# GWpilot: Enabling multi-level scheduling in distributed infrastructures with GridWay and pilot jobs

A. J. Rubio-Montero[a], E. Huedo[b], F. Castejón[a,c], R. Mayo-García[a]

[a]*Centro de Investigaciones Energéticas Medioambientales y Tecnológicas (CIEMAT), 28040 Madrid, Spain.*
[b]*Departamento de Arquitectura Computadores y Automática, Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain.*
[c]*Laboratorio Nacional de Fusión, Asociación EURATOM-CIEMAT, 28040 Madrid, Spain.*

**Abstract**

Current systems based on pilot jobs are not exploiting all the scheduling advantages that the technique offers, or they lack compatibility or adaptability. To overcome the limitations or drawbacks in existing approaches, this study presents a different general-purpose pilot system, GWpilot. This system provides individual users or institutions with a more easy-to-use, easy-to-install, scalable, extendable, flexible and adjustable framework to efficiently run legacy applications. The framework is based on the GridWay meta-scheduler and incorporates the powerful features of this system, such as standard interfaces, fair-share policies, ranking, migration, accounting and compatibility with diverse infrastructures. GWpilot goes beyond establishing simple network overlays to overcome the waiting times in remote queues or to improve the reliability in task production. It properly tackles the characterisation problem in current infrastructures, allowing users to arbitrarily incorporate customised monitoring of resources and their running applications into the system. This functionality allows the new framework to implement innovative scheduling algorithms that accomplish the computational needs of a wide range of calculations faster and more efficiently. The system can also be easily stacked under other software layers, such as self-schedulers. The advanced techniques included by default in the framework result in significant performance improvements even when very short tasks are scheduled.

*Keywords:* Grid scheduling, pilot jobs, resource characterization, resource management, application compatibility.

## 1. Introduction

The consolidation of distributed computing infrastructures (DCIs) is a long process involving continuous changes, standardisation processes, upgrades of middleware tools, implementation of new capabilities, dissemination of their use, porting of applications, etc. In any case, this effort has enabled the integration of a large pool of resources around the world, resulting in a high-throughput computing (HTC) platform that is ready for high loads whose best proponents are the current grid infrastructures. Despite recent advances, the work is not yet complete, particularly for job scheduling. Users, developers and site administrators continuously experience poor performance, complexity, and resource underutilisation.

Because of the dynamism and heterogeneity that are present in the majority of DCIs, especially those based on the grid [1], calculating the best match between computational tasks and resources in an effectively characterised infrastructure is, by definition, an NP-complete problem [2], and only sub-optimal schedules can be found for these environments. Nonetheless, the persistent problems on DCIs have avoided deployment of those complex algorithms in production, even though they have been well-tested on simulators and in controlled environments. Grid information systems (GIS) [3] (usually based on GLUE [4]), which are commonly misconfigured, do not provide an effective characterisation of the offered resources. Even more, the systems do not show any information about the basic bandwidth and latency, the shared policy of the remote queues, or the average waiting time. As a consequence, no valuable functionality devoted to advanced reservation of resources has been implemented. Therefore current middleware do not allow the knowledge of the type of resources that will be effectively assigned *a priori*. A lack of tools for generic user application profiling is an added difficulty. Another problem is the continuous overload of centralised services and resources, which increases the response lag and queue times, resulting in poor turnaround of individual jobs.

According to the performance aspect considered [5], the scheduling problem can be oriented to increase the throughput and the resource utilisation of the infrastructure, to improve the user share, the prioritisation and the data allocation, or to reduce the application makespan. Algorithms for these policies are widely utilised by local resource manager systems (LRMS) to exploit the clusters and constellations belonging to same institution. Because their computational environment experiences few changes, their schedules are based on static descriptions of resources. Grid schedulers (GSs) (also called meta-schedulers, super-schedulers, resource brokers (RBs), or workload management systems (WMSs) are cognisant of grid dynamism, and they constitute the first mechanism to reduce DCI complexity, thus providing automated and unattended access to all the resources.

Diverse GS implementations have been proposed, but generic GS can only perform very simple job scheduling if only

information from the GIS is used. Thus, a few GSs have survived, not because of their capacity for complex scheduling but because of some other aspects, such as their adaptation to the grid characteristics, scalability or ease of use. Currently, the GSs based on Condor [6] are the predominant ones in the largest grid infrastructures in its two versions: Condor-G [7] on Open Science Grid (OSG[1]) and gLite/UMD WMS [8] on the European Grid Infrastructure (EGI[2]). Among their features, they have demonstrated high stability and scalability and are able to support hundreds of thousands of jobs every month. Nonetheless, Condor has some known drawbacks, e.g., it is hard to install, maintain and customise, and it has some rigidity on the grid because it was initially designed as LRMS. Usually, these GSs are overloaded by users because they are deployed as central services. These issues have a larger effect on its gLite/UMD flavour, where the overhead imposed by several layers of middleware results in poor job turnaround.

GridWay [9] is a potential alternative; this system implements advanced scheduling policies, is easily customisable [10], and can be remotely accessed through standard interfaces [11, 12]. However, its deployment is oriented towards individual institutions or users and, with the exception of a few medium-sized infrastructure projects [13], has not been selected as part of the production manager systems (PMS) of large virtual organisations (VOs); consequently, its suitability for these communities has yet to be demonstrated.

In any case, neither Condor nor GridWay has yet addressed consistent methods to overcome the resource characterisation problem, despite the implementation of behaviour models [14, 15]. The consequences are especially evident when executing short jobs, in which the middleware overhead represents an important percentage of the time consumed. As a result, GSs are devoted to improving the throughput of long jobs from an organisational point of view.

To overcome the lack of reliable information provided from the GIS, many application-oriented frameworks have assumed some of its roles and other functionalities that initially correspond to the GS, such as statistical accounting or remote profiling, to improve the quality and quantity of the supplied resources. In this sense, GSs are commonly used as a tool for resource provisioning [16]. Moreover, because those frameworks select the resources, they are able to re-implement basic functionalities from the GS. Systems included in this category are some workflow managers [17], PMS [18, 19, 20, 21], and self-schedulers [22, 23, 24], which promise a better performance based on a deeper knowledge of the computational needs of a specific application. This is an expensive approach that increases the intrinsic difficulty in testing innovative algorithms in a real environment. Another issue is the specialisation of such specific types of applications. As a consequence, the performance gain over those applications provided by a general-purpose GS is usually measured in controlled environments, such as simulators, small laboratories, or infrastructures with only a few production nodes. Thus, although they show interesting results, their improvements are not usually extrapolated to real infrastructures or to different application types with the same good performance.

In opposition to the aforementioned early-binding methods, where the workload is scheduled to resources before they have been effectively assigned, the relatively recent pilot job technique (or late-binding model) is being introduced in DCIs to overcome the current limitations of middleware. This approach accomplishes computational user tasks in a more flexible, stable and reliable way while reducing overhead. Moreover it represents a powerful scheduling layer that can be combined with traditional mechanisms to achieve the required performance levels. In this sense, this new scheduling overlay is logically placed between the ones provided by the applications and the provisioning tools. Nevertheless, the current systems based on pilot jobs [19, 21, 25, 26, 27, 28, 29, 30] are not exploiting all the advantages that the technique offers, or they lack compatibility or adaptability.

In this study, a different approach based on pilot jobs is proposed. The new approach is more flexible and adjustable than the existing ones, it is adaptable to legacy applications and it is suitable to both user and VO levels. Implementation of this system represents a remarkable step forward in the use of DCIs because it goes beyond establishing simple network overlays to overcome the waiting times in remote queues or to improve the reliability in task production. It properly tackles the characterisation problem in current infrastructures, allowing users to arbitrarily incorporate customised monitoring of resources and their running applications into the system. Any user can easily take advantage of this feature to perform a specialised scheduling of his application workload. For this purpose the necessary tools to declare policies based on this monitoring are available without the need of modifying any code in the pilot system. Users can automatically guide the provisioning without the need of explicitly indicating one resource or manually submitting pilots. Furthermore, all of these features can also benefit skilled developers to build complex scheduling policies such as the ones in [5, 31]. Additionally, self-schedulers or workflow managers can be easily adapted to establish an upper scheduling layer that will take into account the real characteristics of resources, although they were federated. Therefore, the new approach presented in this work enables a multi-level scheduling model that users can actually manage, unlike the mechanisms offered by current pilot job systems.

The structure of this paper is as follows. Section 2 presents a general description of pilot systems that is useful for describing the strengths and weaknesses of the existing frameworks, the reason for their design and the compiling of motivations to develop a new system. In Section 3, the GWpilot architecture is introduced, as well as its implementation that makes its features possible. The simplified mechanisms to properly manage the different scheduling layers is explained in Section 4. The reasons for approaching a functional comparison with other pilot systems can be found in Section 5. Then, Section 6 shows two sets of experiments: one describes the obtained results of comparing GWpilot with these other frameworks on a real envi-

[1] http://www.opensciencegrid.org
[2] http://www.egi.eu

ronment; the latter offers a discussion on the suitability of GWpilot to perform a customised scheduling in a multi-application context. Finally, conclusions are presented in Section 7.

## 2. Pilot jobs

### *2.1. Overall vision and nomenclature*

The current necessity of a grid job to access different grid services anywhere outside the specific administrative domain of resources where it is being run implies that this job could require access to the Internet through NAT or proxy services. This can be done through a master-slave application where the slave, running inside a worker node (WN), can directly communicate with a coordinator server through its own communication protocol and bypass (some) grid middleware. The main motivations for establishing this network overlay within a DCI are:

- to reduce overhead, mainly the waiting time in remote queues at the LRMS and shared GSs using slot appropriation;
- to effectively characterise the assigned resources (the WNs) by sending their real properties and current status to the master;
- to enable compatibility with other legacy systems and applications by checking and creating special configurations; and
- to reduce grid complexity by directly using assigned resources and monitoring the user tasks.

Thus a pilot system is defined as a framework where many satellite programs (pilots) running inside grid jobs branch out to monitor a set of computational user tasks that are continuously assigned by a server following the master-slave scheme. The difference between user task and grid job must be introduced; in this study, a pilot job is a regular grid job, and each of the computational parts of the user application that are being run inside a pilot job are user tasks. In this sense, the series of procedures needed to set every pilot job up is called resource provisioning, as well as the mechanism to assign every task to pilots is named workload or task scheduling. On the other hand, user-level or application scheduling denotes the division of the calculation workload into tasks because is usually performed by user's tools. It is noteworthy to mention that some abstraction models of pilot jobs have been already proposed [14, 32, 33]. These abstractions and formalisms undoubtedly result adequate to describe scheduling algorithms based on pilots, but are succinct to detail the design of some frameworks. However, the presented scheduling roles (user-level, workload and provisioning) can be logically distinguished in these abstractions or in any pilot system, although they usually are mixed up.

There is a large collection of literature on distributed master-slave applications for scientific computation. In addition, the definition of a pilot job is commonly extended when the satellite program is being executed on resources belonging to other types of DCIs such as clouds or even on multiple standalone clusters. For this reason and because of the wide plethora of topics that have been covered, the systems that have been tested on, or are closely related to, standard grid environments, are described in this section. This collection is adequate to illustrate the method applied in this study. On the other hand, even though there is a wide range of approaches and possible implementations, any pilot system is conceptually composed of the following main components:

- One (or several) user task queue (UTQ). It is usually a (priority) queue that is accessed by means of a local command line interface (CLI), a web portal, a web service or an API.
- A master coordinator or pilot server (PiS). It is devoted to manage the running pilots, their requests and the task-pilot relationship.
- Pilot jobs (pilots) or agents. In addition to consecutively accomplishing user tasks and enabling remote monitoring of these tasks and the assigned resource, they perform other functions, such as checking and preparing the application environment (downloading and configuring software), executing multiple tasks (multi-task pilot jobs) or accepting tasks belonging to different users (multi-user pilot jobs, MUPJ).
- One or several pilot generators, suppliers (or *provisioners*), fabrics or factories (PiF). They perform the submission of pilots to a certain computing infrastructure (local clusters, the grid or the cloud). These generators can be manually executed by a user, but they are generally triggered by a daemon that locally monitors the UTQ for pending tasks.

Other additional components that can form part of a pilot system to facilitate the proper behaviour of the aforementioned modules or improve the performance are the following:

- Message accumulators (MAs) or customised proxies are often deployed to provide a type of bidirectional outbound communication between the PiS and pilots when firewalls or NAT are present.
- To improve the performance of the computation, the PiS can abandon the passive role and use the support of a scheduler module that implements advanced algorithms to effectively coordinate the pilot-task matchmaking. If only a simple fist come first serve (FCFS) or a fit resource first serve (FRFS) approach is considered for workload scheduling, this role is usually assumed by the PiS.

The implementation of this type of system involves multiple design decisions. The most important of these decisions concerns the method of communication between server and slaves, that is, whether the system adopts a pushing or a pulling behaviour. In the first case, the server can initiate the communication with the slave, e.g., to submit tasks, and in the second case, the slaves initiate all communications with the server. Additionally, the connections can be synchronous or asynchronous, state-less or state-full, and they suppose overheads that the

scheduling algorithms cannot overcome. Consequently, these decisions determine the choice of completely implementing the Internet application layer (i.e., directly sending information over sockets), using pure remote procedure call (RPC) libraries (RMI, CORBA), coding embedded methods into web pages (PHP, Python), delegating to external scripts (CGIs), or following a more standardised web services (WS) schema, where the specification of messages has to be selected (XML-RPC, SOAP, WSDL, REST, WSRF), independently from the common protocol (HTTP, SNMP, TCP) used. Furthermore, assuring the security in these communications is necessary because some standard grid middleware will be bypassed. In general, current pilot systems use SSL/TSL authentication and encryption methods based on the user proxy delegated in the WNs, but additional mechanisms may be necessary. Additionally, a level of isolation in task execution should be considered, especially when tasks are executed on behalf of other users [34].

Cost estimation for re-implementing any GS functionality must be performed in advance. The use of third-party tools should follow the same procedure. The design complexity and the dependence among components is another important factor that influences deployment and performance. Moreover, the types of interfaces offered to the users, programmers and managers of the system, such as CLI tools, APIs or web interfaces, must also be considered. Regarding all these aspects, a classification of existing design approaches with their corresponding examples is described in the following subsections.

### *2.2. GS/LRMS embedded pilot systems*

LRMS and GSs (to a lesser extent) have already incorporated advanced scheduling policies. Therefore, it would be a great advantage to include a pilot in their resource pools and schedule tasks over this pilot pool as usual. In this way, ranking and selection of the best resources to meet the task requirements are permitted. This approach was first introduced in grid environments in 2001 by the glidein technique [7], which was developed using the Condor framework. Glidein uses the *condor_startd* daemon, executed inside any computational node and then monitored and managed by Condor using *condor_sched*. This binary file can be wrapped in a script and submitted as a grid job to a remote resource; then, the daemon connects to the Condor *Collector* and it is enrolled in the Condor resource pool. This technique is not suitable to be used by itself in production grid infrastructures, and, for this reason, glidein was recently complemented with other developments [35], resulting in glideinWMS [26].

Currently, this framework is being used to provision resources to high-level systems, such as workflow managers, e.g., Pegasus [36] (through Corral [37]), or it is used directly through the Condor CLI by users allowed to access core services in OSG [38]. The main differences from a pure glidein system are an enhancement of the grid job wrapper for *condor_startd* and the inclusion of a specialised PiF compound of two principal modules: the *VO* (or *Corral*) *Frontend* and the *glidein Factory*. Multiple instances of *Frontends*, *Factories* and their related software can be deployed to provide load balancing or to access multiple VOs. Alternatively, the powerful classified advertisements (ClassAds) language is used for describing users, tasks, glideins and machines containing services, and, consequently, it enables the Condor matching mechanism by means of the evaluation and comparison of the attributes, expressions, preferences and constraints declared for each actor. However, ClassAds has perhaps derived in a tool that is too complex for conventional users. Additionally, to assure confidentiality, the system generates key pairs for each ClassAd, even for the user's data stored at web servers, increasing the complexity.

The obvious drawbacks of this system are the need for bidirectional outbound connectivity, the compatibility of the *condor_startd* binary, the difficulty of installation and the customisation (inherited from Condor and the newly implemented modules). The first drawback is overcome by using a general-purpose message accumulator (the *Generic Connection Brokering*) or the specifically designed *Condor Connection Brokering* to enable the communication between the glideins and the *condor_sched*, although this approach creates a delay penalty because compiled messages in an intermediate server cannot offer the same performance as LAN connectivity. The second drawback is usually masked by the similar configuration of the resources in the large infrastructures. However, the latter two issues remain important drawbacks to the general adoption of glideinWMS because installing a complete instance could be daunting and difficult for a specific application. Therefore, many users that launch their own applications cannot profit from the scheduling features provided by this system.

Other systems that include remote grid or cloud resources into LRMS or GS systems have been proposed. For example, MyCluster [39] builds personal clusters on demand by provisioning nodes from TeraGrid or Amazon EC2 and enrols them into the LRMS by using user-space remote file systems and virtual private networks over WANs. Other works [40] propose mechanisms for the elastic growth of grid infrastructure making use of cloud providers.

### *2.3. Pilot systems related to LHC VOs*

Condor-G is used directly in OSG and is the base of gLite/UMD WMS. All previous versions have been deployed on EGI and its preceding phases since the beginning of the last decade. The glidein technology seems the obvious choice to introduce pilot jobs into large infrastructures, especially for those devoted to grid computation from the Large Hadron Collider (LHC[3]). However, this idea was only evaluated in recent years. As a result, on the date of the debut of glideinWMS [26], three of the VO communities out of the four LHC experiments had already implemented their own pilot job techniques and had integrated them into their legacy and centralised production manager systems (PMS) for massive data production and processing. These three were DIRAC [19] for LHCb, AliEn [20] for ALICE, and PanDA [21] for ATLAS.

The uncertainty of achieving better performance results or getting some added value, compared to their own mature pro-

[3] http://public.web.cern.ch/public/en/LHC/LHC-en.html

duction systems, became a significant drawback for gLiteWMS. Since then, its adoption by LHC VOs has been irregular, also because of the influence of OSG on each specific collaboration (mainly on CMS and ATLAS). Notably, these PMS act not only as a pilot system but also as a complete tool that provides more services. They are frameworks specifically designed to compute the great amount of data generated by these experiments, and therefore, they are composed of additional software mainly devoted to managing data access, allocation and replication. The requirements of tasks processed are specific and they could only accept regular user tasks collaterally. Consequently, PanDa and AliEn frameworks only support high energy physics (HEP) experiments. The exception is DIRAC, that is used in other types of initiatives, such as the *biomed* VO [41].

The DIRAC WMS [19] was the first pilot system to be deployed in production on a large VO [42], and it continues to use the gLite/UMD WMS as the tool for submitting its pilots to the grid. The pilots (*pilot agents*) install DIRAC (and VO software) from repositories if it has not already been configured by VO managers. They can adopt a *general* or a *specific* role if configured to run either any task or to only accept tasks from a certain user, calculation or specific requirement. This association is possible because of a double-matching mechanism against the UTQ items based on the ClassAd language. Thus, DIRAC can request gLite/UMD WMS the information about resources that accomplish these requirements, filter them according to previous executions and finally submit the pilots. When these pilots contact with the server directly fetch the first waiting task in the corresponding UTQ. Therefore, the system is following a FRFS policy where the user fair-share is implemented to a small extent, although an attempt to include MOAB [43] is proposed for future versions. It is noteworthy to mention that DIRAC is deeply described in the Section 5, due to it is used as one of the pilot frameworks to be compared with GWpilot on a real grid environment.

A similar two-level scheduling concept [38] is proposed by the PanDA pilot framework [21], but based on Condor-G and datasets. Several PiFs (*AutoPilot* [44]) are installed with Condor-G when it is deployed at central services to adjust the number of pilots and tasks in such a way that computing elements (CEs) will not be overloaded. This schedule is based on the input dataset of tasks and their expected output. This is important because pilots have multitasking capabilities and can retain outputs from previous executions to be reutilised or uploaded at a later time.

In 2004, a completely different pilot approach [27] was implemented for the AliEn framework [20]. A daemon (*Computer Agent*) is installed in VO-Boxes at each grid site to continuously monitor the state of its corresponding CE. It also has a PiF role and submits pilots directly to the CE [45] when it detects that tasks have been assigned at UTQ to that resource. Central AliEn services schedule these tasks using the information provided by *Computer Agents*; therefore, it is the *Computer Agent*, and not the pilot, that is triggering the task matching using a general characterisation of the resource. That is, the framework implements more advanced scheduling policies without using a central GIS, avoiding a high percentage of monitoring communications, but at the cost of certainty of a proper WN and the ability to control the task behaviour. In this way, an approach based on site-allocated PiFs was also proposed in DIRAC [46], but the utilisation was limited to standalone clusters [47]. Similarly, some PanDA developers [48] considered the creation of a new site-allocated PiF that supplied glideins [49], but this solution did not seem to be feasible because it forced an unnecessary re-implementation of glideinWMS functionalities and the infrastructure was overloaded with two pilot systems [38].

With respect to the communication mechanism utilised in the aforementioned frameworks, all actors (user, CLI tools, pilots, core modules) in DIRAC, AliEn or PanDa must use their own PKI/GSI protocol based on XML-RPC/HTTP [50], WSRF/SOAP or simple HTTP requests, respectively. For this reason, in addition to the web portals offered by PMS, more common interfaces are provided to developers to facilitate integration with other systems or applications, although they are not necessarily standardised. Examples are the Ganga [51] compatibility that is usually offered or the REST interface recently added to DIRAC. With respect to the coding language, Python is predominantly used. Additionally third-party open source tools such as web servers are used.

Unlike the other LHC collaborations, CMS has not implemented their own pilot system and simply incorporated glideinWMS. With this system, great results were obtained [52]. However, the CMS community has two PMS [18], one for data analysis [53] and one for Monte Carlo production [54], that can submit tasks to Condor-G, gLite/UMD WMS, ARC [55] and some LRMS through a connector called BOSSLite [56]. Thus the use of pilot jobs has played a collateral role and has been bound to OSG sites [57] because of the policy of CMS collaboration. However, efforts are being made to incorporate glideinWMS, such that the whole CMS production is expected to pass through this technology very soon.

### *2.4. Application-oriented overlays*

HEP researchers involved in the four LHC experiments generally use the same software for processing their data; therefore, it is feasible to make the effort to customise a common framework for them. Nevertheless, there are many users that employ several types of applications not installed on remote sites. Thus, the development of a PMS only makes sense for middle-large collaborations. Moreover, the deployment of any type of centralised system for queuing and prioritising tasks may not be the best approach to improve the performance of an individual application because only its programmer knows its specific characteristics. For this reason, some approaches have been proposed for providing a developer-oriented interface for distributed environments, which tackles the implementation of master-worker applications. Initially they focused on offering an API that simplifies the basic operations of the slave (monitoring, forking tasks, getting results, etc.). Two similar approaches were proposed: AMWAT [58] for AppLeS [59] scheduler and M/W [60] for Condor. The former is a collection of C/C++/Fortran functions and the latter implements a set of C++ template classes that the programmer must customise

Table 1: Comparison of some distinguishing features of main pilot systems. This compilation is focused on their compatibility and their capacity for interoperation in grid computing; their tools to adapt applications and their management by users; their default scheduling capacities; and how users can influence the scheduling.

| | **GS/LRMS embedded** | | **PMS** | | | **Application-oriented** | |
|---|---|---|---|---|---|---|---|
| | glideinWMS | GWpilot | DIRAC | AliEn | PanDA | DIANE | BigJob |
| **Deploy-ability:** | | | | | | | |
| Pilot requirements | Compiled binary (*condor_startd*) | Python >= 2.4.3 | Specific Python release and multiple binary dependences | Executable locally stored in shared directory (VO-boxes) or proxy-cache (CVMFS) | Python 2 (only if generic pilot is deployed, but specific versions and middleware were need) | omniORB libraries | Python 2.7 |
| Connectivity (Pull/push) | TCP sockets (Push. MAs are need: GCB/CCB) | HTTP/S (Pull) | DISET, i.e. XML-RPC/HTTP (Pull) | WSRF/SOAP (Pull) | HTTP/S (Pull) | CORBA call-backs (Pull) | Redis (Push. *BJ-Manager* uses the *Coordination Service* as MA) |
| Grid interoperation for provisioning[(a)] | Integrated with Condor-G: direct submission to GRAM2/4/5, CREAM and ARC resources (mainly based in middleware libraries) | Integrated with GridWay: direct submission to GRAM2/4/5, CREAM, OGSA-BES and ARC resources (mainly based in middleware libraries) | Submission delegated to WMS. Additionally, allows direct submission to statically defined GRAM2 and CREAM resources (based on middleware commands) | Site-allocated PiFs perform local submissions to GRAM2, CREAM, or ARC resources (based on middleware commands) | Submission delegated to Condor-G | Relies on Ganga: submission delegated to WMS, PanDA and DIRAC or directly to GRAM2, CREAM (based on middleware commands). SAGA backend (deprecated) | Relies on SAGA-python and SSH: submission delegated to Condor-G (potentially can be extended or use other SAGA implementations to handle grid interfaces) |
| **Usability / Interactivity:** | | | | | | | |
| Local CLI | LRMS-like | LRMS-like | - | - | - | Limited | Limited |
| Remote CLI | - | SAM based (through OGSA-BES) | WMS-like (extended) | UNIX-like prompt | Few parameterised commands | - | - |
| Task description (for CLI) | Proprietary | JSDL, proprietary | JDL | JDL | - | - | - |
| Task management API | Proprietary (Perl, SOAP). DRMAA (C/C++) | DRMAA (Java, C/C++, Perl, Python), OGSA-BES | Ganga. REST. Proprietary (Python) | Proprietary (C/C++/Perl) | Ganga, REST | Proprietary (Python) | Proprietary (Python) |
| **Scheduling:** | | | | | | | |
| Default workload policies | Condor matchmaking | GridWay adaptive scheduling | Classification of tasks in UTQs, subsequently FRFS | Tasks in a priority UTQ are assigned to sites if their local monitoring shows free slots and they fulfil constraints | Classification of tasks based on their datasets. Subsequently FRFS | FCFS | Developer must explicitly indicate the pilot to execute tasks |
| User defined policies | Free definition of constraints or ranking | Free definition of constraints or ranking | Few types of constraints | Constraints related to software installed | Constraints related to software, input files or sites | Free definition of constraints. It allows dynamically setting scheduling options | - |
| Provisioning policies | PiF dynamically evaluates policies defined in tasks and static options to generate pilot descriptions. Subsequently, Condor performs their scheduling and submission | PiF dynamically evaluates policies defined in tasks and static options to generate pilot descriptions. Subsequently, GridWay performs their scheduling and submission | For every UTQ, PiF select resources from a list provided statically or dynamically by WMS. Subsequently resources are filtered by previous pilot profiling and static options | PiF discretionally submits pilots if tasks are assigned to its hosting site at central UTQ | Several sites are selected to send the inputs of tasks with similar datasets. Then, PiFs discretionally submits pilots to these sites using Condor-G | Only a type of resource can be selected. Scheduling is generally delegated. If WMS is used, PiF can also perform a limited scheduling based on previous executions | Developer must explicitly indicate the resource to execute pilots (if it is not delegated to Condor-G) |
| Pilot characterisation [(b)] | ClassAds (Not customisable) | Unstructured key-value pairs (Customisable) | ClassAds (Not customisable) | ClassAds (Not customisable) | Key-data structure pairs (Not customisable) | Key-data structure pairs (Limited[(b)]) | Key-data structure pairs (Not customisable) |

(*a*) SSH or cloud interfaces are not considered as grid interfaces. Accessing to LRMS is only considered if the motivation is to bind with grid interfaces, as in Condor-G.
(*b*) Characterisation by users without modifying pilot code. Special case is DIANE, since *Worker Agent* could be considered part of the user application, due to CORBA model.

for his application. Both APIs simply wrap the available message passing protocols (MPI, PVM, or Condor-PVM) or the file sharing (Condor-Local I/O, Globus I/O or Nexus) mechanisms to enable bidirectional communications in a distributed environment.

Due to the high latency in wide area networks, these protocols are not suitable for grid environments. For this reason, other approaches based on GridRPC [61] were attempted [62], but all of these were lacking adoption when standard APIs for HTC programming were proposed (DRMAA [63], SAGA [64]), which did not consider a role for the programmer to manage the slaves. Related to this, an initiative called BigJob [28] has been recently presented for adapting pilot jobs to SAGA, but characterisation or scheduling tools are not included. Developers must explicitly indicate the pilot job to execute a task, and usually indicate the site (its URI) to run the pilot too. Thus, the *BJ-Manager* assumes the UTQ, the PiS and (partially) the PiF roles to facilitate developers the management of pilots (the *BJ-Agents*). The objective is to maintain the freedom to implement any provisioning policy, but without the necessity of directly implement SAGA code. However, due to the volume of resources belonging to current grid infrastructures, provisioning is usually delegated to Condor-G. Another component of BigJob is the *Coordination Service* which acts as a message accumulator between the *Manager* and the *Agents* based on the Redis[4] database and protocol.

In contrast, DIANE [25] is an integrated framework with a UTQ and simple scheduler (*Task Scheduler*), PiF (*Agent Factory*) and PiS (*Run Master*) that is written as an importable API library for Python applications. The communication between pilot jobs (*worker agents*) and the PiS is done by means of the CORBA free implementation omniORB[5]. However, it maintains a pull behaviour, where the exchange of information is done via call-backs. The *Agent Factory* can use simple heuristics [65] to fit resources by evaluating its error completion rate. Nevertheless, the resource discovering is based on the statistical results obtained from the pilot submission to the Ganga [51] library to a set of GSs, as is deeply explained in [65]. The *Task Scheduler* implements a FCFS policy by default, although its code can be modified to incorporate other procedures. Additionally, the *Agent Factory* and the *Task Scheduler* can work together to exchange information about the characterisation of resources and tasks. Therefore, specialised developers can extend DIANE to allow ranking policies and improve the resource provisioning with statistical data. The *adaptive workload balancing* (AWLB) proposal [33, 66] is a good example.

The primary advantage with DIANE or BigJob is their library design, which provide an easy-to-use and standalone installation. In the case of DIANE there is an added benefit because developers only have to divide the computational workload into tasks and to format them into a set of simple abstract classes. Then, DIANE will autonomously manage the necessary pilots and, through its own *File Transfer service*, will make the stage-in and -out process. However, both APIs are not standardised, and they even differ from the Ganga interface or SAGA standard, respectively. Moreover, the programmer could be forced to modify the DIANE or the BigJob code if the existing one did not meet his needs, as DIANE developers did in [67]. The performance data and characteristics of resources can only be compiled for the current execution of the application; therefore, this information can be shared neither with other users nor with other applications of the same user that are being executed at the same time.

To better understanding the concepts described in this section, Table 1 summarises the distinguishing features of main pilot systems in relation to their deploy-ability, usability and scheduling. The comparison is specially focused on the mechanisms offered to users to adapt their legacy applications to these frameworks, as well as to perform some type of customised monitoring and scheduling. Additionally to this table, it is important to mention that DIRAC and DIANE frameworks are deeper explained in Section 5 because they are used to be compared with GWpilot.

### *2.5. Other frameworks*

In general, desktop grid approaches are similar to standalone pilot systems, but they have been logically designed for a volunteer computing environment. Therefore, they were focused on exploiting idle CPU cycles from a large number of personal computers variably configured and allocated around the world. This fact implies a great effort in the development of compatible and secure *Clients*, which can run on heterogeneous platforms, and also centralised services able to coordinate hundreds of thousands of *Clients* while supporting high latency rates in their communications. In the case of BOINC [68], significant segments of the *Clients* and the core server are implemented in C++, in particular the communication that relies on XML/HTTP(S) requests from the *Clients*. The rest of BOINC, e.g., the mechanism for downloading files, is based on PHP web pages. Another example is XtremWeb [69], where *Clients* connect through RMI or XML-RPC to *Coordinators*. The servers are published at an *Advertisement Service*, also coded as an RPC. This enables a type of P2P load-balancing mechanism because *Clients* can obtain inputs from and store outputs to different *Coordinators*.

There have been efforts to adapt XtremWeb and BOINC architectures to a grid infrastructure, such as EDGeS [29] or GridBot [30], but legacy grid applications are difficult to port to these frameworks. Other initiatives have tried to solve this issue by implementing systems from scratch that were immersed in the current grid middleware and its interfaces. This is the case of Falkon [70], which was implemented as the Globus Toolkit 4 (GT4), that is, a WS-oriented system to be deployed on existent GT4 infrastructures. According to its initial design, a PiF (*Provisioner*) directly manages the execution of pilots (*Executors*) running on WNs, assuming *a priori* that GT4 supports brokered WS notifications, i.e., the PiS (*Dispatcher*) must push WS notifications directly to *Executors* running on a remote resource; however, this feature has not yet been implemented in GT4.

[4] http://redis.io

[5] http://omniorb.sourceforge.net

### 2.6. Lessons learned

In the previous subsections, the primary pilot systems available to date have been described. They provide a broad set of functionalities and good results in their different environments. Nevertheless, every one of them is missing some important capability that limits their feasibility or performance when other types of users or calculations are considered. These capabilities, such as a general compatibility and adaptability to the diverse existent grid software, the ease of use and deployment, or the customisation for any requirement, have been implemented together in this study, while several performance aspects are also taken into account. In this sense, the main design requirements can be summarised in the following subsections.

#### 2.6.1. Minimal functionalities

First, a general-purpose pilot system should offer a friendly interface for users, developers and administrators. For this reason, a new pilot system should provide standardised APIs and LRMS-like CLIs that fulfil the needs of every role, and facilitate the incorporation of external systems, such as upper scheduling layers.

Security, based on grid standards, is a must. Then, other implementations, different from the ones provided by the middleware, must follow these specifications and general infrastructure rules. New services or protocols should not be implemented if they are correctly provided by the middleware, i.e., the system should avoid duplication. For example, this includes the mechanism to stage files [25, 26], which is already available using GridFTP, SRM, etc.

The platform and library dependencies of remote pilot software [19, 25, 26, 29, 30], i.e. the pilot jobs, decrease the compatibility with heterogeneous resources. Therefore, it is better to use a widely extended interpreted language [21, 28] so those future modifications will be easier to implement as the system remains widely compatible. In addition, the allocation of modules at remote sites (such as site-allocated PiFs [27, 46, 47, 48]) should be discarded because it implies external collaboration for their installations, which is not suitable for standalone solutions. With respect to the local installation of the pilot system, it is desirable that it were easy enough to be carried out even by inexpert users [25, 28]. This feature will extend the suitability of the system for individual calculations.

The use of the pull mechanism and common protocols is the most suitable approach for pilot communications. It facilitates passage through site firewalls and proxies. Conventional RPC protocols and bidirectional implementations are not recommended. The use of message accumulators also increases the complexity and lag times in communications [26, 28, 32]. However, the choice of a pull method can influence the scheduling algorithms selected, as will be explained later. In any case, it is necessary to reduce and maintain controlled the overheads introduced by the system because they could constraint any scheduling decision.

To reduce the overheads, the middleware layers must be limited to the minimum. It is counterproductive to mix different pilot systems (as in [28, 48, 49]) or to complicate the design in excess (as in [19, 26]). A high number of standalone modules in the system does not only imply more difficulties in their deployment, but it also multiplies the quantity and size of communications among these modules, increasing the overhead (see Section 6). Well-defined WS message protocols are precisely proposed to easily integrate software from different providers and to improve the future extensibility of codes by other developers [27, 50], in exchange for a high increase of the message size. However, if the system complexity is low and the performance in communications is a priority requirement, their use could be avoided [21, 30]. Although these simplifications were made, the different pilot calls always introduce their own overhead, which has to be measured and controlled because it has an especial impact on task turnaround.

#### 2.6.2. Advanced scheduling and hierarchy

In general, pilot systems should distinguish the three-level hierarchy already established for scheduling in Subsection 2.1: the user-level, the workload and the provisioning layers. To fully support a personalised scheduling at user-level, the new pilot framework must provide the latter two layers with mechanisms that permeate the user's requirements. Thus, these requirements must influence on the whole scheduling because applications properly run on a type of resource that should be provisioned. However, it is difficult to guide provisioning if the pilot execution is delegated to an external GS [19, 25, 44]. Thus, to build a framework that manages tasks and provisioning together [26, 28, 32] is more feasible.

To really solve the characterisation problem of Grid Computing, the system must allow the free definition of constraints for every task, among a customisable set of characteristics for every pilot. For this purpose, the language used can not be complex as in [7]. Additionally, the customisation and characterisation of resources is a typical role of the provisioning phase [19, 21, 27], but will imply the modification of the pilot system code for every calculation type. Therefore, the new pilot system should also provide the mechanisms to properly characterise resources inside tasks. Moreover, applications that perform reactive scheduling need interfaces to dynamically know the current task, pilot and resource assignation and characterisation.

To satisfy the needs of institutions and VOs, the system should be multi-user and multi-application. Thus, another desirable functionality is the ability to implement fair-share and prioritisation policies [26] at workload and provisioning levels from the start. It becomes a major drawback if they are need post-development actions [43]. It is also useful to have accounting generically provided to the scheduler to improve the behaviour of current and future applications in the system. Therefore, a unique application-oriented approach (such as [25, 28]) is not suitable for designing the new framework.

In this sense, it is necessary to include advanced algorithms into these layers to delegate as much as possible the implementation of scheduling capabilities from user-level applications to pilot system. Therefore, the new pilot system must be able to implement more advanced scheduling algorithms than FRFS for workload scheduling. However, this requirement deserves

a fuller explanation because is different to the approach implemented for current pilot systems that make use of a pull mechanism. This is so because in a pure pull scenario, the simple FRFS task-pilot matchmaking policy has clear benefits. The technique has a small dispatch time (below 1 second [25, 47]); it maximises the usage of assigned resources (CPU occupancy); only one call type against PiS should be implemented, and it removes the necessity of implementing a scheduler module. Nevertheless, the disadvantages are also obvious. Although UTQ could prioritise tasks, the user fair-share is not really performed [43]. Additionally, applications can experience large, variable completion times because any task can uncontrollably fall into slow nodes. Then, this policy supposes a large drawback from the user point of view and from the use policy of certain VOs. Nevertheless, any algorithm different from FRFS has a variable time penalty according to its computational complexity, arising from evaluating a large quantity of possible pilot-task matches at every time. This could also have a negative influence on the task dispatching time (when it wastes time on the order of minutes), especially for short-duration tasks, and must be evaluated.

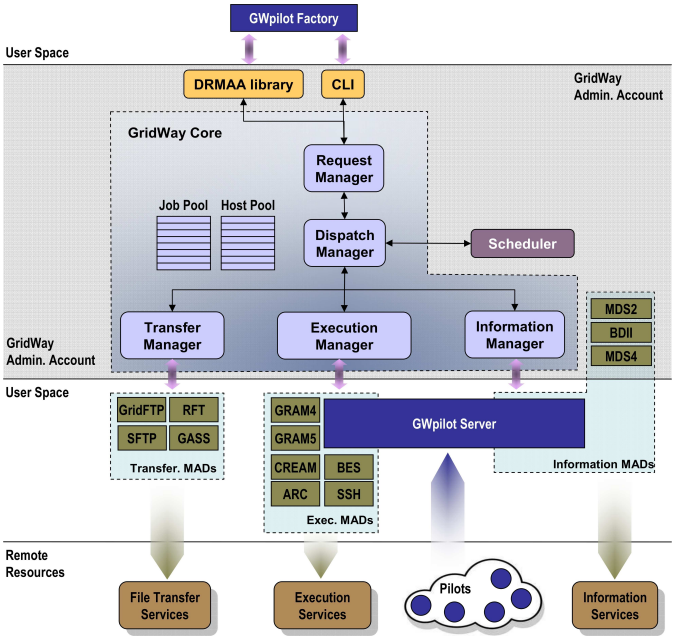



Figure 1: GWpilot components in GridWay architecture.

## 3. The GWpilot system

GWpilot is designed to accomplish the previously enumerated requirements and to profit from the numerous GridWay[6] advantages [71], while maintaining a simple design and implementing new functionalities that result in measurable and valuable improvements in several performance aspects of computing scientific applications. GWpilot acts as a GS-embedded pilot system, where pilots are included in the GridWay Host Pool as any other resource; thus, user tasks will be included in the GridWay Job Pool and subsequently scheduled among pilots as if they were common grid jobs. Accordingly, the GridWay Scheduler module is used to perform the task-pilot and pilot-resource matchmaking. Therefore, the user fair-share and prioritisation capabilities can be incorporated into the pilot system if a careful design is constructed. Similarly, the advanced scheduling capabilities of GridWay are improved by the proper characterisation and the online monitoring of resources and tasks that pilots provide, particularly the task migration and checkpointing functionality and the matchmaking decisions based on statistical data obtained from previous executions, per user and per resource. Additionally, the incorporation of pilots makes new features possible, such as reservation, data-allocation awareness and caching. The user can easily implement specific scheduling policies if the necessary tools for declaring customised characteristics of tasks and of the pilot environments are available. In consequence, the new framework will allow users, developers and administrators to build every scheduling level (application, workload and provisioning) accordingly to their specific needs in a unified and standardised way. The main design and implementation of GWpilot that make all these features possible are described in this section, while the improved management of scheduling are deeply explained in Section 4.

### 3.1. Architecture overview

The GridWay Job Pool conceptually corresponds to the UTQ in the framework. Because of it, any user or developer can use the available interfaces from GridWay to manage tasks. As mentioned above, another benefit is to make use of GridWay Scheduler to incorporate its algorithms to workload scheduling. The inclusion of every pilot in the GridWay Host Pool is the mechanism that allows this feature, which would have not been possible if the friendly interfaces had not been available or the language to descript resources had been difficult. In this sense, every resource (either pilot or the remote site) is described by means of unstructured label-value pairs. Users can easily inspect these values through CLI and set constraints into their descriptions of tasks. Subsequently, Scheduler calculates the best matches between the elements of two pools. In addition, the selection of resources for every pilot is visible for the users, who can influence on provisioning. The advantages of this unified vision of the workload scheduling and provisioning will be described later.

To preserve the performance and deploy-ability levels of GridWay, GWpilot must maintain its modular architecture [9]. Thus, the new elements of the system must be implemented like any other pluggable driver in the framework. For this reason, the system contains two main components in addition to the pilots: the GWpilot Server (GW PiS), which is implemented as a middleware access driver (MAD), and the GWpilot Factory (GW PiF), which acts as a common application at user-level and can be started by GW PiS. Subsequently, no communication will be performed between these components, and the system will run in a standalone way for submitting pilots wrapped

[6] http://www.gridway.org

inside grid jobs as needed. To perform this action, GW PiF uses other MADs (GRAM 2/4/5, ARC, CREAM, OGSA-BES, SSH), to obtain resources from multiple DCIs.

The code used for both components, PiS and PiF, is Python 2.4.3 compatible and, as any other MAD or user application within the framework, its installation is straightforward over a previously configured GridWay server. It only requires the copy of few source files to a basic GridWay installation and the deployment as Python module of a (third-party) generic open source HTTPS server framework[7]. GW PiS options are fully configured by editing the same line in the configuration file. These parameters are then propagated to GW PiF and, consequently, to the submitted pilots, so no more actions must be performed by inexpert users. Additionally, administrators can modify default parameters to tune general aspects of scheduling. On the other hand, MADs are generally executed in the user-space, which offers the following benefits: they can be independently instantiated by multiple users and they can directly use their grid proxy certificates for encoding communications. As GW PiF is executed by PiS, it profits from the same advantages.

Pilots communicate with the PiS through simple HTTP requests. A pilot will periodically pull PiS for a description of the task to run. This includes environment variables, the GridWay wrapper and the task options, as they were any remote GridWay job. Pilot translates these items and performs the necessary file staging to execute the wrapper. Subsequently, the wrapper performs the execution of the task as usual, assuring backward compatibility. Therefore, the stage-in and stage-out mechanism is established by the user as usual, i.e., by means of defining the source and destination of every file with local names or URIs through any standard grid protocol supported on the remote resource, e.g., GridFTP or SRM. On the other hand, HTTP requests are used to periodically advertise PiS about pilot characteristics and the statuses of tasks. In addition, tasks can communicate with pilots to arbitrarily include customised tags in its characteristics. As these items will be notified to PiS, and subsequently will be included in the Host Pool, the mechanism enables the personalised characterisation of the pilot.

The implementation of pilots is fully compatible with Python 2.4.3 and its standard modules and can run, for example, on any Red Hat 5 minimal installation. Additionally, the code requires less than 1,000 lines and only uses approximately 40 KB and can be manually executed. Thus, GWpilot is potentially suitable for deployment on any currently distributed platform type such as the cloud or desktop computer and for establishing network overlays among them, at least running in its insecure mode (i.e. without using grid certificates).

The design allows GWpilot to incorporate the following features that accomplish the requirements outlined in Subsection 2.6:

- Friendly user, administrator and developer interfaces from GridWay, such as the CLI, the submission mechanism based on templates, and the DRMAA and OGSA-BES standard interfaces. The latter also allows the use of remote commands through external implementations [72]. GridWay also provides DRMAA bindings to different languages such as Java, C/C++, Python and Perl.
- The security in communications and the file staging mechanisms, based on grid standards.
- The capacity to extend the pilot overlay to multiple DCIs, such as the grid, the cloud or even local resources, because PiF can use the current and future GridWay plug-ins (MADs) for provisioning and pilots allow manual execution
- An easy and standalone deployment on a unique server, independent of other middleware instances in these DCIs. Currently, GridWay is available through .deb and .rpm packages and their local dependences (of grid middleware) are managed by official repositories (from Linux distributions and IGE[8]).
- Pilot communications based on minimal HTTP pull requests. Overheads controlled by reducing the number and size of messages among system modules.
- Management of workload scheduling and provisioning capabilities in a box. Accounting from both layers (users, tasks, pilots and resources) is available.
- Mechanisms to properly characterise resources at user-level, based on a simple description language compatible with the friendly interfaces provided.
- The possibility of performing a personalised scheduling by every user, because independent PiS and PiF can be instantiated for all of them. Additionally users can run manually PiFs or even substitute them by new developed ones to customise provisioning.
- Management of multiple users and applications with fair-share and prioritisation policies. Potentially, PiS and PiF can be shared among users to share their managed pilot jobs and therefore, to carry on scientific production of a specific project or VO.
- Supporting the workload and provisioning layers with a default set of scheduling capabilities inherited from GridWay but powered by the pilot characterisation.

However, as mentioned in Subsections 2.6 and 3.1, overheads introduced constraints about what scheduling algorithms can be actually implemented on the pilot system. In particular, it is desirable that these overheads were predictable and even configurable. Consequently, beyond this overall description of GWpilot, a more complete explanation of its components is outlined below, paying special attention to the performance issues, but also to implementation details that make their advanced features possible.

[7] http://www.cherrypy.org

[8] http://www.ige-project.eu

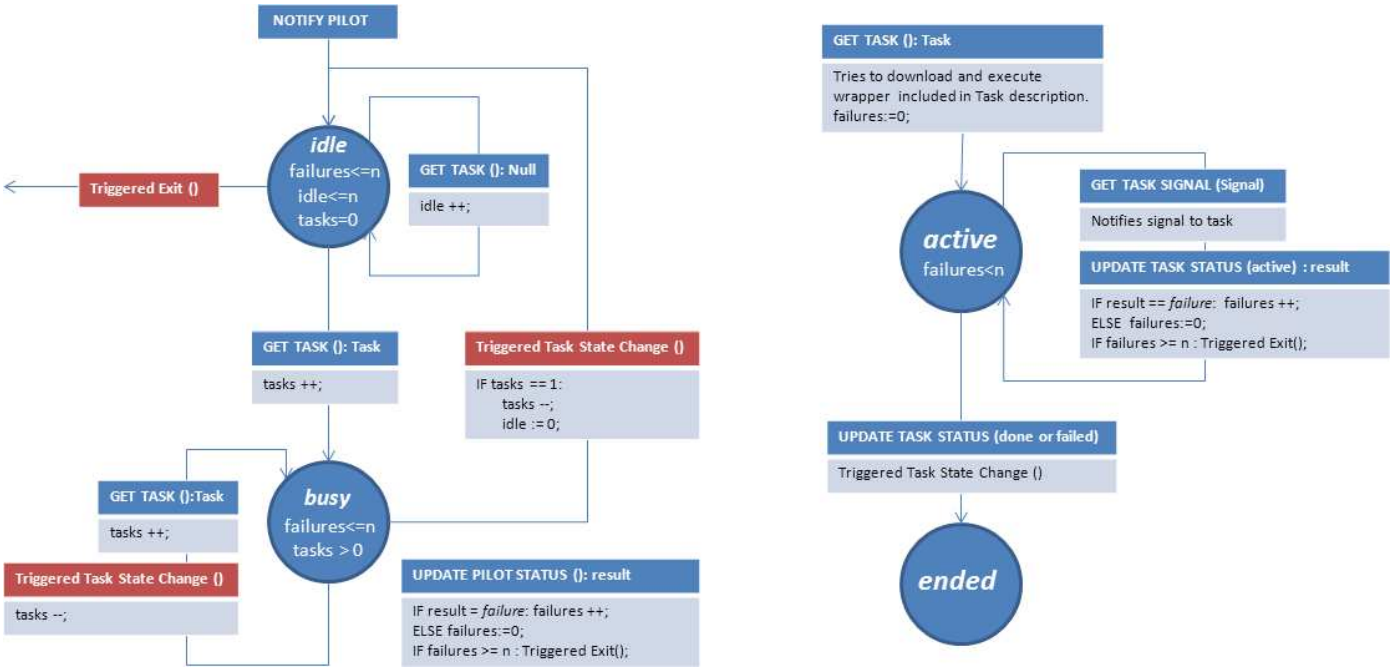



Figure 2: State machine diagrams representing pilot internal behaviour (left) and task management (right).

### *3.2. Pilots*

Communication with the PiS is a pull mechanism via simple HTTP requests from pilots. Due to the tag-based language used in GWpilot all the information interchanged is specific enough to be represented as a set of unstructured label-value pairs, although their significance is crucial for some of them, as will be explained below. Additionally, few request types are necessary. Therefore, formatting the information in XML and using complex message protocols is unnecessary. Instead, the tags are set into variables of GET methods or received as plain text in their responses. Despite the similarity with the remote execution mechanisms and to differentiate from pure RPC protocols, the pilot requests are referred to as pilot operations throughout the remainder of this paper.

The internal states are simple: a pilot can be *idle* or *busy*, and a task can be *active* or *ended* (*done* or *failed*). There are no pending tasks inside a pilot because whenever one is fetched from GW PiS, the pilot immediately tries to run it. As a consequence, there are only five operation types that the pilots request to the GW PiS:

- *Notify Pilot*: pilot advertises itself to PiS by sending its identification code and static characteristics.
- *Update Pilot Status*: when pilot considers itself enrolled to a specific PiS, it periodically sends its dynamic tags to that PiS.
- *Get Task*: when the pilot is in *idle* state, it asks PiS for a new task.
- *Update Task Status*: it is used to notify PiS of an *active* state when a task is going to be executed or the final state if the task has *ended*.
- *Get Task Signal*: it asks PiS for a POSIX signal to be passed to an *active* task. It is periodically performed when a task is being executed.

The last four operations are immediately triggered after a state change, thus reducing the turnaround. When no state change occurs, these operations are looped in a time interval with a limited number of retries, after which the pilot ends, either because PiS are not accessible or, in general cases, because there are no more tasks that can be assigned to the pilot. To better illustrate these state transitions two state machine diagrams extended with pseudo code are depicted in the Figure 2.

Usually, parameters such as PiS port number, deactivation of SSL security, pulling frequency against the server, and number of retries are propagated by GWpilot configuration to pilots. However, pilots also allow manual execution, and they can be launched by customised factories relying on other GS or LRMS. Furthermore, these parameters do not only enable the communication with the server. Retries determinate the provisioning policy, i.e. how often pilots are discarded and interchanged by new ones. On the other hand, pulling frequency is responsible of an important overhead component in task turnaround. Thanks to the simple design of pilot, this overhead is maintained roughly constant for every task, as will be demonstrated experimentally. The implications of both aspects will be explained in the last Subsection 6.4.

Secured communications are allowed using the delegated user proxy stored in the WN as a certificate for encrypting calls. Although SSL authentication is enough to distinguish the pilot owner, the pilot must identify itself with a unique code whenever an operation is performed against the server. This identifier is formed by the WN hostname, the site name, the user name and a hash number, which is created when a pilot starts run-

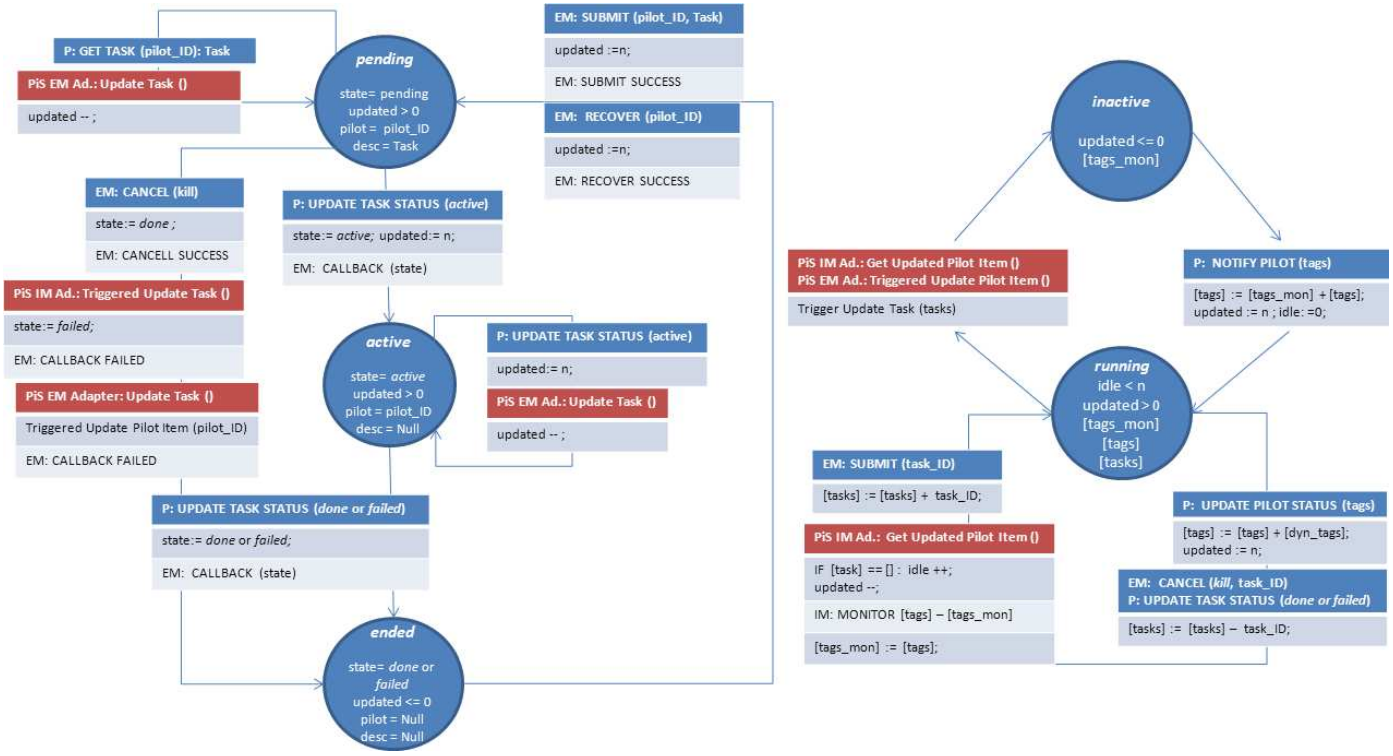



Figure 3: State machine diagrams representing the management of tasks (left) and pilots (right) by the GWpilot server.

ning and is maintained until its end. This method is performed to allow the proper consistency check of requested operations and to reject failing pilots. The exact identification of the remote and assigned resource will allow GWpilot to individually account performance statistics from each pilot. Such identification could be reutilised later if any other pilot is allocated in the same resource (this mechanism and its advantages will be explained in next subsection and Section 4, respectively).

Identification code is sent to PiS as any other property. Other tags have also special significance. *LRMS_NAME* = "*jobmanager-pilot*" is the type identifier that will differentiate pilots from other resources in the GridWay Host Pool; thus, it is always statically published when a pilot is enrolled in the system. On the other hand, queue tags dinamically shows the number of available slots to accomplish tasks by the pilot. Scheduler will discard those pilots without free slots published in the Host Pool.

To assure the backward compatibility with previous GridWay developments, its remote execution wrapper is utilised in pilots. The execution of user task is actually performed by the wrapper in the specific directory where the wrapper is downloaded. Pilot receives the location (URI) of wrapper, the final locations of their output streams (its logs), and the necessary environment variables. These outputs are retrieved by the system, enabling the possibility of fine troubleshooting. The pilot also gets the complete description of the task because it is a parameter of the wrapper. Thus, the pilot is able to translate the description of every task to cache its executable, input, output and restart (for checkpointing) files. These files are retained according to their size and their local path. Subsequently, their MD5 are published. Additionally, pilot creates a named pipe on which the user task will can write pilot the customised tags. Subsequently, the pilot tries to run the wrapper and monitor it to trigger the status changes and to fill some specific tags about the execution (see Table 2). The wrapper also enables the capacity of checkpointing the task execution. The possibilities that those features offer are explained in Section 4.

### *3.3. The GWpilot Server (GW PiS)*

The GW PiS is the most important module inside the GWpilot system, and its responsibilities go beyond simply putting GridWay in contact with pilot jobs. In this sense, PiS performs an active role by checking, filtering and caching operations from GridWay Core and pilots to improve the scalability and performance of the whole system. For this purpose, PiS must communicate with two GridWay managers: the Execution Manager (EM), that performs generic operations for tasks; and, the Information Manager (IM), that includes the characteristics of pilots in the GridWay Host Pool. This determinates the PiS internal design (see Fig. 4), which is mainly composed of an EM and an IM adapter, the Task and Pilot Pool lists, and the implementation of necessary procedures embedded in a (third-party) generic open source HTTPS server framework[7] for Python.

The system will usually initialise one PiS on behalf of each user in order to use his certificate and to encrypt his communications. Therefore, the system can support several PiSs listening to different TCP ports and their belonging pilots are private. However, a PiS can be shared among users if their IM and EM operations pass through the same stream (the implications of this configuration is commented in Subsection 4.4).

The pilot and GridWay Core operations against PiS modules have associated internal procedures that imply changes in the task or pilot states and, consequently, trigger more operations to other modules and again, doing it outside PiS. Main procedures used by adapters and the information workflow are shown in Fig 4, while the state machine diagrams that depict the internal behaviour of the server are depicted in Fig. 3. In addition, there are other mandatory functions devoted to checking the validity of the task, the pilot or the match performed in an operation. It is relatively common that pilots perform invalid operations against the PiS HTTP server due to network overloads or cuts.

In general, GridWay Core uses its manager modules (EM and IM) for sending common operations to the MADs, and then it waits for asynchronous responses. To reduce the turnaround overhead, all of them immediately respond, although some processes related to *SUBMIT* (sends a task description to certain pilot), *RECOVER* (claims for a task after a system restart) and *CANCEL* (removing a task execution request from PiS and killing it if it is being executed on a pilot) operations are performed in the background. However, GridWay Core also fills its overload with un-requested responses, so PiS uses this feature not only to later inform of submission and cancellation results but, more importantly, to improve the performance. Thus, the majority of responses for *POLL* (returns task state), *DISCOVER* (returns name identifications of active pilots) and *MONITOR* (returns pilot status) operations are originated by the PiS module adapters. It is not possible to perform the overloading technique for the other MAD operations. These responses are not arbitrarily retrieved by GridWay managers because the EM adapter immediately notifies only the task state modifications with a *CALLBACK* message, and the IM adapter caches the tag updates from *running* pilots in order to only periodically report their information changes.

Task states are slightly different from the aforementioned states in the pilot implementation because they must provide more information. Now, a task introduced in the Task Pool can also adopt a *pending* state, which will not change if this task is not fetched by any pilot and it updates its status. However, the description of the task (i.e., the wrapper, streams, grid paths, as environmental variables and parameters) provided through the SUBMIT operation is only maintained in the Pool until a successful *Update Task Status* is performed by a pilot.

However, the significance of pilot states are completly different for PiS. Pilots contained in their Pilot Pool can be *inactive* or *running*. These states stand for a pilot that is discarded by the system or accepted for processing tasks, respectively. This is so because the determination of a *busy* state in a pilot is of interest to the Scheduler, not to the PiS, which only stores the task-pilot matches from the former. The PiS only needs to know if the pilot has no tasks assigned or does not successfully update its status for a certain period of time, so it should be discarded. This deadline corresponds to the multiplication of the pulling frequency by the number of retries configured in pilots. The IM adapter is in charge of automatically changing the pilot state to *inactive* and updating the virtual queue tags to indicate Scheduler that no slots are available for this pilot.

The information related to the host, site and user name from

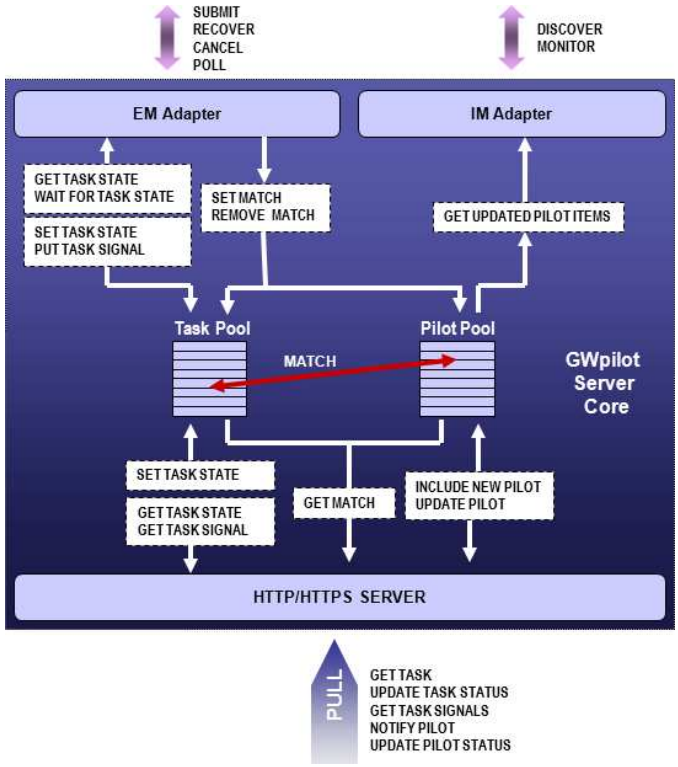



Figure 4: The GWpilot PiS internal modules, procedures, information workflow and its relation to external operations.

a discarded pilot is maintained in the Pilot Pool to be compared if another pilot is allocated in the same WN (or type in the same site) and tries to be enrolled in GWpilot. This correspondence is advised by the comparison with the identification code provided by the new pilot. However, the correspondent identifier name passed to GridWay Core will always be the same, enabling system to gather performance statistics from each pilot individually.

All these processes are extensively described in this subsection because they could become an important bottleneck that usually is not analysed in other systems. In general, users perceive this overhead as an increase in the dispatching time of every task. In the case of GWpilot, the simplified design of the PiS and the streaming mechanism to communicate with GridWay Core implies that this overhead will be manageable, as will be demonstrated with the experiments of Section 6.

### *3.4. The GWpilot Factory (GW PiF)*

The Factory is an efficient DRMAA-enabled program that follows a producer-consumer model to generate and replace pilots in a continuous flow. To improve its performance makes use of the CLI to check the statuses of pilots. In general, a GW PiF dynamically calculates the number of necessary pilots in the system by checking if the *LRMS_NAME = "jobmanager-pilot"* sentence is set in the requirements of the tasks belonging to its particular user. Moreover, any specific characteristic notified by pilots (or even published by GIS), can also be taken into consideration for submitting pilots. This feature improves the

Table 2: Some characteristics notified by pilots to GW PiS and subsequetly published into GridWay Host Pool.

| Characterisation | Tag name | Description |
|---|---|---|
| Identification | *PILOT_HASH_NAME* | Pilot identification code. |
| | *LRMS_NAME* | = "*jobmanager-pilot*" |
| Generic | *ARCH*, *OS_NAME*, *OS_VERSION*, *CPU_MODEL*, *CPU_MHZ*, *SIZE_MEM_MB*, *SIZE_DISK_MB*, *CPU_FREE*, *FREE_MEM_MB*, *FREE_DISK_MB* | Real hardware of the assigned node and generic monitoring. |
| GLUE-style and middleware tags | *DEFAULT_SE*, *SW_DIR*, *SCRATCH_DIR*, *DFLT_SE_FREE*, *SCRATCH_FREE* | *close-SE* and scratch directory configured. Additionally, pilot shows the vailable MB in these storages. |
| Virtual queue | *QUEUE_NODECOUNT*, *QUEUE_FREENODECOUNT*, *QUEUE_ACCESS*, *QUEUE_MAXTIME*, *QUEUE_MAXCPUTIME* | Number of virtual slots, available slots, accessing restrictions (VO, user distinguished name), and emaining wall and CPU time |
| Task | *PILOT_TASK_MEM_MB*,*PILOT_TASK_MEM_USED*, *PILOT_TASK_CPU_USED* | MB resident and the memory and CPU (%) usage by the task. |
| Basic network profile | *PILOT_LAG*, *PILOT_XFER_BW*, *PILOT_XFER_SE_BW* | Average lag in pilot operations and periodical bandwidth test against PiS and *close-SE*. |
| Caching | *CKPT_FILE_TASK*, *LAST_EXECUTABLE*,*LAST_INPUT_FILES*, *LAST_OUTPUT_FILES* | Name and MD5 of the checkpointing, staged and produced files in last execution. |
| User defined | *PILOT_*${*GW_USER*}*_VAR_* < *number* > | Key-Value written on pilot pipe. |

scheduling capabilities of GWpilot in a heterogeneous environment due to it is able to distinguish the type of pilots (architecture, middleware or software installed...) running at particular sites and constraint the submission of new ones to these sites. Additionally, users do not need to worry about provisioning because GW PiF will take account of task requirements to select suitable resources. This advanced feature and the configuration of PiF are deeply explained through the next section.

## 4. Multilevel scheduling

The features listed in Subsection 3.1 not only accomplishes a set of requirements that other systems do not. The main benefit of the tools presented is that they can be combined to build a broad range of scheduling algorithms. In particular, the main added advantage of GWpilot over other pilot systems is the capacity offered to users and developers to dynamically build their own scheduling policies, but assisted by the framework. With GWpilot, the scheduling is actually simplified into the three-level hierarchy logically established in Subsection 2.1: the user-level layer, where the applications dynamically modify their workload division and set specific requirements for every task; the workload layer, where the pilot system perform the task-pilot matchmaking following the user-defined requirements; and, the provisioning layer, where the system search in the grid for resources that fit those requirements. Thus, one main objective of GWpilot is to facilitate the exercise of dynamically propagating down the requirements of any application from the user-level to the lower layers. On the other hand, communities demand to improve specific aspects such as the global throughput or the fair-share. For this purpose, the generic configuration of workload and provisioning layers is supported. Therefore, the proposed abstraction is adequate to simplify the scheduling from the user's, developer's and administrator's points of views.

### *4.1. Dynamic personalisation of scheduling at user-level*

The level of scheduling customisation of any pilot system depends on the language used to characterise resources and tasks running, as well as on how the users can establish preferences or constraints for the execution of their tasks based on this language.

Every member of Host Pool is described by means of unstructured label-value pairs. With the exception of the identifier of every member, they can operate as un-typed variables, i.e. they can dynamically take a numerical value or turn into an arbitrary string. Additionally, the number of tags is not limited. Therefore, the characterisation of any resource from any DCI can be fully stored. GWpilot also performs this action with the information supplied by enrolled pilots (see Table 2).

In this sense, the framework actually gives the possibility to users to directly customise some tags. The mechanism does not imply the modification of pilot implementation by the user. Tasks running can communicate with pilot through a named pipe, the path of which is unique for each task and it is stored in an environment variable (${*PILOT_PIPE*}). Subsequently, tasks can write on pipe the customised tags declared as *PILOT_*${*GW_USER*}*_VAR_* < *number* >=< *arbitrary value* >. These tags will be published in the Host Pool and are maintained among task executions. Thus, the user application only has to store the needed information (an integer, float or string) into these variables in order to specify that any other intermediate file was stored in a scratch directory, any specific profile has been performed or any configuration has been done at the remote resource.

Any user can select multiple tags from resource descriptions to formulate a Boolean expression as requirement, which can be combined with a numerical expression to rank every candidate host (those pilots for which the requirement expression is true). In addition, user can include the supplementary tags dynamically customised. Scheduler fully supports the resolution of these expressions. Thus, those pilots with higher ranks are first used to execute his tasks (and to submit his pilots, as will be explained in Subsection 4.3).

Therefore, as a result of the integration of GWpilot framework, users and developers count on tools to really build a personal scheduling. Particularly, they can:

- dynamically inspect and filter those tags through *gwhost* command;
- notify any custom characteristic from their running tasks inside pilots;
- dynamically establish requirements and ranking expression based on these tags in their task descriptions to fit their computational needs.

However, the GWpilot also offers some other useful tools that allow users to dynamically know:

- what tasks are running in certain pilots (through *gwps* command) and how many tasks were successfully executed or failed in these pilots, as well as their accumulated transfer and execution times (through the *gwacct* command);
- what pilots have been submitted to real sites and which are really running in them (also with *gwps*). Same accounting is performed with pilots, so user can know the same accumulated registers than the ones provided for tasks (with *gwacct*).

These tools facilitate the tuning of GWpilot configuration for a specific calculation. They also allow users to select pilots that are effectively accomplishing tasks faster than others. Additionally, wrapper and middleware logs are stored in a simple tree structure based on the task and pilot numerical identifier, which facilitates the troubleshooting.

### *4.2. Feasible workload scheduling*

Through last subsection the simple tag-based language and the tools to personalise requirements has been introduced. However, the reader only get a glimpse of the possibilities those mechanisms offer.

Fist, the meaning and classification of the tags to be notified to the PiS and, consequently, published into Host Pool deserve a more complete explanation because they enable more advanced scheduling mechanisms than those that are associated with a typical characterisation of the resource. In this sense, in addition to the generic static and dynamic characteristics of a resource, such as the CPU type or the CPU consumption, pilots can specifically notify specific tags. For example, the real memory and CPU usage of the running tasks allow the Scheduler to properly detect performance losses and consequently to initiate the checkpoint of task and subsequently its migration.

Moreover, GLUE-style tags can be notified not only to describe the configured middleware (such as the *close-SE* available) but also to describe a virtual queue description that could be customised to provide the Scheduler with a number of fictitious or virtual free slots and the distinguished names of allowed users. By doing so, not only the multi-task and the multi-user (MUPJ) capabilities are enabled, but also resources composed by multiple cores or GPUs can be managed by Scheduler. Additionally, the remaining wall time in the remote resource (i.e. how many time remains for the end of pilot) is included in this virtual queue. Thus, Scheduler can fill pilots with tasks of adequate duration.

Furthermore, some files related to a task, i.e., executables, inputs, and outputs, are declared with their MD5 code to be cached for later reutilisation by other tasks. Any tag declared is directly visible by either the user or the developer, who could include it as a requirement to build complex workflows based on file dependences, which will be fully supported by Scheduler

The pilot implementation goes beyond offering the user the default tags contained in Table 2, which are feasible for a broad range of calculation types, but they could not be used to accomplish some concrete problems. The proposed mechanism to customise the characterisation of pilots not only facilitates the software deployment to VO administrators or even temporarily to unprivileged users. This characterisation methodology also allows the inclusion of advanced scheduling algorithms, especially those usually provided by third-party self-scheduler layers, which can now be added to the system without re-implementing GS functionalities. The provided possibility of direct profiling and monitoring on the WN (properly estimating outbound bandwidth or the application performance, for example) is essential for improved and reactive task chunking. Additionally, advertising the availability of some files facilitates the data-allocation policies. Moreover, an effective virtual type of advanced resource reservation, based on pre-emption, could be allowed in long-term GWpilot systems if this mechanism of file awareness is combined with the checkpointing features provided. This can be very useful for supporting some complex scheduling algorithms [31], MPI and very long executions, or enabling on-demand prioritisation of some calculations over others, such as real-time applications.

### *4.3. User-guided provisioning*

The capacity of GW PiF to inspect the requirement and rank expressions of tasks enables such type of user-guided provisioning. These preferences are dynamically taken in consideration when a pilot is submitted to progressively improve the quality of appropriated resources. The difference with approaches as [66], is that this mechanism is directly managed and customised by any user without requiring the modification of the pilot system code. In addition, it is more generic than other approaches [47] because does not constraint the submission of one pilot to accomplish a concrete task. Moreover, the guidance in provisioning is completely refined because PiF is able to distinguish among the customised characteristics of pilots to properly select resources.

The requirement expression of every pilot is simply built with the logical disjunction of the requirements of every task. However, the following formula builds the rank expression for user-guided pilots:

$RANK = \sum p_i \cdot rk_i \cdot n_i/n$

where $n$ is the total amount of tasks in the system, $n_i$ is the number of identical tasks with the same ($rk_i$) rank expression and $p_i$ is their current priority given by Scheduler. Thus, GW PiF maintains the fair-share among running applications. Moreover, as the PiF can be configured to deal with several users, it allows such a type of user fair-share in provisioning. This mechanism

Table 3: Static options for GWpilot configuration

| | |
|---|---|
| **Scheduler loop options:** | |
| *SCHEDULING_INTERVAL* | Interval to perform a new scheduling of pending tasks and pilots |
| *DISPATCH_CHUNK* | Maximum number of tasks and pilots dispatched in every scheduling interval |
| *MAX_RUNNING_RESOURCE* | Maximum number of pilots concurrently submitted to same site (or task to same pilot) |
| **Dispatch priority of a pilot or task (*j*):** $P_j = \sum_i w_i \cdot p_{ij}$**, where** $w$ **is the weight and** $p$ **the priority contribution of every** $i = \{FP, SH, WT, DL\}$ | |
| *FP_USER*,*FP_GROUP* | Fixed priority per user or group (default 0) |
| *SH_USER*,*SH_GROUP* | Ratio of submissions of a user or group over the rest (default 1) |
| *SH_WINDOW_SIZE* | Timeframe over which user submissions are evaluated (in days) |
| *SH_WINDOW_DEPTH* | Numer of frames (present frames are most relevant) |
| *DL_HALF* | When pilot or task should get half of the maximum priority assigned by this policy (in days) |
| **Suitable priority of a resource (*h*):** $P_h = f \cdot \sum_i w_i \cdot p_{ih}$**, where** $w$ **is the weight and** $p$ **the priority contribution of every** $i = \{RP, RA\}$**.** $f$ **is** 1 **when resource** $h$ **is not banned.** (Note that UG policies should be disabled) | |
| *RP_HOST*, *RP_IM* | Fixed priority per site or per every resource discovered by an IM |
| *FR_MAX_BANNED* | $T_\infty \cdot (1 - e^{\Delta t/C})$, where $T_\infty$ is the maximum time that a resource can be banned, $\Delta t$ is the time since last failure, and $C$ is a constant that determines how fast the $T_\infty$ limit is reached |
| *FR_BANNED_C* | The value of the C constant in the above equation |
| **GWpilot PiS parameters:** | |
| -i < *PI* >, -t < *T* > | Pilot pulling interval and number of tries |
| -n < *unsig. int* > | Max. number of pilots |
| -o < *unsig. int* > | Over-submission of pilots |
| -c <*REQUIREMENT*> | Expression to constraint pilot submission |
| -r <*RANK*> | Expression to rank resources for pilot submission |
| -g < *unsig. short* > | % of guided pilots |
| -nosec, -s | Unsecure mode and shared mode of PiS |

provides great results and has been used in the tests shown in Subsection 6.3.

### *4.4. Effects of configuration on the scheduling layers*

Despite of the advantages provided to the users with dynamic scheduling policies, configuration parameters are not suitable for being dynamically modified. This is justified by the need of control by an administrator when GWpilot is used by multiple users and by the request of communities to improve certain metric, such as the throughput or the resource utilisation.

Some of these parameters are specific to PiS or have been mentioned as the pilot parameters. Most important for scheduling are the pilot pulling interval, the number of tries and the max number of pilots. In addition, requirement and rank expressions are allowed as configuration option of PiF. However, it could be desirable to partially perform the provisioning guidance in some types of calculations. For this reason, the number of guided pilots can be limited to a percentage. Another powerful feature is its capacity to perform a flooding of the infrastructures with a limited number of pilots, but above the real need. Therefore, administrators can effectively control the pilot production of the system.

However, as GWpilot is an embedded system, the options inherited from GridWay must be also included in this category because they have obvious implications on the GWpilot behaviour. For example, the maximum number of hosts limits the volume of resources (i.e. sites and pilots) that the system can manage. Other options limit the number of clients (application calls), users, tasks or pilot jobs in the system. Those options with a special significance were summarised in Table 3. In this sense, fair-share policies, i.e. fixed (FP), share (SH), deadline (DL) and waiting-time (WT) policies, could negatively impact on the effective dispatching time of certain pilots and tasks. In addition, (RP) policies constraint the use of resources and are mainly related to provisioning. In any case, they allow the management of the system in a way more close to a PMS service [19, 45].

Many of fair-share policies will be disabled if pilots cannot be shared among different users. For this reason, PiS and PiF sharing was allowed to enable MUPJs. Running in this mode, if the PiS is owned and initialised for a specific user, for example, the production manager of a VO, the other users must allow its distinguished name (DN) into the local *grid-map* file to correctly stage-in and stage-out their files through GridFTP. Nevertheless, although this procedure is feasible, it is an insecure method that cannot accommodate the policies of some infrastructures [73] because the pilots do not isolate users who are executing codes with the same DN role of pilot owner. This mechanism is allowed under the responsibility of the administrator or the VO manager, and because the user-identity-switching tools (e.g., gLExec [34]) are not widely installed and they are currently only required by large production VOs.

Other parameters are fundamental for both workload scheduling and provisioning. For example, the weight of rank (RA) expression in the task or pilot jobs should be enabled (i.e. set to one). Nevertheless, the usage (UG) statistics prioritise resources with shorter execution times per job, and consequently, are counter-productive for pilots, and should be disabled. The failure rate (FR) policies allow users and administrators to automatically discard pilots and sites that accumulate persistent task failures. The dispatch chunk and the scheduling interval determinate the volume of tasks and pilots dispatched, that is, the productivity. However, pilot submission should be lesser frequent than task generation. Subsequently, these last parameters and the pilot pulling interval mainly determinate the task turnaround overhead as will be demonstrated in the experiments section.

## 5. Functional comparison

Two frameworks have been selected as representatives of two different approaches in the design of pilot systems: DIANE, which is perhaps the most used application-oriented framework

on EGI-related infrastructures; and DIRAC as the PMS most adapted to other fields different to HEP calculations. The intention is two-fold: to compare GWpilot with other systems that propose different solutions, and to go into detail about technical issues not described in Section 2. These aspects are of importance if users and developers want to adapt their calculations to these frameworks and also to emphasise the advantages of GWpilot, but to properly introduce the significance of results obtained through the following Section 6 as well.

### *5.1. DIANE*

In first place, DIANE is a small pilot system conceived to integrate the user's application into a Python framework. In addition, it is implemented to be easily configured and managed by final users. For this purpose, a short script is provided to facilitate the installation. However, although DIANE is written in Python, the mechanism also downloads binaries of external software, and even Python libraries that are not included in default OS installations. Additionally, some of them are also required by pilots (*worker agents*), and they have to be downloaded into WNs before their execution. As mentioned in Subsection 2.4, its most important dependence is on omniORB, because CORBA is the basis of master-worker architecture of DIANE. Moreover, the pilot submission relies on Ganga, which has its own dependencies. Thus, a complete installation requires ~150MB, ~70 MB out of which are external binaries (beside of basic grid middleware). These issues do not constitute a serious problem for users if they compute on CERN related operating systems, i.e. Scientific Linux (SL) 5-6, where DIANE deployment is straightforward (for example on EGI infrastructures). Additionally, an experienced user can easily compile these dependencies and modify the installation script to adapt them to other platforms.

Getting started on grid only requires an elementary configuration of Ganga, i.e. editing some parameters in *~/.gangarc* file. Nevertheless this implies the background use of glite/UMD commands to remotely connect to a central WMS to perform the pilot provision. *Agent Factory* sequentially performs these operations. Additionally, there are few options to customise the provisioning behaviour, as it can be seen in Table 4. Users can only perform some control on pilot scheduling if they modify the *~/.gangarc* file for every application. These issues seriously limit the provisioning capacity of the system, as will be depicted in Subsection 6.2.2.

*Worker agents* actually notify a complete set of characteristics to *Run Master*, but this feature is not completely used in the framework. For example, ranking expressions are not available. In this sense, extensions of DIANE have been implemented to take advantage of this characterisation to incorporate some generic self-scheduling algorithms into the workload scheduling layer. In particular, AWLB [33, 66] is able to find sub-optimal distributions of variable sized bags of tasks (BoTs) among the characterised pilots, according to a CPU and bandwidth consumption profile previously determined for the application. Nevertheless, the development of this type of approaches is restricted to advanced Python programmers. Moreover, they can expend many efforts in modifying DIANE code (the *Master*, *Scheduler*, *Factory* and *worker agents*) to accomplish the needs of certain legacy applications, which perhaps, were already implemented following a distributed computing standard (such as DRMAA or SAGA), or even following other extended specifications (such as Ganga). Additionally, task requirements can be set if *worker agents* are again modified. That is, it is needed to maintain a specific pilot by every special application. Other weakness is the impossibility of sharing pilots or statistics among running applications.

Therefore, the available workload scheduling for conventional users is based on FCFS. The advantages of this approach were commented in Subsection 2.6.2: maximises the usage of resources and maintains task overheads under minimum possible. For these purposes, DIANE allows the customisation of policies through *config.WorkerAgent*, *config.RunMaster* and *input.scheduler* variables (see Table 4). All of them can be dynamically set into the code, but due to their generality, only the related to the management of tasks ones are suitable to be modified during one calculation.

### *5.2. DIRAC*

DIRAC is a PMS that provides an installation script following the DIANE fashion, i.e. compiled dependencies are downloaded together with the DIRAC software. However, external dependencies expend now ~700 MB, while DIRAC software only ~230 MB. This implies continuous upgrades to maintain compatibility. Moreover, although the software related to LHCb can be omitted, any basic DIRAC server requires ~40 modules (*Agents* and *Services*) running associated to a set of ten databases. Every module runs as a system daemon that continuously fills its database and generates logs which exponentially increase the disk usage. All of them must be taken into account during the installation, requiring ~400 parameters. Many of these options have not default values and the administrator must search about their significance. The ones related to the acknowledgement of failed tasks or pilots are measured in hours or days. Some scheduling functionalites are fixed. Those options related to the comparison performed in this work are depicted in Table 4, and clearly demonstrate that DIRAC is only oriented to constitute a platform for multi-project production, where some skilled administrators and VO managers have the role of maximizing the throughput of long jobs.

However, the main obstacle to customise some scheduling in DIRAC is that its components are designed and optimised to efficiently accomplish workloads similar to the ones generated by the LHC Experiments. As a result, its design tightly couples task scheduling and provisioning to optimise these calculations, and does not tackle the possibility of setting different requirements for tasks. For example, DIRAC offers compatibility with applications based on JDL (Job Description Language) templates. That is, it provides users with commands like the glite/UMD WMS ones. Contrary to JSDL (Job Submission Description Language), JDL is not standardised but it is widely extended and allows the inclusion of pseudo-scripts and ClassAd expressions. Nevertheless, most of the JDL parameters related to scheduling are overlooked: users can not perform any ranking and can only constraint resources with four types of require-

ments (see Table 4). Besides JDL, DIRAC also provides developers the REST and Ganga interfaces, and its specific Python library. Nevertheless, they do not also allow the specification of more requirements.

Therefore, the scheduling is summarised in two levels: first, the system classifies tasks according to their *CPUTime* but also to other specific requirements not set by user. Subsequently, it places tasks into a queue specifically created for this classification. Posteriorly, *TaskQueueDirector* *Agent* requests to any gLite/UMD WMS the information about resources that accomplish these common requirements. Then, the system filters those obtained resources according to benchmarks compiled during previous executions of pilots. Finally, one pilot per task is submitted to a concrete resource, if enough resources are listed. That is, the WMS performs the grid match-making, while DIRAC performs a parallel scheduling and takes the final decision about where pilots will run.

Consequences are far away from being tied to UMD middleware. First, their scheduling parameters are not really configurable by administrators. The depicted double-matching mechanism is based on specific GLUE tags and local measurements. Thus, to change the *Rank* is not recommendable to inexpert administrators because it has influence on other processes. Second, pilots can only host one task because the matchmaking is optimised for jobs whose duration approaches the maximum wall-time at remote resources. One solution can be to run the *JobAgent* together with the pilot to fetch short tasks. Nevertheless, the maximum number of tasks that a pilot can accomplish is 100 (*MaxJobsInFillMode* parameter), and includes the number of retries when the corresponding queue is empty. Third, besides *TaskQueueDirector*, many other DIRAC modules participate in the scheduling process and they generate overheads. In this sense, WMS commands are relatively slow and sandboxing files in DIRAC server increases the penalty in task turnaround. These circumstances will be explained in the results Subsection 6.2.2.

### *5.3. Comparison*

Besides other features mentioned in Section 3, GWpilot differentiates among these systems by its installation (which is mainly based on OS packets and requires ∼5 MB ); its configuration (which only requires ∼30 lines in two completely customisable files and some *sudo* tips); its remote compatibility and lightweight (since pilot is a short script file that only requires Python 2.4.3 as minimum release in WNs); and its management of user data (that allows staging files locally and through standards protocols). In addition, GWpilot is multiuser, allows MUPJ (opposite to DIANE) and compiles general execution statistics (opposite to DIRAC).

However, the main differences from user's and developer's point of view are that GWpilot allows them to directly run their legacy applications (written in diverse languages and following accepted standards), and fully supports the customisation of the whole scheduling at user-level, guiding both task scheduling and provisioning. Thus, developers can dynamically include customised policies in these legacy applications basing them on a complete characterisation of resources without the need of modifying GWpilot code. Therefore the GWpilot is so flexible that is possible to incorporate personalised schedulers on top of the system.

Obviously, DIRAC and DIANE frameworks have advantages over GWpilot. The former has been used during years in production and can support hundred thousand of jobs and thousand of users, which has not been tested with GWpilot, neither with GridWay. Additionally it offers a complete web page that shows detailed statistics and facilitates users with different roles for their common procedures: to control site availability, to inspect tasks and pilot logs, to list statuses of tasks and cancel them, etc. In particular, the possibility of permanently banning a resource is not available in GWpilot without restart their daemons. DIANE offers the highest performance in terms of overhead in task scheduling when the default FCFS was used. An evidence of this affirmation is that the option *PULL_REQUEST_DELAY* must be set to avoid the overload of *Master*.

## 6. Experiments

The aim of the experiments is to demonstrate the viability and suitability of GWpilot, as well as its performance as an improvement achieved in comparison with other pilot systems, when common scientific applications are executed on resources belonging to large production DCIs. For this purpose, GWpilot must be tested on a real infrastructure and measured with applications that create non-ideal conditions for a computational distributed environment, i.e., filling the system with a high volume of variable-duration short tasks in a continuous flow, which must be dispatched individually. This approach is of enormous importance because centralised pilot systems usually validate their performance only with long tasks. This methodology is well founded for the specific calculations for which these systems were initially implemented, i.e. the LHC production. However many user applications are composed of short tasks that should beneficiate from their extensive distribution.

The objective is not to expose the advantages of certain complex scheduling algorithms that GWpilot can incorporate but to show how the default GWpilot functionalities result in a valuable improvement over other systems.

Therefore, two different types of experiments are carried out in this work. First, the behaviour of two representative frameworks (DIANE and DIRAC) is compared with GWpilot. To better analyse the results obtained, only one-user and one-task pilot jobs over a unique grid VO will be utilised. Additionally, general issues on their configuration and adaptation are commented, not only to be contrasted with GWpilot capabilities, but also to establish the same scheduling requirements to perform equivalent tests among them. Thus, the basic performance of GWpilot is properly measured with respect to some existing approaches, and therefore an important performance gain is expected for any other feature of GWpilot to be used.

In this sense, the second experiment shows how the characterisation of pilots and the customisation of scheduling can be easily performed at user-level to obtain better resources and, consequently, to reduce final makespan of several application

Table 4: Scheduling policies with similar significance for the pilot systems compared in this work. Values set to accomplish the experiments are shown.

| | GWpilot | DIANE | DIRAC |
|---|---|---|---|
| **Provisioning new pilots:** | | | |
| Max. amount and overload | PiS option (*gwd.conf*), PiF params. (manually) | *Agent Factory* or *diane-submitter* (*diane − worker − number*) param. (manually) | Same as number of tasks and adds: *TaskQueueDirector/extraPilots*(= 2) |
| Max. suspension in LRMS or GS | PiS option (*gwd.conf*), PiF param. (manually) | *Agent Factory* or *diane-submitter* (*pending − timeout*) param. (manually) | *ExpireTime* is not set in the pilot JDL, then remote WMS waits its default timeout (typically 1 day) |
| Submission limits | *DISPATCH_CHUNK*(= 100) (*sched.conf*) | It is secuential, but Agent Factory/diane-submitter limit with (*diane − max − pending*) param. (manually) | *TaskQueueDirector/pilotsPerIteration*(= 100) |
| Ranking resources | PiS option (*gwd.conf*), PiF param. (manually), or collected from any *RANK* in tasks (dynamic) | *Rank* option (~ */.gangarc*), or Agent Factory (*square − fitness*) param. (manually set, but it is a fixed policy, based on completion rate: [(*running* + *completed*)/*total*]) | *TaskQueueDirector/glite/Rank* (Expression based in GLUE descriptions of CEs obtained from top-BDII) |
| Constrainting Resources | PiS option (*gwd.conf*), PiF param. (manually), or collected from any *REQUIREMENT* in tasks (dynamic) | *Requirement* option (~ */.gangarc*), or *Agent Factory* or *diane-submitter* param. (manually, but only allows list of CEs, i.e. round-robin) | Fixed in code and based on *Rank* expression and HepSpec06 of CEs also obtained from top-BDII. Additionally, only 4 requirements are also collected from task JDLs to generate constraints: *CPUTime*, *Site*, *BannedSites*, and *Platform* |
| External broker | — | *glite_wms.conf* | *TaskQueueDirector/glite/ResourceBrokers* |
| **Pilot behaviour:** | | | |
| Pulling new task | Inmediately, and then *PI*(= 30*s*) | *config.WorkerAgent.PULL_REQUEST_DELAY*(= 0.2*s*) | *JobAgent/SubmissionDelay*(= 10*s*), and then 120s |
| An *idle* pilot ends | $T \cdot PI$(= $20 \cdot 30s$) | One attempt every: *config.WorkerAgent.HEARTBEAT_DELAY*(= 10*s*) Pilot ends if an attempt last more than: *config.WorkerAgent.HEARTBEAT_TIMEOUT*(= 30*s*) | *JobAgent/StopAfterFailedMatches*(= 10) failed attempts, or when the number of completed tasks and failed attempts reachs to: *TaskQueueDirector/glite/MaxJobsInFillMode*(= 100) |
| **Policies to discard pilots:** | | | |
| Outage | $T \cdot PI$(= $20 \cdot 30s$) | *config.RunMaster.LOST_WORKER_TIMEOUT*(= 60*s*) | *PilotStatusAgent/PilotStalledDays*(= 3*d*) |
| Idles | $T \cdot PI$(= $20 \cdot 30s$) | *config.RunMaster.IDLE_WORKER_TIMEOUT*(= 600*s*) | — |
| Task failure | Immediately | *input.scheduler.policy.* *REMOVE_FAILED_WORKER_ATTEMPTS*(= 1) | *JobAgent/StopOnApplicationFailure*(= *true*) (on first task failure, pilot ends) |
| **Policies to manage tasks:** | | | |
| Execution attepms | *RESCHEDULE_ON_FAILURE* (= *no*) , *NUMBER_OF_RETRIES* (= 3) in tasks (dynamic) | *input.scheduler.policy.* *FAILED_TASK_MAX_ASSIGN*(= 3) *LOST_TASK_MAX_ASSIGN*(= 3) | Reschedule fixed in 3 retries |
| Avoid ending pilots | *REQUIREMENT* = "*QUEUE_MAXTIME* > (30*m*)" in tasks (dynamic) | *input.scheduler.policy.WORKER_TIME_LIMIT*(= 0*s*) (max. time in execution) | JobAgent compares *CPUTime* requirement set in task JDL and the remaining time notified by pilot before matchig. |
| Matching time | *SCHEDULING_INTERVAL* (= 10*s*) (*gwd.conf*) | Inmediately | *TaskQueueDirector/ListMatchDelay*(= 10*s*) |
| Avoid overloaded or failing pilots | *SUSPENSION_TIMEOUT* (= 61*s*) in tasks (dynamic) | *config.RunMaster.LOST_WORKER_TIMEOUT*(= 60*s*) | *StalledJobAgent/StalledTimeHours*(= 6*h*) *StalledJobAgent/FaledTimeHours*(= 2*h*) |

types. For this purpose, three GWpilot instances support the execution of the same four applications, but every one perform a different scheduling for their tasks and provisioning.

Additionally, other performance statistics are provided to determine the current and future scalability of GWpilot, as well as overhead measurements that justify its design.

### *6.1. Test bed setup*

The *fusion* VO has been for the calculations because it offers a large, heterogeneous and overloaded number of resources shared with other highly demanding VOs. When the tests were performed, the *fusion* VO counted on more than 40 computing elements (23 sites), and up to 29,464 slots. Of course, few of them are actually available, but every framework should obtain 1,000 slots. This value is used as the minimum amount of pilots that every system will request during the experiments.

To perform accurate comparisons among frameworks, three identical virtual machines (4-cores, 24 GB RAM; SL 6.3; UMD UI 2.0.2-1) are configured with DIRAC (v6r8p14 release), DIANE (2.4) and GWpilot (on GridWay 5.14), running on a dual Xeon X5560 (16 cores, 2.8GHz). Every instance stores their logs, user data and databases on virtual disks created on RAM. This is done to avoid the interference among pilot systems on servers' performance, because only one framework is started in every virtual machine booted, and the intensive I/O operations are isolated in every reserved memory. Additionally, three different user certificates are used during the experiments. This is to assure a fair-share between tests at remote queues, a key point that is indispensable to achieve accurate results on an overloaded infrastructure such as the *fusion* VO in EGI.

To ensure the accuracy of the results obtained, tests are run in parallel and configuration options are set as similar as possible. Table 4 shows these configuration equivalencies which are also

Table 5: Time complexity of long multiplication and real time measurements

| $< Number\ of\ tasks >$ $mod\ 10$ | $\Theta(n^2)$ | Xeon X5365 3GHz (Launch date: 2007) | Xeon E5620 2.4GHz (Launch date: 2012) |
|---|---|---|---|
| **Very short tasks:** $<$ *lower limit* $>= 3$ ($n = 30,000 \cdots 75,000$) | | | |
| 0 | $\Theta((3 \cdot 10^4)^2)$ | 66.97 s | 84.50 s |
| 1 | $\Theta((3.5 \cdot 10^4)^2) = (1,1\overline{6})^2 \cdot \Theta((3 \cdot 10^4)^2)$ | 91.18 s | 115.03 s |
| 6 | $\Theta((6 \cdot 10^4)^2) = 2^2 \cdot \Theta((3 \cdot 10^4)^2)$ | 269.64 s | 338.31 s |
| 8 | $\Theta((7 \cdot 10^4)^2) = (2.\overline{3})^2 \cdot \Theta((3 \cdot 10^4)^2)$ | 369.03 s | 459.81 s |
| 9 | $\Theta((7.5 \cdot 10^4)^2) = (2.5)^2 \cdot \Theta((3 \cdot 10^4)^2)$ | 419.43 s | 529.27 s |
| **Short tasks:** $<$ *lower limit* $>= 9$ ($n = 90,000 \cdots 135,000$) | | | |
| 0 | $\Theta((9 \cdot 10^4)^2)$ | 605.21 s | 761.04 s |
| 1 | $\Theta((9.5 \cdot 10^4)^2) = (3.1\overline{6})^2 \cdot \Theta((3 \cdot 10^4)^2)$ | 672.86 s | 851.62 s |
| 6 | $\Theta((12 \cdot 10^4)^2) = 4^2 \cdot \Theta((3 \cdot 10^4)^2)$ | 1074.32 s | 1354.96 s |
| 8 | $\Theta((13 \cdot 10^4)^2) = (4.\overline{3})^2 \cdot \Theta((3 \cdot 10^4)^2)$ | 1262.12 s | 1587.38 s |
| 9 | $\Theta((13.5 \cdot 10^4)^2) = (4.5)^2 \cdot \Theta((3 \cdot 10^4)^2)$ | 1360.91 s | 1713.19 s |

taken as a model for later experiments. In this sense GWpilot parameters are adapted to be as the default policies used in DIANE. However, the default options are maintained for DIRAC with the exception of the activation of filling mode, because no comparison would be possible without it. Therefore, this is the configuration found by any user at central DIRAC servers.

To eliminate the impact of provisioning policies, resource ranking and filtering capabilities are disabled on GWpilot for these first experiments. PiFs (GW PiF and *Agent Factory*) have been restricted to create a maximum of 1,000 *running* pilots. The maximum suspension timeout, before cancelling a pilot job whenever it is queued at the remote LRMS for a long time, has been set to 30 minutes. This value is considered reasonable according to the duration of the first tests (i.e. 3-10 hours). Additionally, it provides a level playing field with DIANE's provisioning mechanism, even though GWpilot could work better with lower timeouts [74, 75]. However, banning feature is enabled (set to 1 hour) in GWpilot. This is necessary to avoid sending pilots again to the same failing resources (the suspension timeout is considered as a failure in this work). Thus, to improve in a similar way the provisioning in DIANE executions, the *square-fit* option is passed to *Agent Factory*.This implies that DIANE will guide the provisioning process for some tasks. Moreover, DIRAC will also perform by default the profiling of every site and will use statistics to select resources. Therefore, GWpilot is *a priori* put at a disadvantage in the provisioning phase. Finally, caching files option is also disabled in GWpilot, while DIANE and DIRAC perform staging operations as usual.

However, other GWpilot parameters must also be set to perform any calculation, although they effectively limit the usage of the infrastructure. They were configured to allow the submission of a maximum of 100 pilots per site (which is defined with the *MAX_RUNNING_RESOURCE* statement). Additionally, GWpilot is allowed to schedule a maximum of 100 tasks (to pilots) and pilots (to sites) every 10 seconds. These values are set for similarity with DIRAC, although they limit the capacity of GWpilot to access resources with respect to the other frameworks.

Finally, the last two parameters in the GWpilot configuration determine the pilot turnaround overhead and the pilot discarding rate of the pilot system, respectively: the pulling interval ($PI$) for GWpilot, which is established to 30 seconds, and the maximum retries number ($T$), which is set to 20. These values are selected to be similar to the discarding timeout for DIANE's idle *worker agents* (*IDLE_WORKER_TIMEOUT*).

### 6.2. First experiment: comparison with other pilot systems

The shorter duration of tasks implies a higher impact of overheads on the computation performance and on the load supported by systems. In addition, most users do not want to deal with grouping their tasks. Developers are also aware of how the failure rate increases when big bags of tasks are used, and would rather massively distribute minimal tasks to reduce the final makespan. For these reasons the comparison among pilot systems is made with short tasks. Additionally, the intention is to simulate the typical behaviour pattern of a user that submits a daily calculation, for example, carried in background during the night. Therefore, a test with a maximum wall time of 5-10 hours is selected for this experiment.

#### 6.2.1. Simple calculation

For this first experiment, long (or standard) multiplication is selected to be easily measurable and reproducible. The time complexity of multiplying two *n-digit* numbers using long multiplication is $\Theta(n^2)$. Thus, task duration can be easily controlled modifying $n$ as shown in Table 5. To obtain these data, long multiplication was implemented in C language and compiled with *gcc* 4.4, resulting in a binary of 13KB. The executable requires the *n-digit* as a parameter to randomly generate two numbers of that size. Then, it multiplies both of them, and stores the result in a file of $\sim 6 \cdot n$ Bytes.

To generate an effect similar to a real user calculation (but also controlled for further analysis), tasks are submitted with different $n$ values along the tests following this pattern:

$$< lower\ limit > \cdot 10^4 + (5 \cdot 10^3 \cdot (< number\ of\ tasks > mod 10))$$

Thus, setting $<$ *lower limit* $>$ to 3 assures that the duration of task ranges between ~67s and ~420s on the first machine mentioned in Table 5, which is taken as a lower reference machine

of the infrastructure due to their manufacture date. However, significantly lower execution times are not expected in this case due to long multiplication only depends on the velocity of the processor.

Perhaps, processing tasks that last less than five minutes has not much sense in a distributed environment (with the exception to of real-time applications). The sole purpose to carry on this type of calculation is to perform a stress test of the pilot systems. For this reason, a second test that contains an input set that guarantees task duration above 10 minutes is also performed, i.e. setting < *lower limit* > to 9.

The number of tasks of both tests can be determined according to the lower reference machine. $3 \cdot 10^4$ tasks would last approximately 1 hour and 51 minutes on 1,000 X5365 cores, and would generate 8.8 GB as output if a < *lower limit* >= 3 is selected, while the same volume of tasks would last 8 hours and generate 18.86 GB when a < *lower limit* >= 9 is set. However, these calculations will last much more due to the real availability of resources in grid.

A script that creates JDLs and makes use of commands was implemented to port long multiplication to DIRAC. This script is as simple as any conventional user will implement. It sequentially submits and checks the statuses of tasks. If ended tasks are detected, outputs are checked before to submit new tasks. Else the script waits 60 s. However, unlike the other approaches that directly store the outputs in a local user directory, DIRAC is based on the sandboxing of these outputs to be downloaded later by the user. It is noteworthy to mention that DIRAC provides commands for searching for text in the output files without having to download them. Nevertheless, they can last minutes due to the volume of tasks managed in this experiment. These delays avoid any comparison with the other pilot systems and they are not used. Additionally, every task should be deleted after downloading its outputs and checking if they are correct. Every operation requires 1.2-1.5 s. despite of the script is launched on the same machine that runs DIRAC to remove network overhead. This behaviour is unfeasible because only checking the statuses of 1,000 tasks would last more than 2 minutes. For this reason, every command launched will manage up to 100 tasks together.

To make a more proper comparison and to measure the performance of GWpilot CLI, a similar script was implemented to generate templates and manage the long multiplication tasks through GWpilot. However, with DIANE, there is no other possibility than to wrap long multiplication into DIANE libraries, and so it was performed. Naturally, both implementations will check the outputs of every task to ensure the completeness of results. The maximum number of managed tasks is limited to 1,000 in the three implementations. With respect to the basic policies for managing tasks in GWpilot, the same approach for provisioning is followed: neither ranking, nor requirements were dynamically set on tasks by the application, with exception of the necessary ones to avoid ended or failing pilots (see Table 4). The task suspension timeout has to be set slightly above the pilot pulling interval in order to quickly remove tasks from pilots that could be failing.

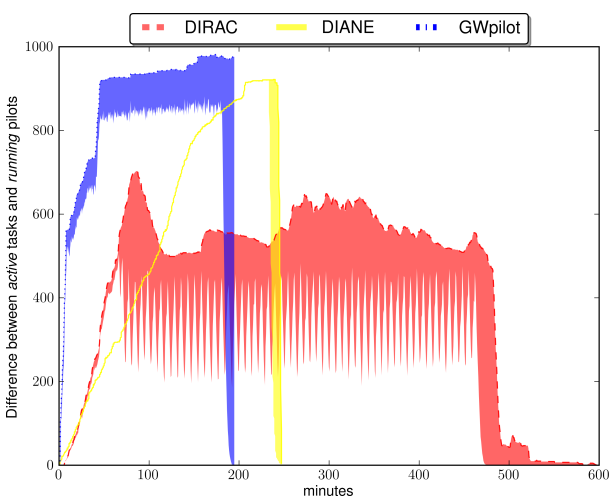
*6.2.2. Results*

Both short (< *lower limit* >= 3) and long (< *lower limit* >= 9) tests were carried out three times to assure accurateness. Makespan obtained with every pilot system is listed in Table 6, which must be provided as the principal speedup measurement from a user point of view. However, to allow a proper comparison among systems and against other approaches, other performance metrics used in previous works [47, 76, 44, 52] are considered. Thus, in addition to the final makespan parameter, Table 6 compiles the number of pilots submitted and effectively enrolled as a measurement of the efficiency in provisioning. Additionally, the average number of pilots actually running and the filling rate of these pilots with tasks are provided. These values are measured before the last 1,000 tasks were remaining in the systems because the end of calculation is not representative of the usual utilisation of resources in any experiment. The number of failed tasks is also added.

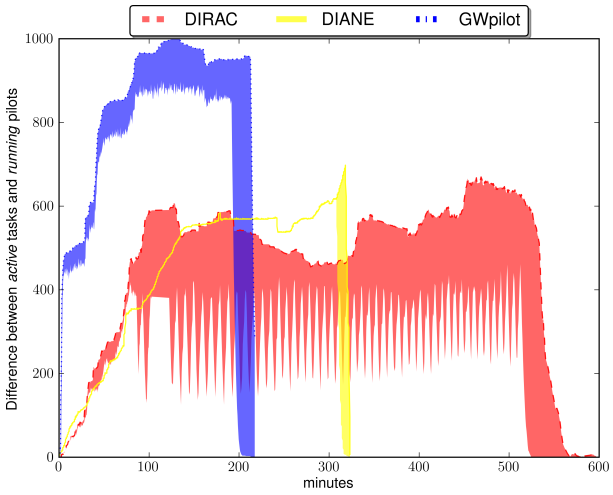
To complement this information and illustrate the evolution of the tests, Figures 5-(a, b, c, d, e, f), show the difference between the number of *running* pilots and the number of *active* tasks in every framework. These figures have been obtained by processing DIANE (*master.j*), DIRAC (*Matcher* and (*JobStateUpdate*), and GWpilot logs.

First observable issue in the figures is the improved capacity of GWpilot for provisioning. It is able to obtain and retain the requested resources one or two hours before than the other systems. Performance achieved by GridWay in early-binding scheduling is known, and results obtained in GWpilot tests should be better than the ones using gLite/UMD WMS, according to previous works [74, 75]. However, this deserves a deeper explanation because most of the scheduling advantages of GWpilot have been disabled in this experiment and the overload of pilots is added. This capability is consequence of the modular design of GWpilot. MAD in charge of controlling job execution in CREAM sites conceptually follows a similar implementation to GW PiS: both manage the continuous data flow from and to system core and remote sites. Additionally, it uses minimal middleware libraries, unlike middleware commands. However, the Scheduler is always performing the match-making among pilots and resources, and among tasks and pilots. For this purpose the system maintains the Scheduler aware of the data flow coming from GW PiS, CREAM and IM MADs. With this information Scheduler performs its match-making algorithm over a limited number of pilots and tasks ($DISPATCH\_CHUNK$), once per interval ($SCHEDULING\_INTERVAL$). Therefore, due to its modular architecture, CREAM driver and PiS can potentially manage thousands of pilots and tasks more quickly and concurrently than DIRAC or DIANE solutions. In fact, it is mainly limited by the computational complexity of the algorithm implemented in the Scheduler.

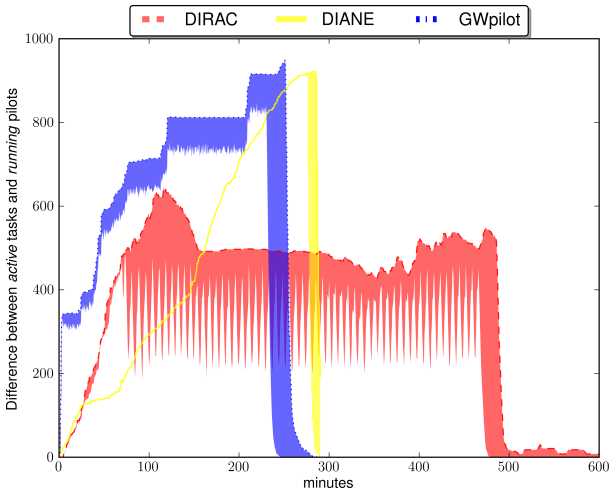
Unlike GWpilot, the other frameworks use gLite commands to submit pilots (or even to perform match-making requests in the case of $TaskQueueDirector$). Every contact with the WMS spends above 3 s. DIRAC implements such capability by a multi-threaded engine and is less exposed to these overheads. Nevertheless, it offers a similar performance to DIANE if the



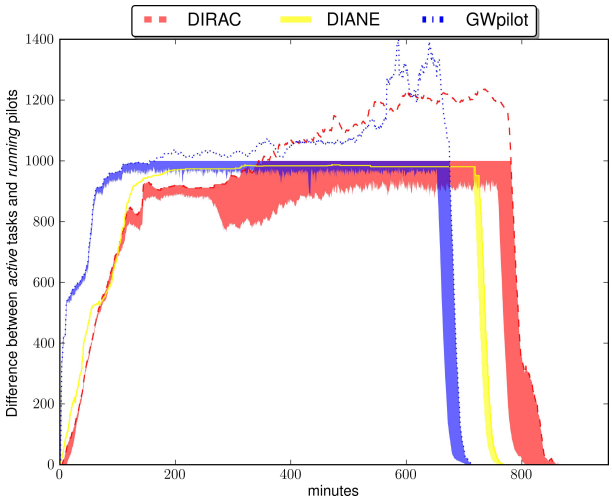

(a) $n = 30,000 \cdots 75,000$



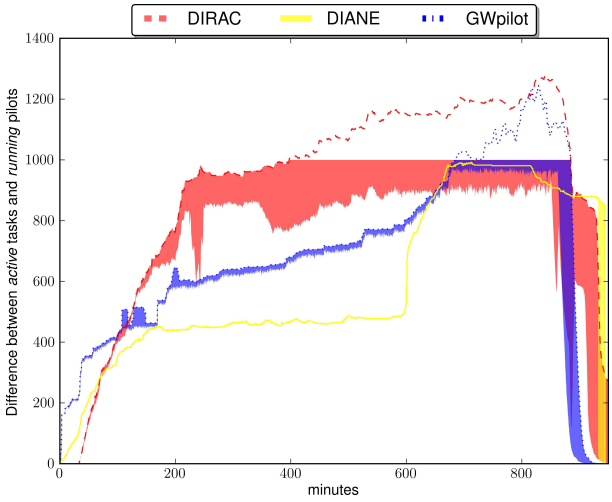

(b) $n = 30,000 \cdots 75,000$



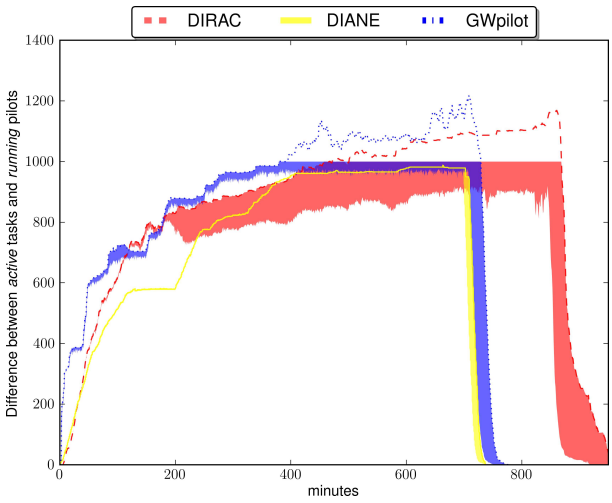

(c) $n = 30,000 \cdots 75,000$




(d) $n = 90,000 \cdots 135,000$




(e) $n = 90,000 \cdots 135,000$




(f) $n = 90,000 \cdots 135,000$

Figure 5: Difference between the number of *running* pilots and *active* tasks in first experiments (blue represents GWpilot, red represents DIRAC, and yellow represents DIANE). There is an upper limit of 1,000 managed tasks per application run on every pilot system. Left (right) column cases a, b and c (d, e and f) corresponds to three different tests with $n = 30,000 \cdots 75,000$ ( $n = 90,000 \cdots 135,000$).

Table 6: Results obtained in first experiment. Values of pilots submitted between parenthesis correspond to the number of pilots guided by DIANE *Agent Factory* to a specific site. Values for *mean up* and *filling rate* are measured before the last 1,000 tasks were remaining in the systems due to the end of calculation is not representative of the usual utilisation of resources.

| Test | System | Makespan | Tasks failed | Pilots | | | |
|---|---|---|---|---|---|---|---|
| | | | | submitted | enrolled | mean up | filling rate |
| **Short tests (very short tasks):** < *lower limit* >= 3 ($n = 30,000 \cdots 75,000$) | | | | | | | |
| (a) | DIANE | 4h: 6m: 49s | 466 | 2563 | 1381 | 525.08 | 99.76 |
| | DIRAC | 7h: 56m: 54s | 4 | 2165 | 1874 | 518.73 | 63.50 |
| | GWpilot | 3h: 13m: 32s | 82 | 2490 | 1011 | 797.72 | 86.85 |
| (b) | DIANE | 5h: 22m: 26s | 1991 | 3431 | 2797 | 428.90 | 99.80 |
| | DIRAC | 9h: 3m: 1s | 8 | 2726 | 1751 | 460.91 | 62.98 |
| | GWpilot | 3h: 27m: 01s | 135 | 2996 | 1054 | 785.62 | 86.99 |
| (c) | DIANE | 4h: 50m: 42s | 608 | 3323 | 1570 | 421.98 | 99.75 |
| | DIRAC | 8h: 21m: 56s | 6 | 2049 | 1297 | 461.74 | 68.50 |
| | GWpilot | 4h: 06m: 42s | 62 | 4655 | 954 | 648.23 | 87.80 |
| **Long tests (short tasks):** < *lower limit* >= 9 ($n = 90,000 \cdots 135,000$) | | | | | | | |
| (d) | DIANE | 13h: 16m: 02s | 522 | 1989(932) | 1531 | 893.99 | 98.63 |
| | DIRAC | 13h: 24m: 0s | 486 | 4226 | 3454 | 968.97 | 87.29 |
| | GWpilot | 11h: 33m: 32s | 183 | 9032 | 3687 | 1001.63 | 94.49 |
| (e) | DIANE | 15h: 29m: 36s | 3140 | 6435(2330) | 4137 | 588.45 | 99.93 |
| | DIRAC | 16h: 6m: 3s | 650 | 12888 | 3095 | 953.03 | 85.43 |
| | GWpilot | 15h: 08m: 36s | 589 | 16369 | 2213 | 670.16 | 95.18 |
| (f) | DIANE | 12h: 16m: 17s | 2322 | 3919(1393) | 3295 | 774.32 | 99.62 |
| | DIRAC | 15h: 19m: 22s | 822 | 3135 | 2537 | 894.35 | 88.58 |
| | GWpilot | 12h: 34m: 49s | 151 | 9444 | 2645 | 880.0 | 94.61 |

number of failed tasks is observed. Every failed tasks triggers the discarding of its hosting pilot (see Table 4) in DIANE. Submission performed by *Agent Factory* is completely sequential, and subsequently the maximum number of slots provisioned is limited to approximately 1,000 per hour for DIANE. On the other hand, GWpilot is able to submit this volume of pilots in 100 seconds, according to the configuration selected in this work. Thus, GWpilot can immediately start the replacement of failing or suspended pilots. However, this feature can not be enough in certain situations as the one presented in test (e), where the infrastructure is overloaded. In this case, allowing large suspension times at remote queues improve the provisioning (see DIRAC behaviour in Figure 5 -(e)).

In view of the results, GWpilot is clearly more adequate to accomplish short duration experiments, as the ones described in Figures 5-(a, b, c). In experiments longer than 15 hours, the benefit is slightly lower. However, when many pilots end due to reaching the maximum wall-time at remote queues (usually 24 hours running), the improved provisioning capacity of GWpilot will maintain the number of resources appropriated unlike other solutions.

Despite of the drawback, it seems that DIANE progressively trims the advantage that GWpilot holds in some tests (f, a, c), although not in others (d). That is because its task completion rate is higher than the one achieved by GWpilot when it reaches a similar number of pilots (a, c) and because *Agent Factory* could be appropriating more powerful resources. It is noteworthy to mention again that DIANE performs such a type of guided provisioning (see Table 6) only on long tests (with < *lower limit* >= 9). That is so because *Agent Factory* needs time to compile enough data about *running* pilots in order to start submitting new ones to certain resources. DIRAC is always evaluating performance of pilots to select suitable resources. However, these features do not seem to be an effective advantage over GWpilot through the tests. Other reason could be the existence of overheads, which will be discussed in the following paragraphs.

The effective resource utilisation is a clear indication of how overheads decrease overall performance. This is usually measured using the area between *running* pilots and *active* tasks graphs [76]. DIANE execution is plotted as a line, with the exception of the end of computation when many pilots are still up, but there are no more tasks to execute. That is so because DIANE reaches average filling rates of about 99%, i.e. it performs the highest pilot utilisation, as expected: with DIANE, any ended task is immediately detected by the user application because the application itself is embedded in DIANE. Thus, a new task that substitutes the previous one is immediately generated, and subsequently dispatched. This process takes less than 0.4 s for every task because it has no scheduling overhead associated. The FCFS approach does not require expending time in solving match-making algorithms.

The jagged shape of the displayed *active* tasks line in GWpilot and DIRAC executions is due to their arbitrary termination. It is justified by the alternation of light tasks with heavy ones proposed as test input. Although this behaviour could be attenuated by decreasing the pulling (*PI*) and the scheduling interval, or increasing the dispatching chunk, it is also determined by the implementation of the script that manages the execution of the application. This script is not able to process and supply new tasks to GWpilot when their duration is very short (these overheads will be properly described in Subsection 6.4). On

the other hand, match-making complexity is so simple in this experiment that it takes the Scheduler less than one second to solve it. Thus, the overhead originated by Scheduler is an issue to be analysed through next subsections, where ranking is extensively used. However, other overheads have influence on the task turnaround time in this experiment and reduce the task completion rate. One of them is file staging, but another one is the number of failed tasks that trigger the banning of pilots by a period of time (see Table 6). However, overheads are actually appreciable with task duration below five minutes. Results obtained in other cases seem to be comparable with those of a FRFS scheduling approach [76], or even in [52], as shown in Figures 5-(d, e, f). This is indicative of the design and implementation feasibility of GWpilot.

Other added overheads are hindering fill the pilots in DIRAC executions. In short tests (a, b, c), the slowness of CLI commands dramatically decreases the number of tasks submitted. That is due to the great number of DIRAC modules that are working together to process the amount of requests coming from the application. These processes maintain the 4 CPU cores of virtual machine above 60% of usage. Consequently, commands take up to 20 seconds to return the results. This is the reason for the pronounced jagged shape of running tasks in these tests. The drawback is clearly attenuated in long tests (d, e, f), however scheduling and pilot overheads still prevent reaching filling rates similar to the other approaches. Explanation is simple: although *Matcher* is able to dispatch most of the tasks in milliseconds [47], tasks queues are empty many times. Then, many pilots must wait 120s to try again getting a task. To classify tasks in queues is not an immediate mechanism, as well as the process in which the tasks are considered completed. Thus, there is a delay between accepting a task and dispatching it; and there is another delay between the task being done and its output being available for download. In the case of long tests (d, e, f), commands do not have influence on the filling rate because the script always maintain 1,000 tasks in the DIRAC system without problems. Nevertheless, the script can not detect finished tasks although their hosting pilots had completed the work. Additionally, the sandboxing mechanism delays the processing of ended tasks. Consequently, many tasks in long experiments are in unproductive states.

Therefore, the underutilisation of pilots and the delay on obtaining results is a problem for individual users that plan to use DIRAC to execute applications composed by short tasks. However, this issue is not a drawback for the managers of large collaborative projects where many clients use PMS at the same time, because pilots can always be filled with other user's tasks if the MUPJ capability is enabled. Thus, DIRAC will maintain the appropriate throughput and resource utilisation to accomplish its associated projects.

### *6.3. Second experiment: customised scheduling*

Through the last subsection the performance of GWpilot was compared with other two pilot frameworks. Those tests only provide a base measurement of the performance gain of GWpilot because no guidance in scheduling was enabled with exception of banning the failing resources. Now, the interest is to perform real calculations as any user usually does, i.e. customising requirements of his legacy applications and starting several calculations at same time. Therefore, in these experiments, some features are introduced that differentiates GWpilot among the other systems evaluated. In particular, sharing pilots among applications is not allowed by DIANE and only partially by DIRAC (due to the $MaxJobsInFillMode$ constraint). Both systems do not offer users suitable mechanisms to dynamically incorporate policies that had influence on provisioning and even on workload scheduling. They neither provide standardised interfaces to support legacy applications.

Moreover, the approach is to insist on the importance of scheduling variable short tasks (with an execution time shorter than 20 minutes) in an overloaded infrastructure because this is the worst potential scenario that a pilot system (and any scheduler) can face. Thus, the features introduced should demonstrate that they improve the execution of user applications while not overloading GWpilot system with excessive overheads. This will be demonstrated through this subsection executing again long multiplications but also several instances of a legacy application at the same time, simulating a multi-application or multi-user environment. Overheads will be evaluated in the last subsection because also compiles measurements made during the previous experiment.

#### *6.3.1. The legacy application*

A well-known grid application that can supply short tasks is selected from among many others [77, 78] that are currently executed with the GWpilot framework. DKEsG (Drift Kinetic Equation solver for Grid) [79] calculates the neoclassical transport in fusion reactors and has already been successfully deployed on EGI in *fusion* VO.

Currently DKEsG follows a producer-consumer design pattern, where a maximum number of DKEsG-Mono tasks is managed by the application through the DRMAA library. Every task is compiled as a 2 MB executable that needs an input file of ∼5 KB and produces output files of ∼10 KB. Every independent DKEsG-Mono instance is a short-duration task whose CPU consumption is directly proportional to only one parameter, the plasma radius index (i.e. the toroidal flux) of the fusion device considered. Therefore, DKEsG is a good example of how a legacy DRMAA-enabled application can benefit from GWpilot. On the other hand, DKEsG-Mono is a common parameter sweep application with a controlled CPU time variability that can be used to demonstrate GWpilot capabilities.

A real example of calculation with DKEsG is to determine the effective ripple in a fusion device. For this purpose, 24 variations of the standard configuration of the TJ-II [80] device, 72 variations of the plasma collisionallity, and 140 radius indexes have been selected. Therefore, this input parameter combination results in $24 \cdot 72 \cdot 140 = 241,920$ tasks.

To introduce any accurate scheduling policy, DKEsG-Mono execution must be profiled. However, unlike long multiplication, which is entirely based on integer arithmetic, DKEsG-Mono execution time depends also on floating point performance, cache size and other hardware parameters. Thus, latest processors should compute DKEsG-Mono faster. In this case,

Table 7: Average DKEsG-Mono execution times obtained from reference machines for some indexed radius in the standard TJ-II configuration.

| TJ-II index for radius | TJ-II normalised toroidal flux $\rho$ (0-1) | Xeon X5365 3GHz (Launch date: 2007) | Xeon E5620 2.4GHz (Launch date: 2012) |
|---|---|---|---|
| 2 | 7.14E-03 | 58.10 s | 48.33 s |
| 25 | 1.71E-01 | 179.54 s | 150.21 s |
| 48 | 3.36E-01 | 297.33 s | 247.80 s |
| 71 | 5.00E-01 | 348.14 s | 291.28 s |
| 94 | 6.64E-01 | 523.58 s | 439.59 s |
| 117 | 8.29E-01 | 624.92 s | 524.07 s |
| 141 | 1 | 751.05 s | 631.95 s |

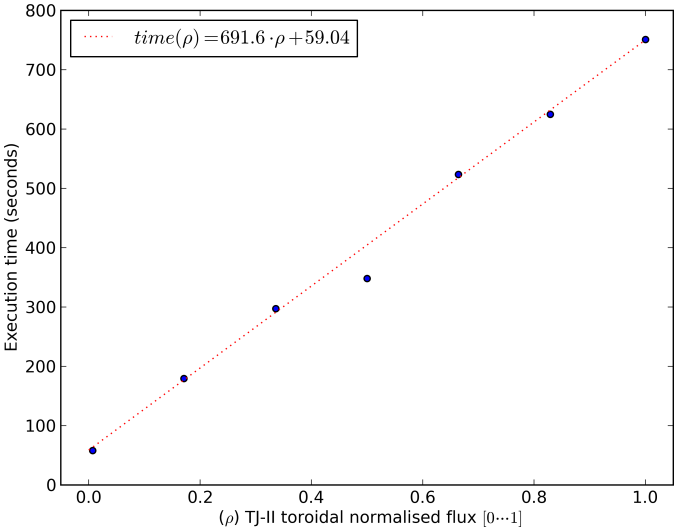



Figure 6: Linear fitting of the values in Table 7 for the he X5365 processor. This is the suggested profile of DKEsG-Mono on that processor.

execution time ranges from 58 to 751 seconds on a Xeon X5365 3GHz, but it is lower on a Xeon E5620 2.4GHz (see the third and fourth columns in Table 7). This issue, far from being a drawback, will be used in this experiment to show how different policies can be enabled for several applications at the same time. These applications will be concurrently managed by GWpilot, resulting in performance improvements for all of them.

### 6.3.2. Customising scheduling at user-level

Now, the objective is to perform a simple customization of the GWpilot scheduling capability based on a personal profiling of the application. This purpose motivates the election of a real application such as DKEsG. As commented in Section 4, advanced policies can be dynamically included in GWpilot by simply including specific pilot tags in the rank expression. For example, if the long multiplication is used in this experiment, only including a ranking expression based on the CPU speed in templates will be enough to obtain a valuable performance improvement:

$$RANK = PILOT\_CPU\_MHZ$$

However, the intention is to go beyond this, showing how users and developers can introduce their personal tags since they are who really know the behaviour of their applications.

Therefore, the $PILOT\_\${GW\_USER}\_VAR\_1$ tag will publish a profiling based on Table 7. As it is shown in Fig. 6, these data are fitted into a polynomial function. Then, the speedup obtained running DKEsG-Mono in any other processor can be expressed by drawing a parallel line above (less performance) or below (better performance) that function. Additionally the speedup for every normalised flux ($\rho$) can be calculated with the following formula:

$$speedup(\rho) = \frac{<wall\ time>}{59} + \rho \cdot 692$$

However, the $PILOT\_\${GW\_USER}\_VAR\_1$ has to be filled by the task whenever it ends its execution. For this purpose, a script that wraps DKEsG-Mono execution at WNs will calculate speedup and publish it with this statement:

```
echo "PILOT_${GW_USER}_VAR_1 = ${speedup}" > ${PILOT_PIPE}
```

On the other hand, the following DRMAA statements must be introduced into the code section of task creation: one to set a ranking policy based on the customised tag and the latter to indicate that task has to be scheduled to a pilot:

```
drmaa_set_vector_attribute(<task description id>,
     DRMAA_GW_RANK, 'PILOT_${GW_USER}_VAR_1')

drmaa_set_vector_attribute(<task description id>,
     DRMAA_GW_REQUIREMENT,'LRMS_NAME="jobmanager-pilot"')
```

Thus, any unskilled developer can perform these actions, and does not need to modify any code of the GWpilot system, including the pilot itself. Moreover, requirement, ranking and profiling expressions could be stored in an auxiliary file which is read when the execution starts (as DKEsG does), making tuning easier to users, but always without modifying GWpilot.

### 6.3.3. Tests proposed

The intention is to show how user policies improve the performance of applications. Nevertheless, as it was mentioned in Section 5, DIANE and DIRAC do not allow setting similar statements for user applications, and only include the possibility of statically configuring the provisioning based on WMS and central GIS. Additionally, to adapt a legacy application implemented in DRMAA to these systems is time-expensive. Therefore, for the sake of comparison with the previous experiments, a GWpilot instance must act as a control test, i.e. supporting DKEsG and long multiplication without the aforementioned ranking customisation and without enabling the GW PiF provisioning guidance.

For this purpose, now three virtual machines with GWpilot are booted. These instances are configured with the static options for GWpilot enumerated in the previous experiments. This assures the comparison with the other systems. Ranking policies are enabled by the user applications running on two virtual machines, and provisioning guidance is also configured in one of these. With this differentiation, the performance gain obtained by each scheduling technique can be measured. Thus, those three deployments are called along this experiment

as control test (*ct*), standalone ranking (*sr*), and guided provisioning (*gp*).

The benefit of guiding provisioning and workload scheduling together, and not simply guiding the latter, is the key achievement that this experiment is trying to demonstrate. All tests will rely on the GIS to find free resources to immediately distribute pilots. However, the user-ranked approach enabled in *sr* and *gp* differentiates the resource provisioning from the task scheduling. Both approaches expect more WNs to be obtained, the best ones of which will be retained if pilots wait a certain time in low overloaded queues. It was commented through the first experiment that application instances (tasks) in the system usually do not reach the maximum number of pilots that can be available in a determined time. This behaviour is especially noticeable with DKEsG, because there is no function in DRMAA 1.0 specification that offers the possibility of returning all job states in a unique call. Then, the execution state of these tasks is sequentially checked in a time interval of 0.1s. However, GWpilot take advantage of this underutilisation when workload policies are enabled. That is, GWpilot achieves an improvement in the quality of resources as a function of time: the best pilots will be selected to run a task and the rest will die when no work is assigned to them.

Besides, *gp* tests finally connects user policies to provisioning. PiF will include the preferences depicted in task descriptions when pilots are submitted (see Subsection 3.4). This scheduling mechanism considers that there is no characteristic that GIS could provide, including the published number of free slots from every resource. That is so because GIS does not assure that these slots are actually available for a specific VO user. However GW PiF can safely use the characterisation of pilots to improve future provisioning, even if this characterisation was customised by the user. Additionally, PiF will use the ranking sentences included in task descriptions, but in a limited way. PiF is configured to only guide half of pilot submissions. That is so to allow GWpilot to better explore the whole infrastructure in order to search for more suitable resources, but assuring the suitability of resources in a 50%. In addition, it is of interest that the influence of these ranking statements in the scheduling mechanism was unbalanced. The intention is to create an experiment close to reality, where there will always be more tasks of some type than other.

Therefore, tests are mainly composed by the DKEsG-Mono tasks needed for the calculation of the effective ripple, but long multiplication is also executed at the same time. The purpose is to use long multiplication as a competitive application in the system and to subsequently analyse its interference on scheduling.

Moreover, to show an effect similar to that found in a competitive multi-application environment (but also controlled for further analysis), the calculation of the effective ripple is split in three parts to be carried out by three different DKEsG instances. Tasks with a lower weight are interspersed along the test. For this purpose, tasks are ordered for later individual submission in a numerical sequence of radius as:

```
Slice 1: (2,47,92;3,48,93;... 16,61,108) + (139,140,141)
Slice 2: (17,62,107;18,62,108;... 31,76,121) + (137,138)
Slice 3: (32,77,122;33,78,123;... 46,91,136)
```

Naturally, three applications will be started at the same time as the long multiplication. Additionally, the radius order assures that the CPU time requirement is increased along the experiment. Every application (long multiplication included) is limited to 250 tasks, and then the maximum number of tasks in the system is 1,000.

Finally, estimating the duration of both executions is necessary. If calculation of the ripple were performed sequentially on the first selected machine described in the previous subsection (see Table 7), it would take approximately 3 years and 45 days; and 36 hours and 30 minutes if 750 cores of that machine were used. On the other hand, long multiplication on 250 cores would take approximately 32 hours if the input set that assures tasks above 10 minutes (< *lower limit* >= 9) were selected.

### *6.3.4. Results*

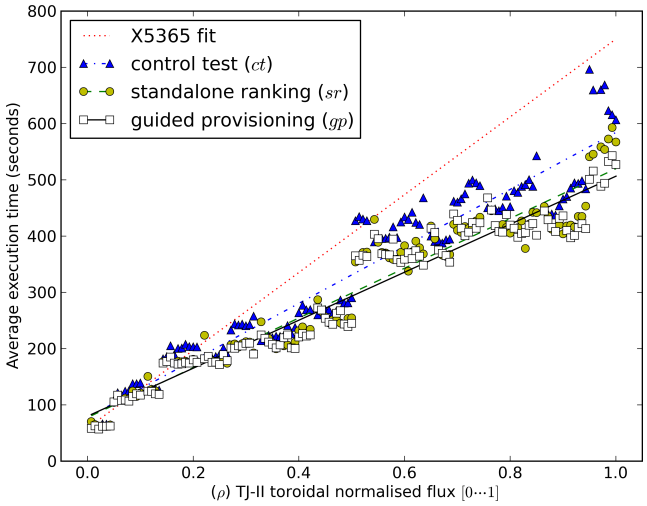
The time spent by the tests is a good introduction to the advantage offered by GWpilot over other solutions, because control test (*ct*) relies on the same configuration parameters used in the experiments described in Subsection 6.2. Now, the experiments are composed of several applications with variable completion times. Additionally, different user policies are applied on long multiplication and DKEsG executions. Consequently, differences between the *ct* and guided provisioning (*gp*) tests ranges from 56 minutes (Test 1) to 8 hours (Test 2). Table 8 shows the makespan of each application and their accumulated spent time.

According to these data, an important reduction (from 11% to 20%) of accumulated spent time in *ct* tests is achieved by *gp* tests. However, standalone ranking (*sr*) tests do not achieve an averaged improvement as it could be expected. This behaviour demonstrates the motivation of this study, i.e. it is necessary the guidance of provisioning because resources finally provided by remote sites could not fit the needs of the applications or they could even arbitrarily fail. GW PiF in *ct* and *sr* tests does not perform any selection of resources where pilots will be submitted with exception of avoiding busy and banned sites. Therefore, when a pilot is discarded, it is arbitrary replaced by a new one. Even, this new pilot can be in same WN than the replaced one if the holding site was not banned. Therefore, the chance to obtain improved resources is based on multiple attempts, but they are better retained in *sr* tests. This circumstance is especially evident in Test 1, where resources with high speedups profiled are appropriated at the beginning of the calculation. Paradoxically, the complete Test 1-(*sr*) was slower than Test 1-(*ct*). That is because pilots more suitable for long multiplication executions (higher CPU speeds) are less retained if they do not entail an improved profile for DKEsG-Mono application.

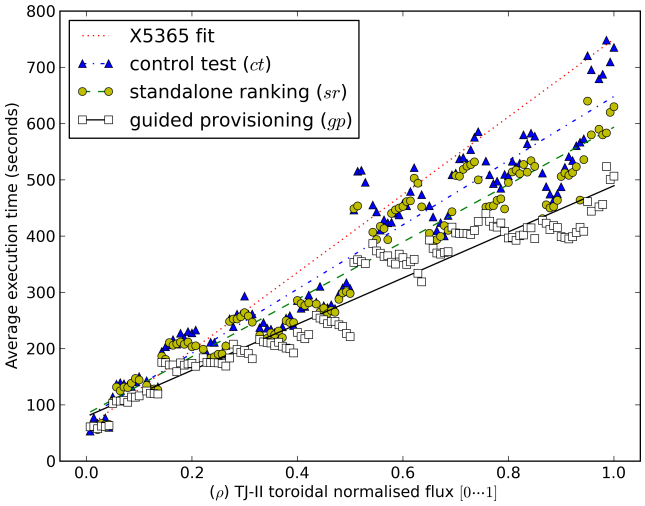
In any case, the algorithm presented in this study devoted to firstly utilise the more powerful pilots in the pool has resulted in a continuous improvement of the resources offered to the user in a ratio proportional to the discarding rate. This fact is supported by the number of tasks per minute considered *done* by the applications, which is increased from the *ct* to *gp*. Nat-

Table 8: Makespan values obtained in second experiment.

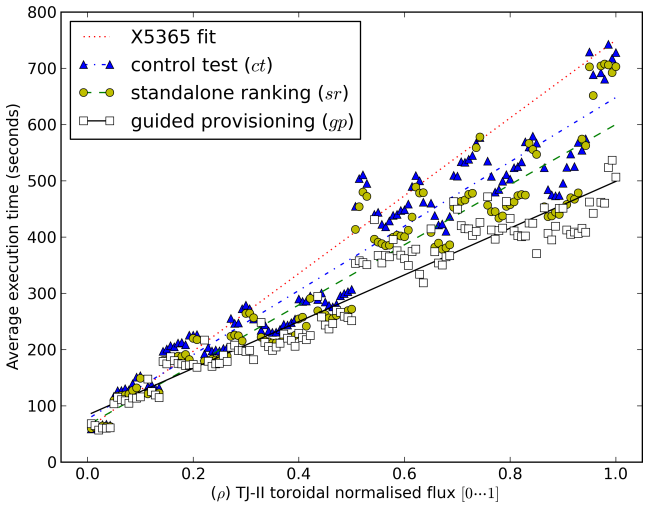
| Test | Type | Makespan | | | | |
|---|---|---|---|---|---|---|
| | | long multiplication | DKEsG-Mono Slice 1 | DKEsG-Mono Slice 2 | DKEsG-Mono Slice 3 | accumulated |
| (1) | *ct* | 1 d: 20 h: 54 m: 23 s | 1 d: 12 h: 09 m: 07 s | 1 d: 19 h: 16 m: 32 s | 1 d: 20 h: 23 m: 50 s | 7 d: 00 h: 43 m: 52 s |
| | *sr* | 1 d: 21 h: 29 m: 12 s | 1 d: 09 h: 41 m: 16 s | 1 d: 12 h: 11 m: 35 s | 1 d: 12 h: 59 m: 43 s | 6 d: 08 h: 21 m: 46 s |
| | *gp* | 1 d: 19 h: 58 m: 07 s | 1 d: 08 h: 36 m: 55 s | 1 d: 11 h: 46 m: 01 s | 1 d: 13 h: 49 m: 22 s | 6 d: 06 h: 10 m: 25 s |
| (2) | *ct* | 1 d: 21 h: 56 m: 13 s | 1 d: 19 h: 50 m: 51 s | 1 d: 21 h: 07 m: 36 s | 1 d: 21 h: 43 m: 34 s | 7 d: 12 h: 38 m: 14 s |
| | *sr* | 1 d: 18 h: 08 m: 21 s | 1 d: 17 h: 32 m: 11 s | 1 d: 18 h: 54 m: 07 s | 1 d: 16 h: 40 m: 27 s | 6 d: 23 h: 15 m: 06 s |
| | *gp* | 1 d: 13 h: 51 m: 20 s | 1 d: 09 h: 18 m: 16 s | 1 d: 10 h: 56 m: 42 s | 1 d: 12 h: 01 m: 03 s | 5 d: 22 h: 07 m: 21 s |
| (3) | *ct* | 1 d: 21 h: 34 m: 02 s | 1 d: 18 h: 23 m: 14 s | 1 d: 20 h: 56 m: 40 s | 1 d: 21 h: 29 m: 42 s | 7 d: 10 h: 23 m: 38 s |
| | *sr* | 1 d: 20 h: 23 m: 52 s | 1 d: 17 h: 47 m: 48 s | 1 d: 20 h: 08 m: 49 s | 1 d: 20 h: 03 m: 18 s | 7 d: 06 h: 23 m: 47 s |
| | *gp* | 1 d: 15 h: 49 m: 48 s | 1 d: 08 h: 17 m: 00 s | 1 d: 10 h: 51 m: 06 s | 1 d: 11 h: 46 m: 44 s | 5 d: 22 h: 44 m: 38 s |



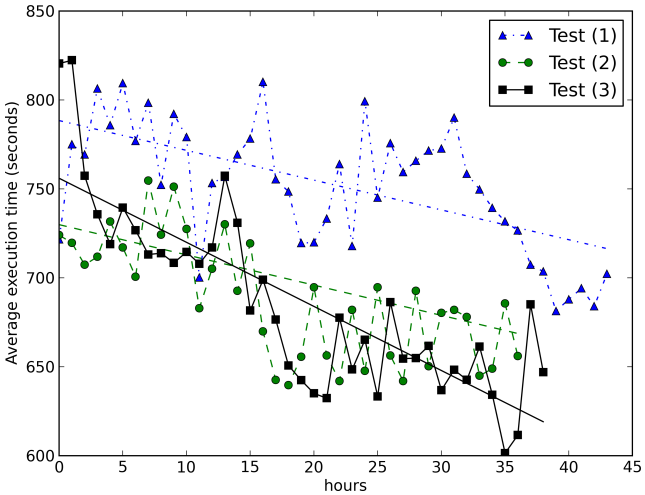

(a) Test 1. Execution time average of DKEsG-Mono per normalised flux ($\rho$).




(b) Test 2. Execution time average of DKEsG-Mono per normalised flux ($\rho$).




(c) Test 3. Execution time average of DKEsG-Mono per normalised flux ($\rho$).




(d) Task execution time average per hour (in seconds) of long multiplication in the three guided provisioning ($gp$) tests (with $n = 90,000$).

Figure 7: Average times obtained in second experiment

urally, values for both rates written in Table 9 were registered when the system maintain the four application running to assure accurateness.

To illustrate how the continuous improvement of resources impacts on the accomplishment of tasks, averaged execution times are shown in Figures 7. However, to differentiate the influence on every application, two types of graphs are included. First, the execution time average of DKEsG-Mono tasks per normalised flux are depicted in Figures 7-(a, b, c). To compare speedup obtained, the data belonging to every test are fitted into

Table 9: Results obtained in second experiment. Values for *tasks done per minute*, *pilots discarded per minute* and the *number of calls per second of every pilot* are measured after the system had enrolled at least 800 pilots and before the last 1000 tasks were remaining to be representative of the usual behavior of GWpilot.

| Test | Type | Tasks | | | Pilots | | | |
|---|---|---|---|---|---|---|---|---|
| | | failed | done/m | max. done/m | submitted | enrolled | discarded/m | calls/s |
| (1) | *ct* | 5140 | 104.41 | 241 | 30882 | 9107 | 2.95 | 62.09 |
| | *sr* | 5307 | 122.03 | 271 | 24844 | 5973 | 1.96 | 64.78 |
| | *gp* | 4625 | 124.96 | 268 | 22631 | 6049 | 2.05 | 67.05 |
| (2) | *ct* | 6322 | 100.05 | 238 | 27421 | 8134 | 2.59 | 65.76 |
| | *sr* | 4962 | 121.30 | 254 | 26041 | 7080 | 2.53 | 67.31 |
| | *gp* | 8663 | 132.72 | 294 | 27631 | 9049 | 3.01 | 66.84 |
| (3) | *ct* | 4718 | 89.68 | 216 | 21572 | 6227 | 2.36 | 59.20 |
| | *sr* | 8726 | 91.82 | 231 | 13295 | 6698 | 2.37 | 61.26 |
| | *gp* | 3954 | 126.05 | 276 | 15497 | 7055 | 2.88 | 68.05 |

a polynomial function. Additionally, the line representing the performance of reference machine is drawn. A supplementary fourth Figure 7-(d) shows the evolution of the execution times of long multiplication for $n = 90,000$ through the *gp* tests. This latter graph is included only to demonstrate that despite of pilots are preferably submitted to sites with resources suitable for DKEsG, the ranking approach in provisioning also assure an improvement for long multiplication. Thus, this approach achieves a progressive reduction of the average CPU consumed by their consecutively submitted tasks. These conclusions are supported by the makespan obtained by every application and compiled in Table 8. In particular, the calculation time of every DKEsG slice is reduced from 11.2% to 25% and the long multiplication from 3.4% to 13.4% in *gp* tests.

Nevertheless, approximately 25% of the discarded pilots are consequences of grid job errors, so the improvement translated to a reduction in the makespan is smaller. Additionally, the number of failed tasks in the pilots' execution deserves a deeper explanation. Although GWpilot is more stable than the traditional early-binding methods, it cannot avoid errors produced by the middleware, network cuts or any other typical problem related to the remote execution of pilots. This is the case for the failed executions shown in Table 9, which are primarily the consequence of a pilot dying while a task was running in it or a task being dispatched to a dead pilot that has not been discarded yet. However, they represent less than 4% of the total submitted executions in every test, while the aforementioned percentage of failed pilots rises to 75% (note that many pilot jobs were discarded before the 30-minute threshold set in the PiF configuration). All of these values demonstrate not only the suitability of GWpilot from the user point of view but also the convenience of splitting resource provisioning from task scheduling.

### *6.4. Performance evaluation*

In the following subsections, other results obtained in this work will be discussed. Even though the comparison between the times spent by the tests is the most visible performance measurement for the user, it does not completely describe all the important performance aspects of the GWpilot system. Therefore, an analysis is made using different approaches that are useful to validate the design affirmations commented in Section 3.

#### *6.4.1. Scalability*

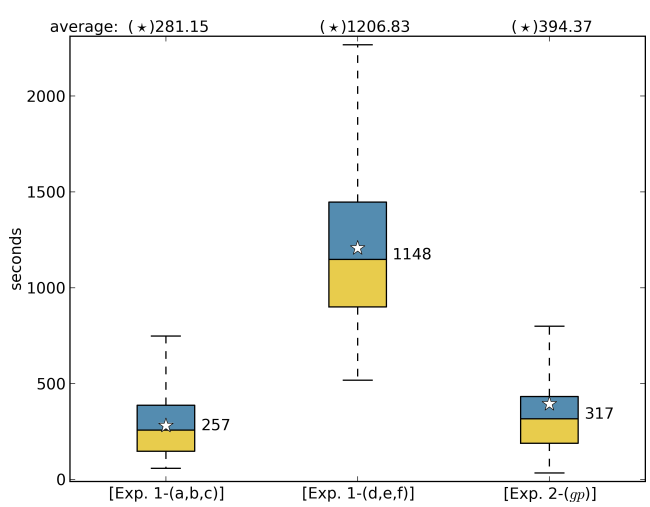
It is clear that scalability only can be demonstrated increasing experimentally the load of system. However, the statistics obtained through the experiments performed in this work can be extrapolated to other magnitudes, or even already show performance values comparable to the obtained by other systems currently in production in large collaborations.

Firstly, the total number of processed user tasks in the experiments is an achievement if we have in mind that other systems consider these volumes [45, 57, 81] as the optimum throughput for months. In relation to provisioning, the number of effectively enrolled pilots presented in Tables 6 and 9 is also important. There were tests with more than 9,000 successfully enrolled pilots [44], achieving maxima of above 200 pilots per minute [52]. However, it is interesting to provide the average number of enrolled pilots and the percentage of discarded pilots per hour [81], which are between 200-300 and above 5%, respectively. Note that a discarded pilot does not imply that its hosting grid job has failed and vice versa. In any case, these values also show how the enrolment and discarding processes in GWpilot are faster and more effective than the mechanisms to provision resources used in other systems [45] which are based on middleware CLI.

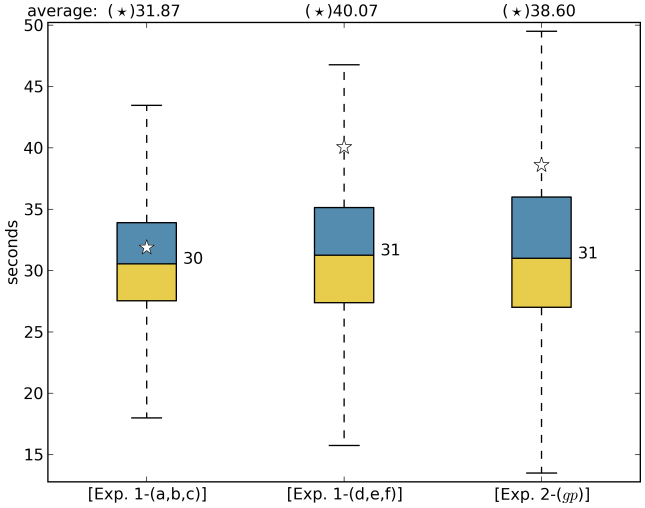
The dispatch capacity is another productivity and scalability aspect that must be examined. In the experiments, the time spent by Scheduler was maintained below one second. Then GWpilot can effectively dispatch more than 500 tasks per minute. Even when other systems usually provide equivalent or lower rates, is due to other performance metrics should be improved in the specific areas for which they were developed. In any case, it is noteworthy that this dispatching aspect is indicative of the future scalability of GWpilot and its suitability for a wide range of calculations. In addition, Table 9 compile the processing rate of pilot operations, whose values are easily manageable by any HTTP server (PiS is based on a lightweight HTTP famework). Therefore, the number of pilots can be potentially increased to greater orders of magnitude.

#### *6.4.2. Turnaround and controlled overhead*

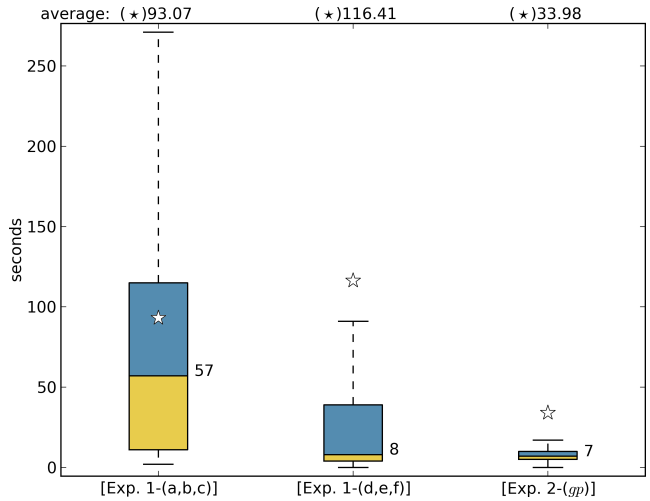
Other metrics should be considered to validate the design and implementation of GWpilot. In particular, the middleware overhead introduced by the system constraints its future scalability



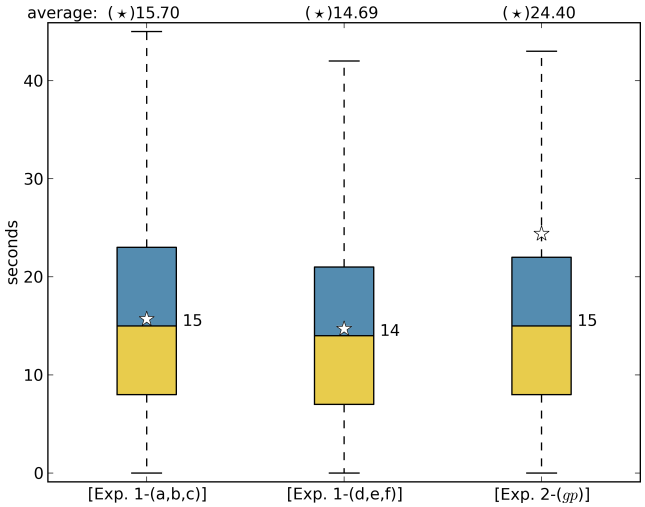

(a) Real execution time of tasks at remote resources.




(b) Pilot overhead and transfer times.




(c) Scheduling overhead, i.e. time spent by GridWay in dispatching a task whenever it is included in the system (it depends on the number of pilots up).




(d) Time spent in generating and monitoring tasks for the applications (it is included even when it is not properly part of the task turnaround).

Figure 8: Box plots of the different turnaround times obtained with every completed task grouped by experiment. Every box stand for: [Exp. 1-(a, b, c)], i.e. the 3 short tests (a, b, c) of long multiplication corresponding to fist experiment; [Exp. 1-(d, e, f)], i.e. the 3 long tests (d, e, f) of long multiplication corresponding to first experiment; [Exp. 2-($gp$)], i.e. the 3 tests with guided provisioning ($gp$). Stars represent the average.

and capacity of hosting complex algorithms, as has been mentioned in Subsection 2.6 and through Sections 3 and 4 . It is desirable that overheads were predictable and even configurable according to the platform and infrastructure on which the pilot system is running. The whole overhead of the system is finally translated to user as higher turnaround times. Therefore, managing turnaround overheads is of main interest and should be measured.

The real turnaround time of each completed task is defined as the time difference between the moment when the task is queued in the UTQ (the GridWay Job Pool) for being dispatched and the moment when the system notifies that it is completed. This value represents the CPU time consumed by the application when it is executed on remote resources plus the overhead introduced by the middleware. In general, for the GWpilot system, the middleware overhead can be decomposed in two different types based on their source: the internal operations necessary to accept, classify, dispatch and consider the task as ended; and the pilot operations necessary to accomplish the task.

Figures 8-(a, b, c) show three turnaround time values in GWpilot tests. They describe the turnaround of every task *done* processed by GWpilot in this work. Tasks are grouped by experiment and subsequently decomposed in the three categories: CPU time, scheduling (internal), and pilot overhead.

Time wasted by internal operations is negligible for the tests performed in this study. This fact is supported by the measurements made through last subsections. Nevertheless, the turnaround increases when the Scheduler is unable to dispatch tasks due to the inexistence of idle pilots. Additionally, other experiments composed of higher volumes of tasks with complex requirements could increase this time. For this reason,

it was called scheduling overhead, although all internal operations are included.

Pilot overhead is produced by the pulling interval performed by pilots and the transfer times. Usually, staging files is considered as part of the internal operations of the system [33] or even a separate overhead in other works [32]. In this study transfer times are included in pilot overhead because the pilot downloads the needed inputs and uploads the outputs generated by a task.

Although it is neither concerned with the turnaround measurement nor with the GWpilot performance, the application overhead is also displayed in Fig 8-(d) because it influences the final makespan of the calculation if there are not enough tasks provided to fill all the available pilots. This circumstance was described in Subsection 6.2.2, where a brief explanation was provided. Now, box plots indicate that the script that wraps the long multiplication application gives the similar performance when it supplies very short (< *lower limit* >= 3) or short (< *lower limit* >= 9) tasks. However, this overhead, along with the one originated in pilots (Fig. 8-(b)) and the configured *SCHEDULING_INTERVAL*, results in an important percentage with respect to CPU time in the first experiments. Therefore, *running* pilots are not completely filled.

CPU time plotted in Fig. 8-(a) shows similar values for short experiments and the ones that perform together DKEsG and long multiplication calculations. The explanation is that the volume of multiplication tasks only represents approximately 11% of all tasks in these experiments. Additionally, DKEsG-Mono tasks are shorter than very short multiplication tasks (< *lower limit* >= 3). Thus, the question is translated to the scheduling overhead shown in Fig. 8-(c). The maximum number of usable pilots is limited to the same maximum quantity of tasks in every experiment. An increase in the scheduling overhead indicates that either the number of *running* pilots is lower than 1,000, or some of these pilots are banned. Both circumstances explain the high values compiled for the short experiments and how they decrease when the duration of experiments is larger up to a median of 7 s (note that the *SCHEDULING_INTERVAL* is set to 10 seconds in every experiment). Thus, the scheduling overhead is actually describing the cost of provisioning resources, but remains statistically stable if the number of tasks is adjusted to the number of *running* pilots. This is precisely one of the main performance measurements that should be analysed in this work: the variability of this overhead should not be taken into account by a self-scheduler application that will use GWpilot framework.

The other important aspect is the pilot overhead, which medians are always close to 30 seconds because it mainly depends on its configured pulling interval. Nevertheless, transferring files have influence on this overhead because the average values are higher in the experiments that deal with greater output sizes. This fact demonstrates the scalability of the GWpilot system, which is not influenced by the number of pilots running.

It is important to mention how the arithmetic average values of the corresponding measured parameters behave. In the figure describing CPU time (Fig. 8-(a)), it can be seen almost a symmetric box with a higher weight in the distribution of the higher values, what can be appreciated in the average values (stars) and in the corresponding longer whiskers. Unlike the scheduling and pilot overheads, where the averages lie outside the third quartile in the long first (< *lower limit* >= 9) and *gp* tests and even are outliers in the scheduling overhead, i.e. there have been some specific values of these overheads which deeply exceed the median and produce a non-symmetric average. This fact is not of importance if the median is symmetric inside the box, since it means that the distribution is almost homogenous and only some few tasks have produced a major overhead (pilot overhead case). However, it is noticeable an asymmetric distribution in the long first (< *lower limit* >= 9) and *gp* boxes of the scheduling overhead . In these cases, not only the average is an outlier, but also the lower values of the overhead are much concentrated around a value, i.e. most of the tasks (close to 50%) have a low scheduling overhead.

As a summary, measurements of pilot and scheduler overheads enable self-schedulers to properly calculate their task grouping regarding only the application characteristics, while the overhead introduced by GWpilot remains stable and similar to its configuration parameters.

## 7. Conclusions and future work

Current pilot systems are accomplishing great results in the area for which they have been implemented and designed. However, they still have some limitations that prevent their deployment and application to other fields or codes and they are not exploiting all of the advantages that pilot jobs offer.

In this work, the GWpilot system is proposed as a new solution that accomplishes the requirements of a wide range of users and institutions and profits from advanced scheduling techniques. GWpilot makes the use of pilot jobs automatic and unattended both to users and developers, while project managers benefit from its fair-share policies to give a boost to their priority computational challenges. Furthermore, GWpilot enables a simplified multi-level scheduling model that is even suitable for improving the behaviour of self-schedulers. Its design makes GWpilot:

- Easy-to-install and standalone.
- Compatible with previously ported applications.
- Interoperable with diverse grid and cloud infrastructures and stackable with other systems.
- Lightweight and scalable.
- Highly adjustable, supporting the customisation of scheduling at several levels.

To demonstrate the capacities described, main performance aspects have been measured in real tests that reproduce a non-ideal scenario for a scheduler that works on distributed resources in production. These measurements are focused not only on demonstrating the improvements achieved from the user point of view but also to show the new available functionalities for the highly skilled developers or administrators, which

lead to a better performance and a customisable approach for different codes and disciplines.

As future work, the integration with gLExec for fulfilling current infrastructure policies related to multi-user pilot jobs will be performed. Additionally, the utilisation of diverse complex algorithms to accomplish the requirements of specific workload problems is being explored. Finally, the incorporation of cloud resources into the framework is being evaluated considering the economical questions that arise in an environment with multiple commercial providers.

## Acknowledgements

This work has been supported by the Behavioural Types for Reliable Large-Scale Software Systems (BETTY, ICT COST Action IC1201), by the European Grid Infrastructure - Integrated Sustainable Pan-European Infrastructure for Researchers in Europe (EGI-InSPIRE, Grant RI-261323), by European Fusion Development Agreement (EFDA-WP12-DAS07, http://www.efda.org), co-funded by the European Commission within its 7th Framework Programme, and by Ministerio de Economía y Competitividad through Service-Cloud research grant TIN2012-31518.

## References

[1] I. Foster, What Is the Grid? A Three Point Checklist, GRID-today 1 (6). URL `http://www.gridtoday.com`

[2] M. Garey, D. Johnson, Computers and Intractibility: A guide to the Theory of NP-completeness, W. H. Freeman&Co, New York, 1979.

[3] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, Grid information services for distributed resource sharing, in: $10^{th}$ IEEE International Symposium on High Performance Distributed Computing (HPDC-10), IEEE CS Press, San Francisco, California, 2001, pp. 181–194.

[4] S. Andreozzi, S. Burke, F. Ehm, L. Field, G. Galang, B. Konya, et al., GLUE Specification v. 2.0, GFD 147 (March 2009). URL `http://www.ogf.org/documents/GFD.147.pdf`

[5] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, Wiley- Interscience, 1991.

[6] M. J. Litzkow, M. Livny, M. W. Mutka, Condor: A Hunter of Idle Workstations, in: $8^{th}$ International Conference on Distributed Computing Systems, IEEE CS Press, San José, California, 1988, pp. 104–111.

[7] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke, Condor-G : A Computation Management Agent for Multi-Institutional Grids, Cluster Computing 5 (3) (2002) 237–246.

[8] P. Andreetto, S. Andreozzi, G. Avellino, S. Beco, A. Cavallini, M. Cecchi, et al., The gLite workload management system, Journal of Physics : Conference Series 119 (6) (2008) 062007.

[9] E. Huedo, R. S. Montero, I. M. Llorente, A modular meta-scheduling architecture for interfacing with pre-WS and WS Grid resource management services, Future Generation Computer Systems 23 (2) (2007) 252–261.

[10] E. Huedo, R. S. Montero, I. M. Llorente, Experiences on Grid Resource Selection Considering Resource Proximity, in: Grid Computing – First European Across Grids Conference, Vol. 2970 of Lecture Notes in Computer Science, Santiago de Compostela, Spain, 2004, pp. 1–8.

[11] C. Vázquez, E. Huedo, R. S. Montero, I. M. Llorente, Federation of TeraGrid, EGEE and OSG infrastructures through a metascheduler, Future Generation Computer Systems 26 (7) (2010) 979–985.

[12] I. Marín, E. Huedo, I. M. Llorente, Interoperating Grid Infrastructures with the GridWay Metascheduler, Concurrency and Computation: Practice and Experience in press.

[13] B. B. P. Rao, S. Ramakrishnan, M. R. R. Gopalan, C. Subrata, N. Mangala, R. Sridharan, e-Infrastructures in IT: A case study on Indian national grid computing initiative – GARUDA, Computer Science - Research and Development 23 (3–4) (2009) 283–290.

[14] T. Glatard, S. Camarasu-Pop, A model of pilot-job resource provisioning on production grids, Parallel Computing 37 (10–11) (2011) 684–692.

[15] R. M. Piro, A. Guarise, G. Patania, A. Werbrouck, Using historical accounting information to predict the resource usage of grid jobs, Future Generation Computer Systems 25 (5) (2009) 499–510.

[16] Z. Yu, Toward Practical multi-workflow Scheduling in Cluster and Grid Environments, Ph.D. thesis, Wayne State Unv., Detroit, USA (2009).

[17] J. Yu, R. Buyya, A Taxonomy of Workflow Management Systems for Grid Computing, Journal of Grid Computing 3 (2006) 171–200.

[18] D. Spiga, CMS workload management, Nuclear Physics B - Proceedings Supplements 172 (2007) 141–144.

[19] A. Tsaregorodtsev, M. Bargiotti, N. Brook, A. C. Ramo, G. Castellani, P. Charpentier, et al., DIRAC: A Community Grid Solution, Journal Physics : Conference Series 119 (6) (2008) 062048.

[20] P. Saiz, L. Aphecetche, P. Bunčić, R. Piskać, J. E. Revsbech, V. Šego, AliEn–ALICE environment on the GRID, Nuclear Instruments and Methods in Physics Research A 502 (2003) 437–440.

[21] P. Nilsson, J. Caballero, K. De, T. Maeno, M. Potekhin, T. Wenaus, The PanDA System in the ATLAS Experiment, in: XII Advanced Computing and Analysis Techniques in Physics Research (ACAT'08), SISSA PoS, Erice, Italy, 2008, pp. 27–1–27–8.

[22] J. Díaz, S. Reyes, A. Niño, C. Muñoz-Caro, Derivation of self-scheduling algorithms for heterogeneous distributed computer systems: Application to internet-based grids of computers, Future Generation Computer Systems 25 (6) (2009) 617–626.

[23] F. Xhafa, A. Abraham, Computational models and heuristic methods for Grid scheduling problems, Future Generation Computer Systems 26 (4) (2010) 608–621.

[24] Y. Gao, H. Rong, J. Z. Huang, Adaptive grid job scheduling with genetic algorithms, Future Generation Computer Systems 21 (1) (2005) 151–161.

[25] J. T. Mościcki, Distributed analysis environment for HEP and interdisciplinary applications, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 502 (2–3) (2003) 426–429.

[26] I. Sfiligoi, glideinWMS - A generic pilot-based Workload Management System, Journal of Physics: Conference Series 119 (2008) 062044.

[27] P. Buncic, J. F. Grosse-Oetringhaus, A. Peters, P. Saiz, The Architecture of the AliEn System, in: Computing in High Energy and Nuclear Physics conference (CHEP'04), A. Aimar et al. CERN, Geneva 2005. CERN-2005-02, Interlaken, Switzerland, 2004, pp. 951–954.

[28] A. Luckow, L. Lacinski, S. Jha, Saga big job: an extensible and interop erable pilot-job abstraction for distributed applications and systems, in: $10^{th}$ IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid), IEEE CS Press, Melbourne, Australia, 2010, pp. 135–144.

[29] E. Urbah, P. Kacsuk, Z. Farkas, G. Fedak, G. Kecskemeti, O. Lodygensky, et al., EDGeS: Bridging EGEE to BOINC and XtremWeb, Journal of Grid Computing 7 (3) (2009) 335–354.

[30] M. Silberstein, A. Sharov, D. Geiger, A. Schuster, GridBot: execution of bags of tasks in multiple grids, in: Conference on High Performance Computing Networking, Storage and Analysis (SC'09), ACM New York, Portland, Oregon, USA, 2009, pp. 1–12.

[31] M. L. Pinedo, Scheduling: Theory, Algorithms, and Systems, $4^{th}$ Edition, Springer, 2012.

[32] A. Luckow, M. Santcroos, A. Merzky, O. Weidner, P. Mantha, S. Jha, P*: A model of pilot-abstractions, in: $8^{th}$ IEEE International Conference on E-Science (e-Science 2012), Chicago, USA, 2012, pp. 1–10.

[33] J. T. Mościcki, M. Lamannaa, M. Bubak, P. M. A. Sloot, Processing moldable tasks on the Grid: Late job binding with lightweight user-level overlay, Future Generation Computer Systems 27 (6) (2011) 725–736.

[34] D. Groep, O. Koeroo, G. Venekamp, gLExec: gluing Grid computing to the Unix world,, Journal of Physics : Conference Series 119 (6) (2008) 062032.

[35] I. Sfiligoi, CDF computing, Computer Physics Communications 177 (1–2) (2007) 235–238.

[36] E. Deelman, G. Singh, M. H. Su, J. Blythe, Y. Gil, C. Kesselman, et al., Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Scientific Programming 13 (3) (2005) 219–237.

[37] G. Juve, E. Deelman, K. Vahi, G. Mehta, Experiences with resource provisioning for scientific workflows using Corral, Scientific Programming 18 (2) (2010) 77–92.

[38] M. Altunay, P. Avery, K. Blackburn, B. Bockelman, M. Ernst, D. Fraser, et al., A Science Driven Production Cyberinfrastructure – the Open Science Grid, Journal of Grid Computing 9 (2) (2011) 201–218.

[39] E. Walker, J. P. Gardner, V. Litvin, E. L. Turner, Personal adaptive clusters as containers for scientific jobs, Cluster Computing 10 (3) (2007) 339–350.

[40] C. Vázquez, E. Huedo, R. S. Montero, I. M. Llorente, On the use of clouds for grid resource provisioning, Future Generation Computer Systems 27 (5) (2011) 600–605.

[41] S. Camarasu-Pop, T. Glatard, R. F. da Silva, P. Gueth, D. Sarrut, H. Benoit-Cattin, Monte Carlo simulation on heterogeneous distributed systems: A computing framework with parallel merging and checkpointing strategies, Future Generation Computer Systems 29 (3) (2013) 728–738.

[42] A. Tsaregorodtsev, V. Garonne, I. Stokes-Rees, DIRAC: A scalable lightweight architecture for high throughput computing, in: $5^{th}$ IEEE/ACM International Workshop on Grid Computing (GRID'04), IEEE CS Press, Pittsburgh, USA, 2004, pp. 19–25.

[43] G. Castellani, R. Santinelli, Job Prioritization and Fair Share in the LHCb experiment, Journal of Physics : Conference Series 119 (7) (2008) 072009.

[44] X. Zhao, J. Hover, T. Wlodek, T. Wenaus, J. Frey, T. Tannenbaum, M. Livny, PanDA Pilot Submission using Condor-G: Experience and Improvements, Journal of Physics : Conference Series 331 (7) (2011) 072069.

[45] S. Bagnasco, L. Betev, P. Buncic, F. Carminati, F. Furano, A. Grigoras, et al., The ALICE Workload Management System: Status before the real data taking, Journal of Physics : Conference Series 219 (6) (2010) 062004.

[46] E. Carona, V. Garonne, A. Tsaregorodtsev, Definition, modelling and simulation of a Grid computing scheduling system for high throughput computing, Future Generation Computer Systems 23 (8) (2007) 968–976.

[47] S. K. Paterson, A. Tsaregorodtsev, DIRAC Optimized Workload Management, Journal of Physics : Conference Series 119 (6) (2008) 062040.

[48] P. Nilsson, Experience from a pilot based system for ATLAS, Journal of Physics : Conference Series 119 (6) (2008) 062038.

[49] P.-H. Chiu, M. Potekhin, Pilot factory – a Condor-based system for scalable Pilot Job generation in the Panda WMS framework, Journal of Physics: Conference Series 219 (6) (2010) 062041.

[50] I. Stokes-Rees, A. Tsaregorodtsev, V. Garonne, DIRAC Lightweight Information and Monitoring Services using XML-RPC and Instant Messaging, in: Computing in High Energy and Nuclear Physics conference (CHEP'04), A. Aimar et al. CERN, Geneva 2005. CERN-2005-02., Interlaken, Switzerland, 2004, pp. 788–791.

[51] J. T. Mościcki, F. Brochu, J. Ebke, U. Egede, J. Elmsheuser, K. Harrison, et al., GANGA: A tool for computational-task management and easy access to Grid resources, Computer Physics Communications 180 (11) (2009) 2303–2316.

[52] D. Bradley, O. Gutsche, K. Hahn, B. Holzman, S. Padhi, H. Pi, et al., Use of glide-ins in CMS for production and analysis, Journal of Physics : Conference Series 219 (7) (2010) 072013.

[53] G. Codispoti, C. Mattia, A. Fanfani, F. Fanzago, F. Farina, C. Kavka, et al., CRAB: A CMS application for distributed analysis, IEEE Transactions Nuclear Science 56 (5) (2009) 2850–2858.

[54] D. Evans, A. Fanfani, C. Kavka, F. van Lingen, G. Eulisse, W. Bacchi, et al., The CMS Monte Carlo Production System: Development and Design, Nuclear Physics B - Proceedings Supplements 177-178 (2008) 285–286.

[55] M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, et al., Advanced Resource Connector middleware for lightweight computational Grids, Future Generation Computer Systems 23 (2) (2007) 219–240.

[56] A. Fanfani, A. Afaq, J. A. Sanches, J. Andreeva, G. Bagliesi, L. Bauerdick, , et al., Distributed Analysis in CMS, Journal of Grid Computing 8 (2) (2010) 159–179.

[57] M. Cinquilli, A. F. G. Codispoti, F. Fanzago, F. Farina, J. Hernández, et al., Tools to use heterogeneous Grid schedulers and storage system, in: $13^{th}$ International Workshop on Advanced Computing and Analysis Techniques in Physics Research, (ACAT2010 ), SISSA PoS, Jaipur, India, 2010, pp. 29–1–29–13.

[58] G. Shao, Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources, Ph.D. thesis, University of Califorina, San Diego, USA (2001).

[59] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, et al., Adaptive Computing on the Grid Using AppLeS, IEEE Transactions on Parallel and Distributed Systems 14 (4) (2003) 369–382.

[60] J.-P. Goux, S. Kulkarni, M. Yoder, J. Linderoth., Master–Worker: An Enabling Framework for Applications on the Computational Grid, Cluster Computing 4 (2001) 63–70.

[61] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, H. Casanova, A GridRPC Model and API for End-User Applications, GFD-R.052 (June 29 2007).
URL `http://www.ogf.org/documents/GFD.52.pdf`

[62] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, S. Matsuoka, Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing, Journal of Grid Computing 1 (2003) 41–51.

[63] H. Rajic, R. Borbst, W. Chan, F. Fersti, J. Gardiner, A. Haas, et al., Distributed Resource Management Application API Specification 1.0, GFD 133 (June 2008).
URL `http://www.ogf.org/documents/GFD.133.pdf`

[64] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. V. Laszewski, et al., SAGA: A simple API for Grid Applications. High-Level Application Programming on the Grid, Computational Methods in Science and Technology 12 (1) (2006) 7–20.

[65] J. T. Mościcki, Understanding and Mastering Dynamics in Computing Grids: Processing Moldable Tasks with User-Level Overlay, Ph.D. thesis, Universiteit van Amsterdam, Nederland (2011).

[66] V. V. Korkhov, V. V. Krzhizhanovskaya, P. M. A. Sloot, A Grid-Based Virtual Reactor: Parallel performance and adaptive load balancing, Journal of Parallel and Distributed Computing 68 (5) (2008) 596–608.

[67] V. V. Korkhov, J. T. Mościcki, V. V. Krzhizhanovskaya, Dynamic workload balancing of parallel applications with user-level scheduling on the Grid, Future Generation Computer Systems 25 (1) (2009) 28–34.

[68] D. P. Anderson., BOINC: A System for Public-Resource Computing and Storage, in: $5^{th}$ IEEE/ACM Int. Workshop on Grid Computing (GRID'04), IEEE CS Press, Pittsburgh, USA, 2004, pp. 4–10.

[69] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, O. Lodygensky, Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with Grid, Future Generation Computer Systems 21 (3) (2005) 417–437.

[70] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, M. Wilde, Falkon: a Fast and Light-weight tasK executiON framework, in: ACM/IEEE Conference on Supercomputing (SC'07), ACM New York, Reno, Nevada, USA, 2007, pp. 1–12.

[71] E. Huedo, R. S. Montero, I. M. Llorente, Grid Architecture from a Metascheduling Perspective, Computer 43 (7) (2010) 51–56.

[72] A. S. McGough, W. Lee, S. Das, A standards based approach to enabling legacy applications on the Grid, Future Generation Computer Systems 24 (7) (2008) 731–743.

[73] D. Kelsey, Policy on Grid Multi-User Pilot Jobs. EGI-SPG-PilotJobs-V1_0, EGI Document 84-v6 (2010).
URL `https://documents.egi.eu/document/84`

[74] J. L. Vázquez-Poletti, E. Huedo, R. S. Montero, I. M. Llorente, A Comparison Between two Grid Scheduling Philosophies: EGEE WMS and GridWay, Multiagent and Grid Systems 3 (4) (2007) 429–439.

[75] M. Rodríguez-Pascual, A. J. Rubio-Montero, R. Mayo, A. Bustos, F. Castejón, I. M. Llorente, More Efficient Executions of Monte Carlo Fusion Codes by Means of Montera: The ISDEP Use Case, in: $18^{th}$ Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2011), IEEE CS Press, Ayia Napa, Cyprus, 2011, pp. 380–384.

[76] C. Germain-Renaud, C. Loomis, J. T. Mościcki, R. Texier, Scheduling for Responsive Grids, Journal of Grid Computing 6 (1) (2008) 15–27.

[77] F. Castejón, A. Gómez-Iglesias, A. Bustos, L. Campos, E. Cappa, M. Cárdenas-Montes, et al., Grid Computing for Fusion Research, in: 3rd Iberian Grid Infrastructure Conference, Vol. 3, NETBIBLO S. L. (Sta Cristina, La Coruña), 2009, pp. 291– 302.

[78] M. Rodríguez-Pascual, A. Gómez, R. Mayo-García, D. P. de Lara, E. M.

González, A. J. Rubio-Montero, J. L. Vicent, Superconducting Vortex Lattice Configurations on Periodic Potentials: Simulation and Experiment, Superconductivity and Novell Magnetism 25 (7) (2012) 2127–2130.

[79] A. J. Rubio-Montero, F. Castejón, M. A. Rodríguez-Pascual, E. Montes, R. Mayo, Drift Kinetic Equation Solver for Grid (DKEsG), IEEE Transactions on Plasma Science 38 (9) (2010) 2093–2101.

[80] C. Alejaldre, J. J. Alonso, J. Botija, F. Castejón, J. R. Cepero, J. Díaz, et al., TJ-II project: A flexible Heliac stellarator, Fusion Technology 17 (1) (1990) 131–139.

[81] J. T. Mościcki, M. Woś, M. Lamanna, P. de Forcrand, O. Philipsen, Lattice QCD thermodynamics on the Grid, Computer Physics Communications 181 (10) (2010) 1715–1726.