-lyon.fr.

Contents

1 Introduction

The VIP execution service enables the execution of simulation workflow on distributed computing resources. In particular, resources of the European Grid Infrastructure[1] are targeted. With more than 270,000 cores distributed in 325 resource centers in 47 countries, this system is one of the largest distributed system in the world. VIP has access to the biomed Virtual Organization of EGI, which is supported by a bit more than 100 resource centers and has 330 registered users. This document describes the architecture, implementation, optimization and results provided by the VIP execution service.

The architecture of the VIP execution service is diagrammed on Fig. 1, where the sequence diagram shows the invocations triggered by the launching of a workflow from the VIP portal. The VIP portal generates workflow execution files from the form filled-in by the user in the graphical user interface. It then delegates execution to the MOTEUR workflow engine [4]. MOTEUR generates application invocations based on the dependencies among workflow activities, and iteration patterns on the input data. Starting from a description of the application, the Generic Application Service Wrapper (GASW) then generates the actual tasks that will be executed on the computing resources. These tasks

[1] http://www.egi.eu

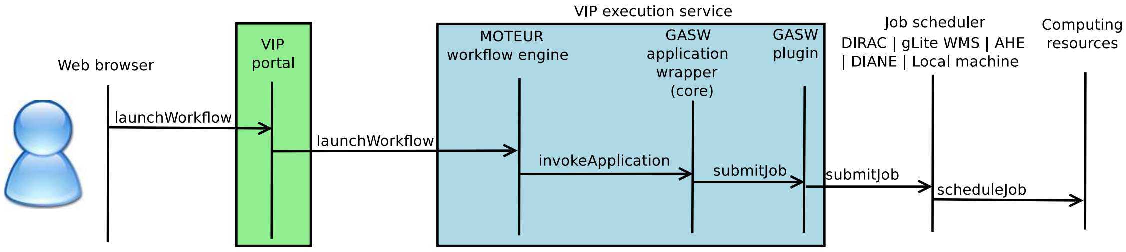


Figure 1: Architecture of the VIP execution service.

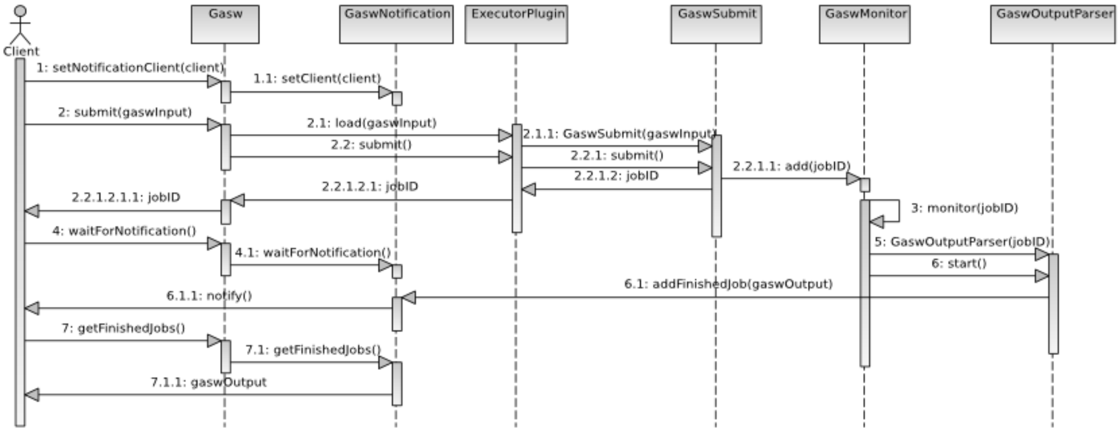


Figure 2: GASW execution sequence

are submitted to the local machine, or to one of DIRAC[2] [5], gLite WMS[3], Application Hosting Environment (AHE[4]) schedulers using the appropriate GASW plugin. Schedulers finally select a compute resources for task execution. This architecture was described in [1].

MOTEUR and the AHE GASW plugin were developed externally to the VIP project. In VIP, we focused on the design and development of GASW core, and its DIRAC, gLite WMS and local plugin executors. Section 2 details the software architecture, Section 3 describes its robust task replication system, and Section 4 shows monitoring features. Section 5 reports some usage statistics. The VIP portal will be described in deliverable D2.3.4.

2 Software architecture with pluggable backends

The GASW execution sequence is shown on Fig. 2. `GaswSubmit`, `GaswMonitor` and `GaswOutputParser` are part of the plugin while `Gasw`, `GaswNotification` and `ExecutorPlugin` belong to GASW core. A simple job submission in GASW involves the interaction of a client with 6 main objects. Client interactions are limited to job submission and notifications about job completeness. Once instantiated `Gasw` class, the first step is to set the notification class by using `setNotificationClient` method. Job submission is done through submit method passing as argument a `GaswInput` object. GASW will try to match the executor name defined in the input with the list of available executor plugins. Once selected an executor plugin, GASW generates the execution bash script and submits it to the corresponding execution system by using `GaswSubmit`. A job identification is generated and passed to the monitor thread (`GaswMonitor`) as well sent to the client as return of the submit invocation. Upon job completion, `GaswOutputParser` is invoked to retrieve job outputs files and to parse them filling the database with job execution information. Then, `GaswNotification` is notified about job completion and which is in charge to propagate the notification to the client. When the client receives the notification it requests the outputs that are delivered as `GaswOutput` objects.

[2] http://diracgrid.org

[3] http://glite.cern.ch/glite-WMS

[4] http://www.realitygrid.org/AHE

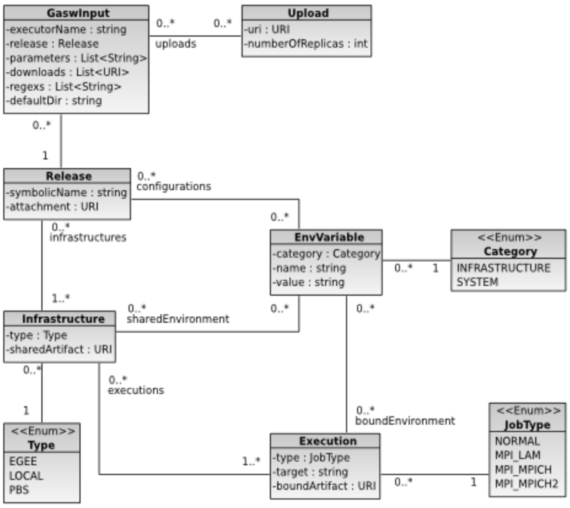


Figure 3: GASW input schema.

2.1 GASW core

The most important classes in GASW core are the GASW input, submitted at step 2 on Fig. 2, and the GASW output, returned at step 7.1.1 on Fig. 2. The GASW input schema is shown on Fig. 3. A single `GaswInput` object represents a job execution. It is composed by 7 attributes:

- `executorName`: defines the executor type that the job should be submitted. If the provided value is null or does not match any available executor, GASW will submit the job to its default executor.
- `parameters`: list of parameters associated to the application execution.
- `downloads`: list of input files to be downloaded in the execution node.
- `regexs`: list of regular expressions to match against outputs.
- `defaultDir`: default directory where to store files matching the regular expressions.
- `uploads`: list of output files to be uploaded to a Storage Element. List elements are represented by an Upload object that is composed by the URI where the file should be uploaded and the amount of replication that should be performed.
- `release`: application described as a `Release` object.

In GASW, applications are described as `Release` objects. A release contains a set of `Infrastructure` objects which represents different infrastructures. An infrastructure contains a set of Execution objects which represents specific application executors for different job types (e.g.: NORMAL, MPI_MPICH2, etc.).

In addition, environment variables (`EnvVariable` object) can be defined on release, infrastructure or execution level, and can be of INFRASTRUCTURE or SYSTEM type. Variables of type INFRASTRUCTURE will be set during job execution by using the export command. SYSTEM variables will be handled by GASW to set internal properties and are normally declared on release level. Variables declared on release level are shared among

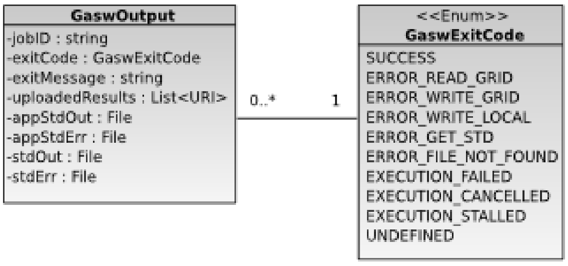


Figure 4: GASW output schema.

all infrastructure and execution modes. Analogously, variables declared on infrastructure level are shared among all execution modes. Environment variables precedence is handled by the most external (release level) to the most internal (execution level).

The GASW output schema is diagrammed on Fig. 4. Results of a job execution are provided as `GaswOutput` objects. The object is composed by the job identification, standard output and error files as well application output and error files, an exit message, a list of uploaded outputs and an exit code represented by `GaswExitCode` enumeration. Exit codes assume the following values:

- SUCCESS: application successfully executed
- ERROR_READ_GRID: error downloading file from grid using lcg-cp
- ERROR_WRITE_GRID: error writing file to grid using lcg-cr
- ERROR_WRITE_LOCAL: error creating execution directory
- ERROR_GET_STD: error downloading standard error and/or outputs files of the application from grid
- ERROR_FILE_NOT_FOUND: error matching result files
- EXECUTION_FAILED: execution failed
- EXECUTION_CANCELED: execution canceled
- EXECUTION_STALLED: execution stalled (for DIRAC execution)
- UNDEFINED: default exit code

2.2 GASW plugins

GASW currently supports 4 plugin executors: DIRAC, gLite WMS, local, and AHE. The AHE plugin was developed in an exemplar project of the Virtual Physiological Human[5].

The DIRAC plugin uses the DIRAC command-line client to submit and monitor jobs. To speed-up monitoring of simulations involving a lot of jobs, the status of several jobs is asked in a single client invocation. In addition, a specific service was developed in DIRAC so that minor statuses can be reported. Minor statuses are used while the job is executing, to have more insight about its execution phase. VIP is seen from DIRAC as a single

[5] http://vip.creatis.insa-lyon.fr:8080/VPH-EP-9

user, which can lead to fairness issues among VIP users. To ensure some fairness among workflows, jobs belonging to different workflows are submitted with different DIRAC job names and CPU times. Due to the performance of DIRAC, this plugin is the preferred plugin for execution in VIP. DIRAC doesn't properly support MPI jobs at the moment though.

The gLite plugin uses gLite command-line clients to submit and monitor jobs. To speed-up monitoring of simulations involving a lot of jobs, the status of several jobs is asked in a single client invocation. It supports all gLite requirements, including MPI jobs. This plugin is used in production in VIP to execute MPI jobs.

The local executor simply forks a UNIX process on the local machine. It is mainly used for testing at the moment.

2.3 Cluster agent

In VIP, DIRAC was extended to support non-grid clusters so that both EGI sites and local personal clusters are available to execute GASW jobs. A few design constraints were identified for this cluster extension. First, no intervention from the cluster administrator must be required, i.e. everything should happen in the user space. In particular, installing a gLite computing element is out of question. Second, security rules must be respected. In particular, clusters must be used with distinct personal user accounts (no sharing of generic user accounts) and login/passwords cannot be stored in the VIP or in DIRAC. Finally, the solution should rely on minimal technical assumptions on the cluster architecture, in particular concerning network connectivity and software installations.

The proposed solution extends DIRAC with a cluster agent launched by the user on his cluster(s) account(s). The user logs in using his/her usual authentication method, independently from the DIRAC system. Once set up, the cluster agent runs a daemon that contacts the DIRAC Workload Management Service periodically. If there are waiting tasks for the current user, it launches DIRAC pilots to the local cluster queue. Once the pilots are running, they behave exactly as regular pilots, but they can only retrieve tasks belonging to the user on behalf of whom they are running. As for any other computing site, the scheduling between local and remote cluster is done by DIRAC.

The cluster agent is released as a software bundle containing the user X.509 certificate, a grid data transfer client and DIRAC installation scripts. During the setup phase, the daemon is adjusted to the scheduler type (PBS, BQS and SLURM are supported). The user proxy is first created by the agent in the setup phase and then periodically renewed from a MyProxy server[6]. It is used for data transfers and to authenticate to the DIRAC server. A cron job is set to automatically restart the daemon in case the cluster is rebooted. The only required software dependency is a Python interpreter.

3 Robustness

3.1 Basic features

GASW automatically resubmit jobs up to a configurable number of times. Failure is detected from the job exit code. Although this ensures that site-specific problems are avoided, this can also (i) flood the infrastructure with lots of unsuccessful jobs in case of permanent errors, and (ii) be totally inefficient in case errors are only detected after a long

[6] http://grid.ncsa.illinois.edu/myproxy/

execution time (e.g. inefficient data transfer leading to timeout). To cope with this, three other basic mechanisms are available in GASW:

1. Job kill: users can kill job sets;
2. Job rescheduling: users can reschedule a job to another site;
3. Job replication: users can replicate a job. Replicas are canceled by GASW as soon as one successfully completes.

Triggering these operations manually is both time-consuming and inefficient. To improve that, we developed the autonomic task replication mechanism described in the next subsection.

3.2 Autonomic task replication

One strategy to automate the handling of incidents is to use a healing process that could autonomously detect and handle them. The self-healing process detailed in [3] handles incidents by automating such operations through a MAPE-K loop: monitoring, analysis, planning, execution and knowledge.

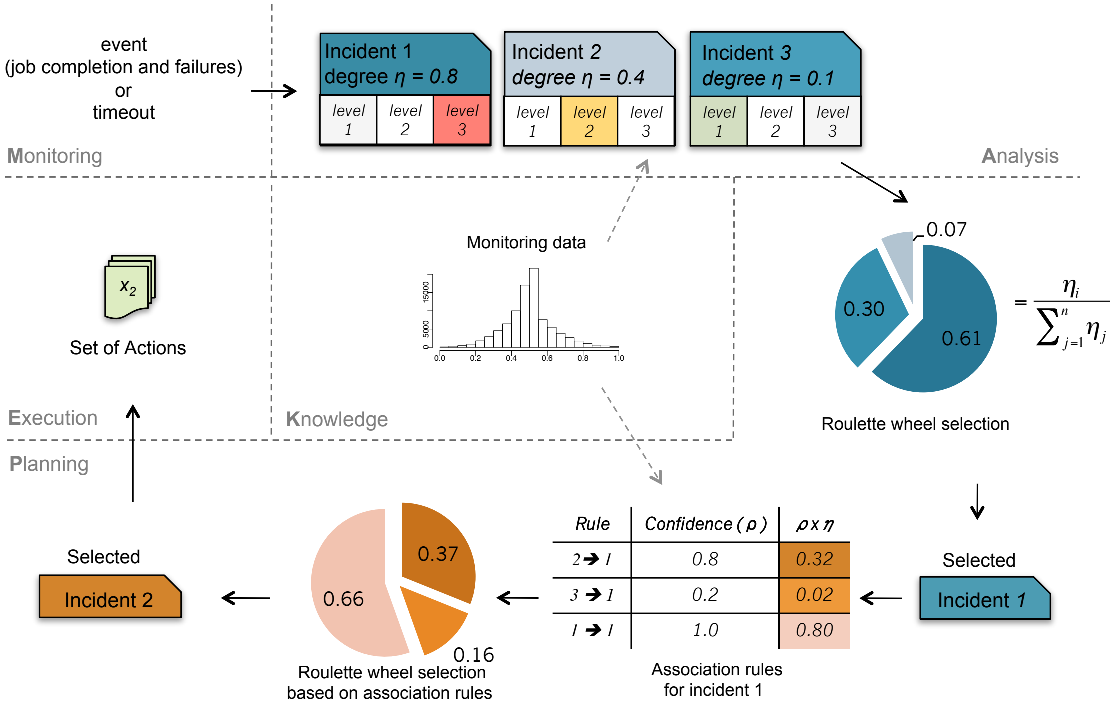
Figure 5 presents the MAPE-K loop of the healing process used in GASW. The monitoring phase consists in collecting events (job completion and failures) or timeouts. In the analysis phase, incident degrees and levels are determined from events and execution logs (historical information) extracted previous execution logs. Then, an incident is selected by using a combination of roulette wheel selection and association rules (planning phase). Roulette wheel selection assigns a proportion of the wheel to each incident level according to their probability and a random selection is performed based on a spin of the roulette wheel. Association rules are used to identify relations between levels of different incidents based on historical information. Finally, a set of actions is performed to handle the selected incident.

Task replication is the action taken to respond to incident blocked activity. We define the incident degree η_b of an activity from the max of the performance coefficients p_i of its n tasks, which relate the task phase durations (`setup`, `inputs download`, `application execution` and `outputs upload`) to their medians:

$$\eta_{\mathrm{b}} = 2.\max\left\{p_i = p(t_i,\tilde{t}) = \frac{t_i}{\tilde{t}+t_i}, i \in [1,n]\right\} - 1 \qquad (1)$$

where $t_i = t_{i\_setup} + t_{i\_input} + t_{i\_exec} + t_{i\_output}$ is the estimated duration of task i and $\tilde{t} = \tilde{t}_{setup} + \tilde{t}_{input} + \tilde{t}_{exec} + \tilde{t}_{output}$ is the sum of the median durations of tasks 1 to n. Note that $\max\{p_i, i \in [1,n]\} \in [0.5,1]$ so that $\eta_b \in [0,1]$. Moreover, $\lim_{t_i \to +\infty} p_i = 1$ and $\max\{p_i, i \in [1,n]\} = 0.5$ when all the tasks behave like the median.

The estimated duration t_i of a task is computed phase by phase, as follows: (i) for completed task phases, the actual consumed resource time is used; (ii) for ongoing task phases, the maximum value between the current consumed resource time and the median consumed time is taken; and (iii) for unstarted task phases, the time slot is filled by the median value. Figure 6 illustrates the estimation process of a task where the actual durations are used for the two first completed phases (42s for `setup` and 300s for `inputs download`), the `application execution` phase uses the maximum value between the current value of 20s and the median value of 400s, and the last phase (`outputs upload`) is filled by the median value of 15s, as it is not started yet.



Rule	Confidence (ρ)	ρ x η
2 → 1	0.8	0.32
3 → 1	0.2	0.02
1 → 1	1.0	0.80

Figure 5: MAPE-K loop of the healing process.

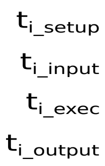


Figure 6: Task estimation based on median values.

Blocked activities are addressed by task replication. To limit resource waste, the replication process for a particular task is controlled by two mechanisms. First, a task is not replicated if a replica is already queued. Second, if replica j has better performance than replica r (i.e. $p(t_r, t_j) > \tau$, see equation 1) and j is in a more advanced phase than r, then replica r is aborted. Figure 7 presents the algorithm of the replication process. It is applied to all tasks with $p_i > \tau$, as defined on equation 1.

```
Input: Set of replicas $R$ of a task $i$

rep = true
for $r \in R$ do
  for $j \in R$, $j \neq r$ do
    if $p(t_r, t_j) > \tau$ and $j$ is a step further than $r$ then
      abort $r$
  done
  if $r$ is started and $p(t_r, \tilde{t}) \leq \tau$ then
    rep = false
  else if $r$ is queued then
    rep = false
done
if rep == true then
  replicate $r$
```

Figure 7: Replication process for one task.

Figure 8 shows the makespan obtained using our self-healing process on 5 repetitions of 2 VIP applications, `FIELD-II/pasa` and `Mean-Shift/hs3`. The makespan is considerably reduced in all repetitions of both activities. Speed-up values yielded by self-healing range from 2.6 to 4 for `FIELD-II/pasa` and from 1.3 to 2.6 for `Mean-Shift/hs3`. This self-healing mechanism is available as a GASW plugin[7].

4 Monitoring and statistics

GASW writes execution information in a database with the schema shown on Fig. 9. Information is kept about the execution node, timestamps of the job execution phases, consumed and produced data. To avoid scalability issues, a separated database is used for each workflow execution. As shown on Fig. 10, this information is used to provide monitoring information in the VIP client, for instance job flow, histogram of transfer times, map of execution, etc.

Such traces are very useful for research on distributed systems, to validate assumptions, to model computational activity, and to evaluate methods in simulation or in experimental conditions. For this purpose, we described and illustrated in [2] a science-gateway archive model, and we published these traces in the Grid Observatory[8].

5 Results: usage of the VIP execution service

The VIP execution service has been used in production since the beginning of 2011, to support the execution of medical simulations on the biomed virtual organization of EGI. The following usage statistics demonstrate its success.

[7] http://kingkong.grid.creatis.insa-lyon.fr:9002/projects/gasw-healing-plugin/wiki/Wiki

[8] http://grid-observatory.org

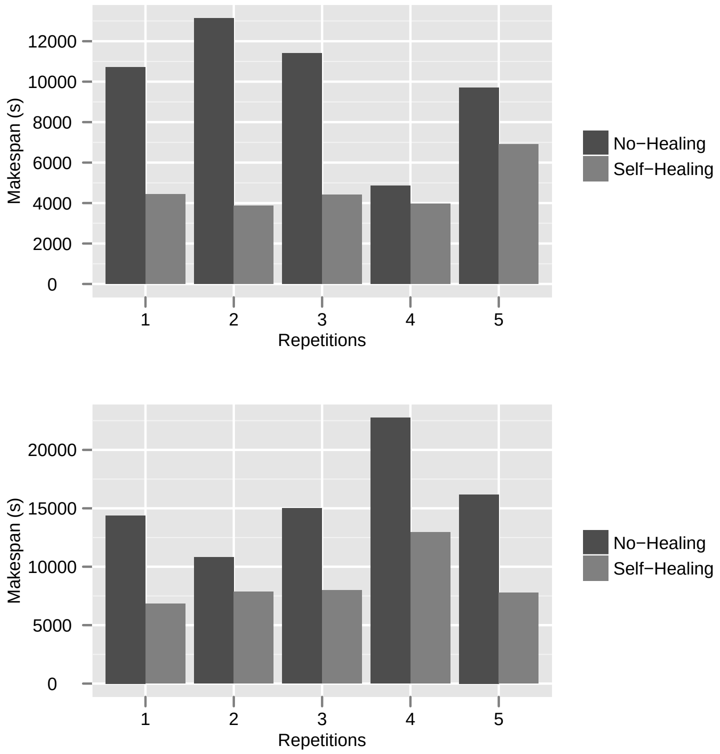


Figure 8: Execution makespan for `FIELD-II/pasa` (top) and `Mean-Shift/hs3` (bottom).

To date, 254 users are registered in the VIP portal. According to data collected by EGI[9], VIP's certificate is the most used of all EGI portals. According to the EGI accounting system[10], 379 normalized CPU years have been consumed by VIP from January 2011 to August 2012. It makes VIP the 2nd user of the whole biomed VO. Since January 2011, 6204 simulations have been launched on VIP, which represent about a million of tasks processed by GASW.

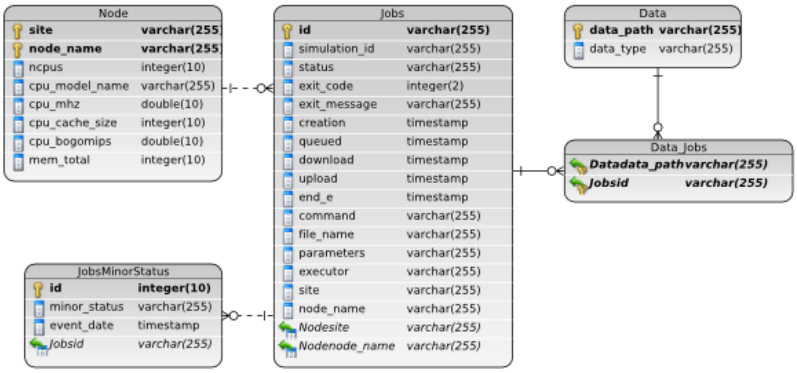
6 Conclusion

The VIP execution service consists of the MOTEUR workflow engine and the GASW application wrapper. The latter was totally re-engineered and re-developed in VIP to provide a robust execution system. More technical information about GASW, including development roadmap, download and usage instructions are available on the VIP software management tool[11]. Its software architecture is made of a core to which are plugged executor plugins. Four plugins are available to date, for DIRAC, gLite WMS, local execution and the Application Hosting Environment. The VIP execution service was also extended to support local cluster execution with DIRAC.

[9] https://wiki.egi.eu/wiki/EGI_robot_certificate_users

[10] http://accounting.egi.eu

[11] http://vip.creatis.insa-lyon.fr:9002/projects/gasw



Node	
site	varchar(255)
node_name	varchar(255)
ncpus	integer(10)
cpu_model_name	varchar(255)
cpu_mhz	double(10)
cpu_cache_size	integer(10)
cpu_bogomips	double(10)
mem_total	integer(10)

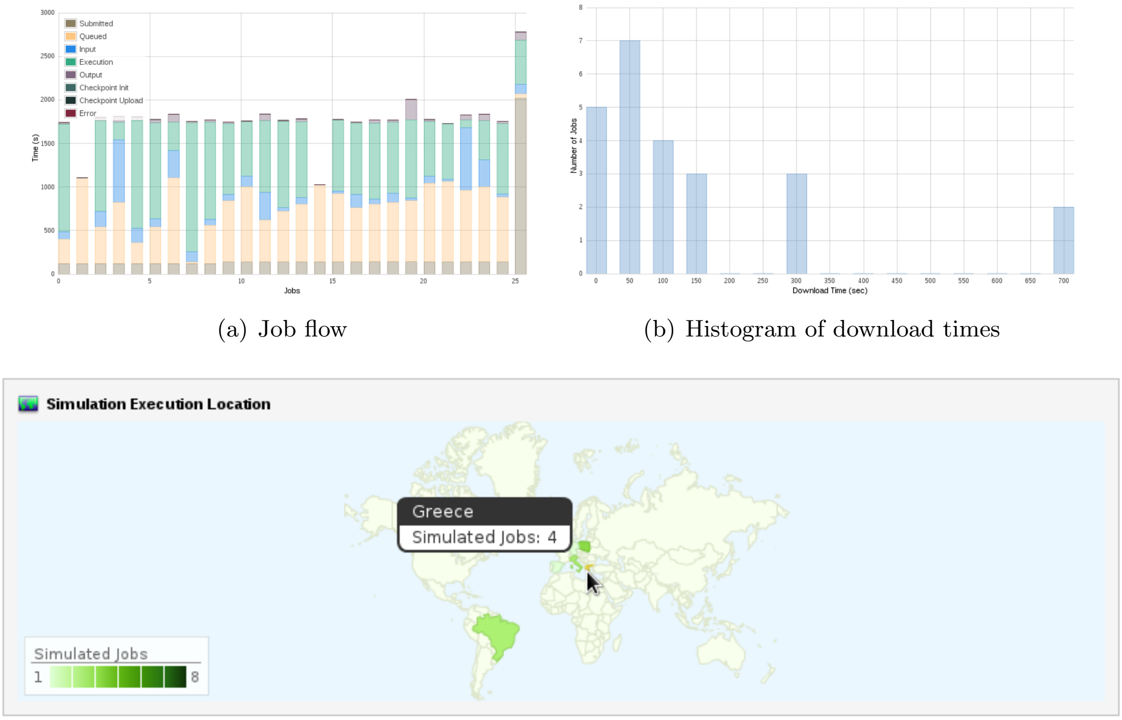
Jobs	
id	varchar(255)
simulation_id	varchar(255)
status	varchar(255)
exit_code	integer(2)
exit_message	varchar(255)
creation	timestamp
queued	timestamp
download	timestamp
upload	timestamp
end_e	timestamp
command	varchar(255)
file_name	varchar(255)
parameters	varchar(255)
executor	varchar(255)
site	varchar(255)
node_name	varchar(255)
Nodesite	varchar(255)
Nodenode_name	varchar(255)

Data	
data_path	varchar(255)
data_type	varchar(255)

Data_Jobs	
Datadata_path	varchar(255)
Jobsid	varchar(255)

JobsMinorStatus	
id	integer(10)
minor_status	varchar(255)
event_date	timestamp
Jobsid	varchar(255)

Figure 9: GASW database schema



(a) Job flow

(b) Histogram of download times

(c) Job execution location

Figure 10: Monitoring information extracted by the VIP client from the GASW database.

Basic robustness features consisting of task resubmission, kill, rescheduling and replication were added. Task replication was coupled to a self-healing method to provide autonomic replication and control the resource waste. Experiments show that it yields very good speed-up results.

GASW captures various kind of information about the execution, such as node properties, and timestamps. Rich monitoring and statistics can therefore be extracted by the VIP portal from the executions.

Until now, the development and exploitation of the VIP execution service produced 3 papers in international conferences [1, 3, 2]. A journal paper about the self-healing method was also submitted. Usage statistics demonstrate the maturity of the VIP execution service: since January 2011, 254 users have registered in the VIP portal, 6204 simulations were executed, 1 million tasks were processed by GASW and 379 normalized CPU years were consumed on EGI. It makes VIP the 2nd most active user in the biomed VO.

References

[1] R. Ferreira da Silva, S. Camarasu-Pop, Baptiste Grenier, Vanessa Hamar, David Manset, Johan Montagnat, Jérôme Revillard, Javier Rojas Balderrama, Andrei Tsaregorodtsev, and T. Glatard. Multi-Infrastructure Workflow Execution for Medical Simulation in the Virtual Imaging Platform. In *HealthGrid 2011*, Bristol, UK, 2011.

[2] R. Ferreira da Silva and T. Glatard. A science-gateway workload archive to study pilot jobs, user activity, bag of tasks, task sub-steps, and workflow executions. In *CoreGRID/ERCIM Workshop on Grids, Clouds and P2P Computing*, Rhodes, GR, 2012.

[3] R. Ferreira da Silva, T. Glatard, and Frédéric Desprez. Self-healing of operational workflow incidents on distributed computing infrastructures. In *IEEE/ACM CCGrid 2012*, pages 318–325, Ottawa, Canada, 2012.

[4] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and Efficient Workflow Deployment of Data-Intensive Applications on Grids with MOTEUR. *International Journal of High Performance Computing Applications (IJHPCA)*, 22(3):347–360, August 2008.

[5] A Tsaregorodtsev, N Brook, A Casajus Ramo, Ph Charpentier, J Closier, G Cowan, R Graciani Diaz, E Lanciotti, Z Mathe, R Nandakumar, S Paterson, V Romanovsky, R Santinelli, M Sapunov, A C Smith, M Seco Miguelez, and A Zhelezov. DIRAC3. The New Generation of the LHCb Grid Software. *Journal of Physics: Conference Series*, 219(6):062029, 2009.