Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

GANGA: A tool for computational-task management and easy access to Grid resources ☆

J.T. Mościcki [f,*], F. Brochu [a], J. Ebke [c], U. Egede [b], J. Elmsheuser [c], K. Harrison [a], R.W.L. Jones [d], H.C. Lee [e,1], D. Liko [f], A. Maier [f], A. Muraru [f], G.N. Patrick [g], K. Pajchel [j], W. Reece [b], B.H. Samset [j], M.W. Slater [i], A. Soroko [h], C.L. Tan [i], D.C. van der Ster [f], M. Williams [b]

[a] University of Cambridge, Cambridge, United Kingdom
[b] Imperial College London, London, United Kingdom
[c] Ludwig-Maximilians-Universität, Munich, Germany
[d] Lancaster University, Lancaster, United Kingdom
[e] NIKHEF, Amsterdam, The Netherlands
[f] CERN, Geneva, Switzerland
[g] STFC Rutherford Appleton Laboratory, Didcot, United Kingdom
[h] University of Oxford, Oxford, United Kingdom
[i] University of Birmingham, Birmingham, United Kingdom
[j] University of Oslo, Oslo, Norway

ARTICLE INFO

Article history:
Received 12 February 2009
Received in revised form 8 June 2009
Accepted 15 June 2009
Available online 18 June 2009

PACS:
07.05.Kf
07.05.Wr
29.50.+v
29.85.+c
87.18.Bb
89.20.Ff

Keywords:
Grid computing
Data mining
Task management
User interface
Interoperability
System integration
Application configuration

ABSTRACT

In this paper, we present the computational task-management tool GANGA, which allows for the specification, submission, bookkeeping and post-processing of computational tasks on a wide set of distributed resources. GANGA has been developed to solve a problem increasingly common in scientific projects, which is that researchers must regularly switch between different processing systems, each with its own command set, to complete their computational tasks. GANGA provides a homogeneous environment for processing data on heterogeneous resources. We give examples from High Energy Physics, demonstrating how an analysis can be developed on a local system and then transparently moved to a Grid system for processing of all available data. GANGA has an API that can be used via an interactive interface, in scripts, or through a GUI. Specific knowledge about types of tasks or computational resources is provided at run-time through a plugin system, making new developments easy to integrate. We give an overview of the GANGA architecture, give examples of current use, and demonstrate how GANGA can be used in many different areas of science.

Program summary

Program title: GANGA
Catalogue identifier: AEEN_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AEEN_v1_0.html
Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland
Licensing provisions: GPL
No. of lines in distributed program, including test data, etc.: 224 590
No. of bytes in distributed program, including test data, etc.: 14 365 315
Distribution format: tar.gz
Programming language: Python
Computer: personal computers, laptops
Operating system: Linux/Unix
RAM: 1 MB
Classification: 6.2, 6.5

☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding author.
E-mail address: jakub.moscicki@cern.ch (J.T. Mościcki).
1 On leave from University of Insbruck, Austria.

doi:10.1016/j.cpc.2009.06.016

Nature of problem: Management of computational tasks for scientific applications on heterogenous distributed systems, including local, batch farms, opportunistic clusters and Grids.
Solution method: High-level job management interface, including command line, scripting and GUI components.
Restrictions: Access to the distributed resources depends on the installed, 3rd party software such as batch system client or Grid user interface.

1. Introduction

Scientific communities are using a growing number of distributed systems, from local batch systems and community-specific services to generic, global Grid infrastructures. Users may debug applications using a desktop computer, then perform small-scale application testing using local resources and finally run at full-scale using globally distributed Grids. Sometimes new resources are made available to the users through systems previously unknown to them, and significant effort may be required to gain familiarity with these systems interfaces and idiosyncrasies. The time cost of mastering application configuration, tracking of computational tasks, archival and access to the results is prohibitive for the end-users if they are not supported by appropriate tools.

Ganga is an easy-to-use frontend for the configuration, execution, and management of computational tasks. The implementation uses an object-oriented design in Python [1]. It started as a project to serve as a Grid user interface for data analysis within the ATLAS [3] and LHC*b* [4] experiments in High Energy Physics where large communities of physicists need access to Grid resources for data mining and simulation tasks. A list of projects which supported the development of Ganga may be found in Acknowledgements.

Ganga provides a simple but flexible programming interface that can be used either interactively at the Python prompt, through a Graphical User Interface (GUI) or programmatically in scripts. The concept of a *job* component is essential as it contains the full description of a computational task, including: the code to execute; input data for processing; data produced by the application; the specification of the required processing environment; post-processing tasks; and metadata for bookkeeping. The purpose of Ganga can then be seen as making it easy for a user to create, submit and monitor the progress of jobs. Ganga keeps track of all jobs and their status through a repository that archives all information between independent Ganga sessions. It is possible to switch between executing a job on a local computer and executing on the Grid by changing a single parameter of a job object. This simplifies the progression from rapid prototyping on a local computer, to small-scale tests on a local batch system, to the analysis of a large dataset using Grid resources.

In Ganga, the user has programmatic access through an Application Programming Interface (API), and has access to applications locally for quick turnaround during development.

Ganga is a user- and application-oriented layer above existing job submission and management technologies, such as Globus [5], Condor [6], UNICORE [7] or gLite [8]. Rather than replacing the existing technologies, Ganga allows them to be used interchangeably, using a common interface as the interoperability layer.

It is possible to make Ganga available to a user community with a high level of customisation. For example, an expert within a field can implement a custom application class describing the specific computational task. The class will encapsulate all low-level setup of the application, which is always the same, and only expose a few parameters for configuration of a particular task. The plugin system provided in Ganga means that this expert customisation will be integrated seamlessly with the core of Ganga at runtime, and can be used by an end-user to process tasks in a way that requires little knowledge about the interfaces of Grid or batch systems. Issues such as differences in data access between jobs executing locally and on the Grid are similarly hidden.

Ganga may be used as a job management system integrated into a larger system. In this case Ganga acts as a library for job submission and control. In particular, Ganga may be used as a building block for the implementation of Grid Portals references [9,10] which allow users access to Grid functionality through their web browsers in a simplified way. These portals are normally domain specific and allow users of a distributed application to run it on the Grid without needing to know much about Grid tools.

Ganga is licensed under the GNU General Public License[2] and is available for download from the project website: http://www.cern.ch/ganga. The installation of Ganga is trivial and does not require privileged access or any server configuration. The Ganga installer script provides a self-contained package and most of the external dependencies are resolved automatically. However, Ganga generally does not attempt to install Grid or batch submission tools or the application software.[3] Typically such software is installed and managed separately by system administrators. Simple configuration files allow customisation and configuration of Ganga at the level of site, workgroup and user.

Between January 2007 and December 2008 Ganga was used at 150 sites around the world, with 2000 unique users running about 250k Ganga sessions.[4]

In this paper, we describe in Section 2 the overall functionality, in Section 3 details of the implementation, and in Section 4 how the progress of jobs is monitored. Section 5 gives an overview of the Graphical User Interface. In Sections 6 and 7 we discuss how Ganga is customised for specific user communities. Interfacing and embedding Ganga in other frameworks is presented in Section 8. In Appendix A we provide some examples of how the API in Ganga can be used.

2. Functionality

Ganga is a user-centric tool that allows easy interaction with heterogeneous computational environments, configuration of the applications and coherent organisation of jobs. Ganga functionality may be accessed by a user through any of several interfaces: a text-based

[2] Ganga is licensed under GPL version 2 or, if preferred by the user, any later version. Details of the GPL are available at http://www.gnu.org/licenses/gpl.html.
[3] Some external dependencies, such as NorduGrid submission tools, are automatically installed.
[4] The usage information was collected from a voluntary usage reporting system implemented in Ganga.

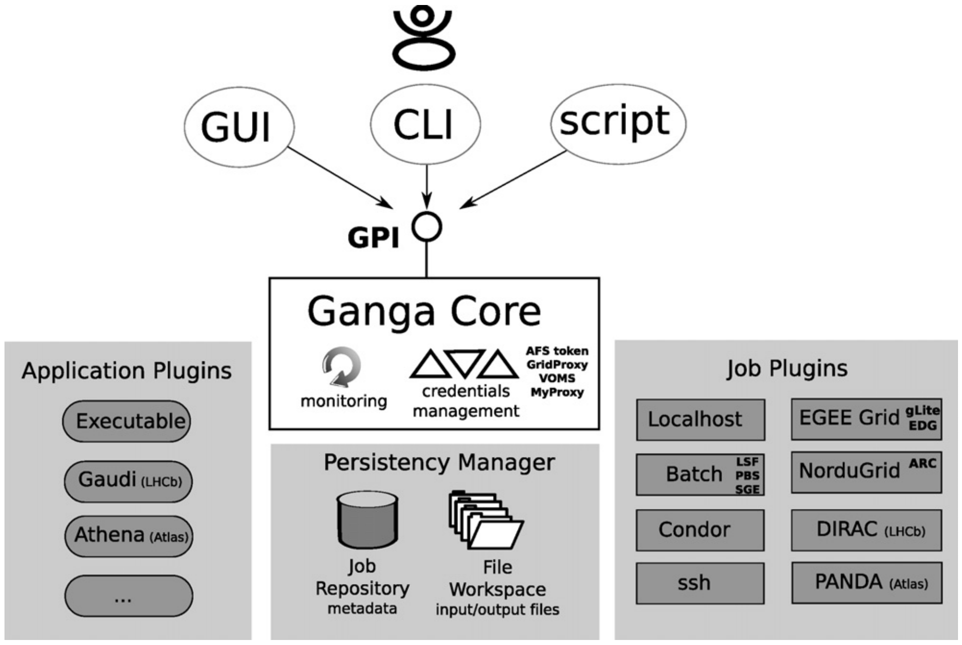


Fig. 1. The overall architecture of Ganga. The user interacts with the Ganga Public Interface (GPI) via the Graphical User Interface (GUI), the Command-Line Interface in Python (CLIP), or scripts. Plugins are provided for different application types and backends. All jobs are stored in the repository.

command line in Python reference [27], a file-based scripting interface and a graphical user interface (GUI). This reflects the different working styles in different user communities, and addresses various usage scenarios such as using the GUI for training new users, the command line to exploit advanced use-cases, and scripting for automation of repetitive tasks. For Ganga sessions the current usage fractions are 55%, 40% and 5% respectively for interactive prompt, scripts and GUI. As shown in Fig. 1, the three user interfaces are built on top of the Ganga Public Interface (GPI) which in turn provides access to the Ganga core implementation.

A job in Ganga is constructed from a set of components. All jobs are required to have an application component and a backend component, which define respectively the software to be run and the processing system to be used. Many jobs also have input and output dataset components, specifying data to be read and produced. Finally, computationally intensive jobs may have a splitter component, which provides a mechanism for dividing into independent subjobs, and a merger component, which allows for the aggregation of subjob outputs. The overall component structure of a job is illustrated in Fig. 2.

By default, the GPI exposes a simplified, top-level view suitable for most users in their everyday work, but at the same time allows for the details of underlying systems to be exposed if needed. An example interactive Ganga session using the GPI is given in Appendix A.

Ganga prevents modification by the user of a submitted job. However, a copy of the job may easily be created and the copy can be modified. Ganga monitors the evolution of submitted jobs and categorises them into the simplified states *submitted*, *running*, *completed*, *failed* or *killed*.

All job objects are stored in a job repository database, and the input and output files associated with the jobs are stored in a file workspace. Both the repository and the workspace may be in a local filesystem or on a remote server.

A large computational task may be split into a number of subjobs automatically according to user-defined criteria and the output merged at a later stage. Each subjob will execute on its own and the merging of the output will take place when all have finalised. The submission of subjobs is automatically optimised if the backend component supports bulk job submission. For example, when submitting to the gLite workload management system [8] the job collection mechanism is used transparently to the user. Job splitting functionality provides a flat list of subjobs suitable for parallel processing of fully independent workloads. However, certain backends allow users to make use of more-sophisticated parallelisation schemes, for example, the Message Passing Interface (MPI) [11] reference update. In this case, Ganga may be used to manage collections of subjobs corresponding to MPI processes.

The GPI allows frequently used job configurations to be stored as `templates`, so that they may easily be reused, and allows jobs to be labelled and organised in a hierarchical `jobtree`.

Ganga has built-in support for handling user credentials, including classic Grid proxies, proxies with extensions for a Virtual Organisation Management Service (VOMS) [12], and Kerberos [13] tokens for access to an Andrew filesystem (AFS) [14]. A user may renew and destroy the credentials directly using the GPI. Ganga gives an early warning to a user if the credentials are about to expire. The minimum credential validity and other aspects of the credential management are fully configurable.

Ganga supports multiple security models. For local and batch backends, the authentication and authorisation of the users is based on the local security infrastructure including user name and network authentication protocols such as Kerberos. Grid security infrastructure (GSI) [15] provides for security across organisational boundaries for the Grid backends. Different security models are encapsulated in pluggable components, which may be simultaneously used in the same Ganga session.

A `Robot` has been implemented for repetitive use-cases. It is a GPI script that periodically executes a series of actions in the context of a Ganga session. These actions are defined by implementations of an action interface. Without programming, the driver can be configured using existing action implementations to submit saved jobs, wait for the jobs to complete, extract data about the jobs to an XML file, generate plain text or HTML summary reports, and email the reports to interested parties. Custom actions can easily be added by either

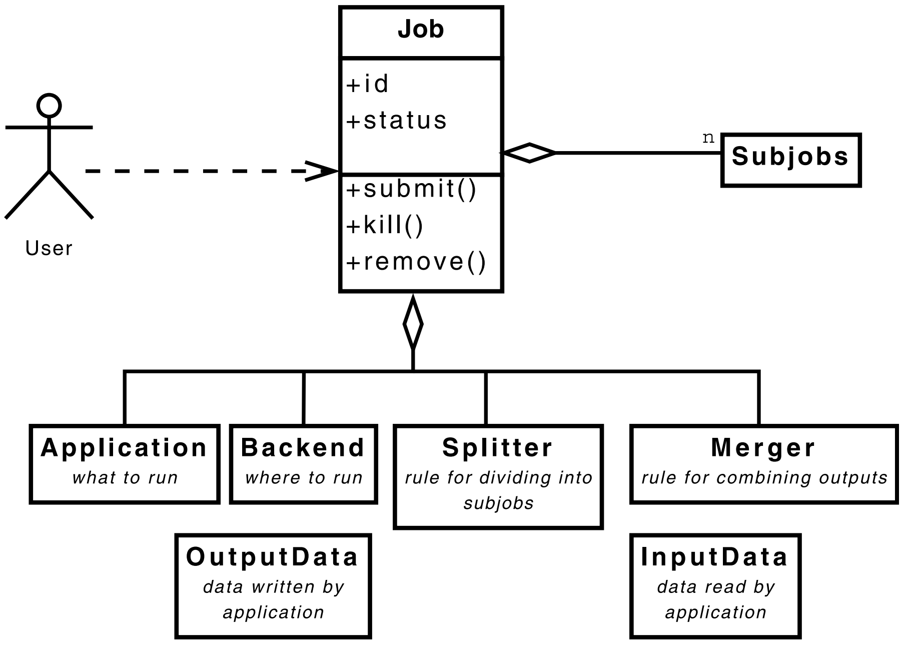


Fig. 2. A set of components in GANGA can be combined to form a complete job. The application to run and the backend where it will run are mandatory while all other components are optional.

extending or aggregating the existing implementations or implementing the action interface directly, allowing for a diverse variety of repetitive use-cases. An example is given in Section 6.1.

Details of the different kinds of GANGA component are given below, along with generic examples. More specialised components, designed for a particular problem domain, are considered in Sections 6 and 7.

2.1. Application components

The application component describes the type of computational task to be performed. It allows the characteristics and settings of some piece of software to be defined, and provides methods specifying actions to be taken before and after a job is processed. The pre-processing (configuration) step typically involves examination of the application properties, and may derive secondary information. For example, intermediate configuration files for the application may be created automatically. The post-processing step can be useful for validation tasks such as determining the validity of the application output.

The simplest application component (`Executable`) has three properties:

`exe:` the path to an executable binary or script;
`args:` a list of arguments to be passed to the executable;
`env:` a dictionary of environment variables and the values they should be assigned before the executable is run.

The configuration method carries out integrity checks – for example, ensuring that a value has been assigned to the `exe` property.

2.2. Backend components

A backend component contains parameters describing the behaviour of a processing system. The list of parameters can vary significantly from one system to another, but can include, for example, a queue name, a list of requested sites, the minimum memory needed and the processing time required. In addition, some parameters hold information that the system reports back to the user, for example the system-specific job identifier and status, and the machine where a job executed.

A backend component provides methods for submitting jobs, and for cancelling jobs after submission, when this is needed. It also provides methods for updating information on job status, for retrieving output of completed jobs and for examining files produced while a job is running.

Backend components have been implemented for a range of widely used processing systems, including: local host, batch systems (Portable Batch System (PBS) [16], Load Sharing Facility (LSF) [17], Sun Grid Engine (SGE) [18], and Condor [19]), and Grid systems, for example, based on gLite [8], ARC [20] and OSG [21]. Remote backend component allows jobs to be launched directly on remote machines using ssh.

As an example, the batch backend component defines a single property that may be set by the user:

`queue:` name of queue to which job should be submitted, the system default queue being used if this left unspecified,

and defines three properties for storing system information:

`id:` job identifier;

`status:` status as reported by batch system;
`actualqueue:` name of queue to which job has been submitted.

In addition, a remote-backend component allows a job defined in a Ganga session running on one machine to be submitted to a processing system known to a remote machine to which the user has access. For example, a user who has accounts on two clusters may submit jobs to the batch system of each from a single machine.

2.3. Dataset components

Dataset components generally define properties that uniquely identify a particular collection of data, and provide methods for obtaining information about it, for example its location and size. The details of how data collections are described can vary significantly from one problem domain to another, and the only generic dataset component in Ganga represents a null (empty) dataset. Other dataset components are specialised for use with a particular application, and so are discussed later.

A strict distinction is made between the datasets and the sandbox (job) files. The former are the files or databases which are stored externally. The sandbox consists of files which are transferred from the user's filesystem together with the job. The sandbox mechanism is designed to handle small files (typically up to 10 MB) while the datasets may be arbitrarily large.

2.4. Splitter components

Splitter components allow the user to specify the number of subjobs to be created, and the way in which subjobs differ from one another. As an example, one splitter component (`ArgSplitter`) deals with executing the same task many times over, but changing the arguments of the application executable each time. It defines a single property:

`args:` list of sets of arguments to be passed to an application.

Specialised splitters deal with creating subjobs that process different parts of a dataset.

2.5. Merger components

Merger components deal with combining the output of subjobs. Typical output includes files containing data in a particular format, for example text strings or data representing histograms. As examples, one merger component (`TextMerger`) concatenates the files of standard output and error returned by a set of subjobs, and another (`RootMerger`) sums histograms produced in ROOT format [22]. Merging may be automatically performed in the background when Ganga retrieves the job output or it may be controlled manually by the user.

3. Implementation

In this section we provide details of the actual implementation of some of the most important parts of Ganga.

3.1. Components

Job components are implemented as plugin classes, imported by Ganga at start-up if enabled in a user configuration file. This means that users only see the components relevant to their specific area of work. Plugins developed and maintained by the Ganga team are included in the main Ganga distribution and are upgraded automatically when a user installs a newer Ganga version. Currently, the list includes around 15 generic plugins and around 20 plugins specific to ATLAS and LHC*b*. Plugins specific to other user communities need to be installed separately but could easily be integrated into the main Ganga distribution.

Plugin development is simplified by having a set of internal interfaces and a mechanism for generating proxy classes [23]. Component classes inherit from an interface class, as seen in Fig. 3. Each plugin class defines a schema, which describes the plugin attributes, specifying type (read-only, read–write, internal), visibility, associated user-convenience filters and syntax shortcuts.

The user does not interact with the plugin class directly but rather with an automatically generated proxy class, visible in the GPI. The proxy class only includes attributes defined as visible in the schema and methods selected for export in the plugin class. This separation of the plugin and proxy levels is very flexible. At the GPI level, the plugin implementation details are not visible; all proxy classes follow the same design logic (for example, copy-by-value); persistence is automatic, session-level locking is transparent. In this way the low-level, internal API is separated from the user-level GPI.

The framework does not force developers to support all combinations of applications and backends, but only the ones that are meaningful or interesting. To manage this, the concept of a *submission handler* is introduced. The submission handler is a connector between the application and backend components. At submission time, it translates the internal representation of the application into a representation accepted by a specific backend. This strategy allows integration of inherently different backends and applications without forcing a lowest-common-denominator interface.

Most of the plugins interact with the underlying backends using shell commands. This down-to-earth approach is particularly useful for encapsulating the environments of different subsystems and avoiding environment clashes. In verbose mode, Ganga prints each command executed so that a user may reproduce the commands externally if needed. Higher-level abstractions such as JSDL [24], OGSA-BES [25] or SAGA API [26] are not currently used, but specific backends that support these standards could readily be added.

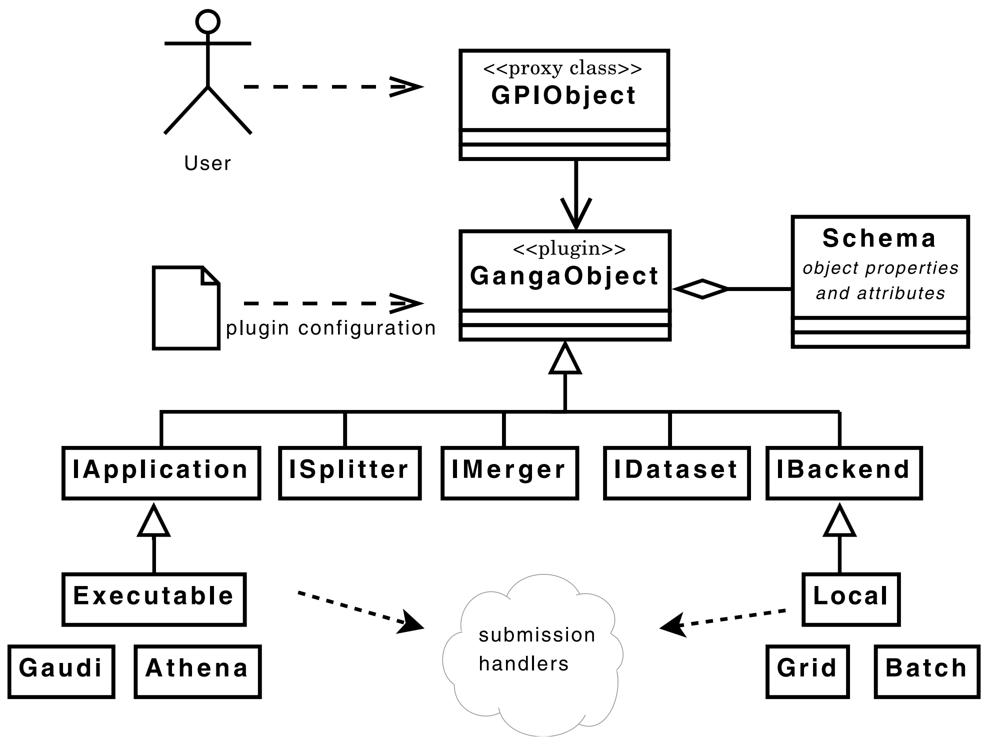


Fig. 3. A component class implements one of the abstract interfaces corresponding to the different parts of a job.

3.2. *Job persistence*

The *job repository* provides job persistence in a simple database, so that any subsequent GANGA session has access to all previously defined jobs. Once a job is defined in a GANGA session it is automatically saved in the database. The repository provides a bookkeeping system that can be used to select particular jobs according to job metadata. The metadata includes such parameters as job name, type of application, type of submission backend, and job status. It can readily be extended as required.

GANGA supports both a local and a remote repository. In the case of the former, the database is stored in the local file system, providing a standalone solution. In the case of the latter, the client accesses an AMGA [28] metadata server. The remote server supports secure connections with user authentication and authorisation based on Grid certificates. Performance tests of both the local and remote repositories show good scalability for up to 10 thousand jobs per user, with the average time of individual job creation being about 0.2 seconds. There is scope for further optimisation in this area by taking advantage of bulk operations and job loading on demand.

The job repository also includes a mechanism to support schema migration, allowing for evolution in the schema of plugin components.

3.3. *Input and output files*

GANGA stores job input and output files in a *job workspace*. The current implementation uses the local file system, and has a simple interface that allows transparent access to job files within the GANGA framework. These files are stored for each job in a separate directory, with sub-directories for input and output and for each subjob.

Users may access the job files directly in the file-system or using GANGA commands such as `job.peek()`. Internally, GANGA handles the input and output files using a simple abstraction layer which allows for trivial integration of additional workspace implementations. Tests with a prototype using a WebDAV [30] server have shown that all workspace data related to a job can be accessed from different locations. In this case, a workspace cache remains available on the local file system.

The combination of a remote workspace and a remote job repository effectively creates a roaming profile, where the same GANGA session can be accessed at multiple locations, similar to the situation for accessing e-mail messages on an IMAP [31] server.

4. Monitoring

GANGA provides two types of monitoring: the internal monitoring updates the user with information on the progress of jobs, and the external monitoring deals with information from third-party services.

4.1. *Internal monitoring*

GANGA automatically keeps track of changes in job status, using a monitoring procedure designed to cope with varying backend response times and load capabilities. As seen in Fig. 4, each backend is polled in a different thread taken from a pool, and there is an efficient mechanism to avoid deadlocks from backends that respond slowly. The poll rate may be set separately for each backend.

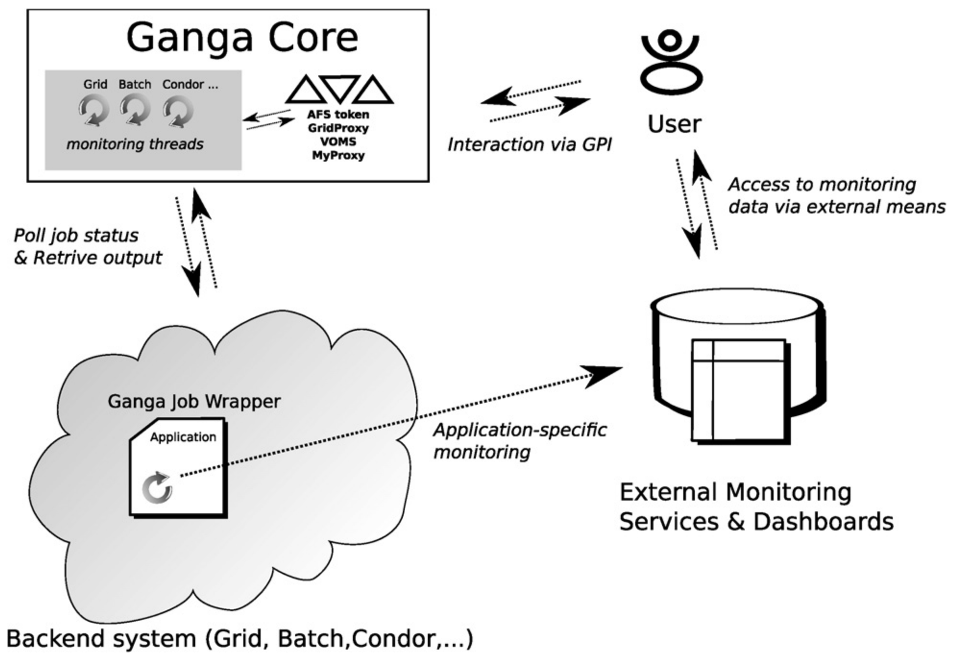

Ganga Core
Grid Batch Condor ...
monitoring threads
AFS token
GridProxy
VOMS
MyProxy
Interaction via GPI
User
Access to monitoring data via external means
Poll job status & Retrive output
Ganga Job Wrapper
Application
Application-specific monitoring
External Monitoring Services & Dashboards
Backend system (Grid, Batch,Condor,...)

Fig. 4. The internal monitoring updates the status of jobs using a pool of threads running in the GANGA core. Additional monitoring thread runs in a job wrapper and sends the monitoring information to external services.

The monitoring subsystem also keeps track of the remaining validity of authentication credentials, such as Grid proxies and Kerberos tokens. The user is notified that renewal is required, and if no action is taken then GANGA is placed in a state where operations requiring valid credentials are disabled.

4.2. External monitoring

GANGA's external monitoring provides a mechanism for dynamically adding third-party monitoring sensors, to allow reporting of different metrics for running jobs.

The monitoring sensors can be inserted both on the client side – where GANGA runs – and on the remote environment (worker node) where the application runs, allowing the user to follow the entire execution flow. Monitoring events are generated at job submission time, at startup, periodically during execution, and at completion.

Individual application and backend components in GANGA can be configured to use different monitoring sensors, allowing collection of both generic execution information and application-specific data.

Use is currently made of two implementations of external monitoring sensors. One is the ATLAS Dashboard application monitoring [32]. Another is a custom service that allows the GANGA user to examine job output in real-time on the Grid. This streaming service is not enabled by default, but must be set up for each user community separately, and may then be requested by a user for specific jobs.

5. Graphical User Interface

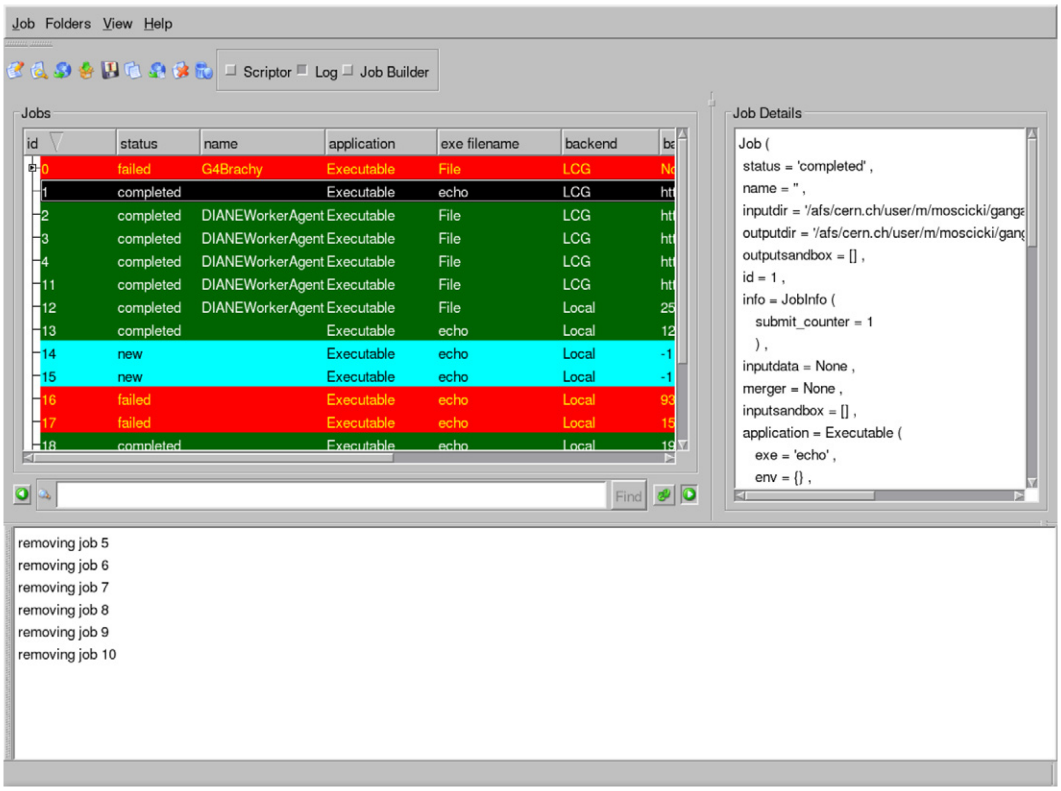
The GANGA Graphical User Interface (GUI), shown in Fig. 5 and built using PyQt3 [33], makes available all of the job-management functionality provided at the level of the GANGA Public Interface. The GUI incorporates various convenience features, and its multi-threaded nature results in a degree of parallelism not possible at the command line: job monitoring and most job-management actions run concurrently, ensuring a good response time for the user.

The job monitoring window takes centre stage, with job status and other monitored attributes displayed in table format. Other features include subjob monitoring, subjob folding/hiding, a job-details display drawer, a logical-collections drawer, and a text-based job-search facility. Many characteristics of the monitoring window can be customised, allowing, for example, selection of the job attributes to be monitored, and of the colours used to denote different job states.

The construction of a job, entailing selection of the required plugins and the entry of attribute values, is achieved from a job-builder window. This displays a foldable tree of job attributes, and associated data-entry widgets. The tree and widgets are generated dynamically based on plugin schemas, ensuring that the GUI automatically supports user-defined plugins without any change being needed to the GUI code. To assist with data entry, drop-down menus list allowed values, wherever these are defined; and tool tips provide explanations of individual job attributes. The job-builder window also features tool buttons for performing a wide range of job-related actions, including creation, saving, copying, submission, termination and removal. Finally, a multifunction `Extras` tool button provides access to arbitrary additional functionality implemented in the plugins.

The GUI also has a scriptor window, providing a favourite-scripts collection, a job-script editor and an embedded PYTHON session. The favourite-scripts collection allows frequently used GANGA scripts to be created, imported, exported and cloned; the job-script editor facilitates quick modification and execution of scripts; and the embedded PYTHON session allows interactive use of GANGA commands.

Finally, a scrollable log window collects and displays all messages generated by GANGA.



id	status	name	application	exe filename	backend	ba
0	failed	G4Brachy	Executable	File	LCG	Nc
1	completed		Executable	echo	LCG	htt
2	completed	DIANEWorkerAgent	Executable	File	LCG	htt
3	completed	DIANEWorkerAgent	Executable	File	LCG	htt
4	completed	DIANEWorkerAgent	Executable	File	LCG	htt
11	completed	DIANEWorkerAgent	Executable	File	LCG	htt
12	completed	DIANEWorkerAgent	Executable	File	Local	25
13	completed		Executable	echo	Local	12
14	new		Executable	echo	Local	-1
15	new		Executable	echo	Local	-1
16	failed		Executable	echo	Local	93
17	failed		Executable	echo	Local	15
18	completed		Executable	echo	Local	19

```
Job (
 status = 'completed' ,
 name = '' ,
 inputdir = '/afs/cern.ch/user/m/moscicki/ganga
 outputdir = '/afs/cern.ch/user/m/moscicki/gang
 outputsandbox = [] ,
 id = 1 ,
 info = JobInfo (
    submit_counter = 1
    ) ,
 inputdata = None ,
 merger = None ,
 inputsandbox = [] ,
 application = Executable (
    exe = 'echo' ,
    env = {} ,
```

```
removing job 5
removing job 6
removing job 7
removing job 8
removing job 9
removing job 10
```

Fig. 5. Ganga Graphical User Interface (GUI). The overview of jobs can be seen to the left, and the details of an individual job are to the right.

6. Use in experiments at the Large Hadron Collider

The ATLAS and LHC*b* experiments aim to make discoveries about the fundamental nature of the Universe by detecting new particles at high energies, and by performing high-precision measurements of particle decays. The experiments are located at the Large Hadron Collider (LHC) at the European Laboratory for Particle Physics (CERN), Geneva, with first particle collisions (events) expected in 2009. Both experiments require processing of data volumes of the order of petabytes per year, rely on computing resources distributed across multiple locations, and exploit several Grid implementations. The data-processing applications, including simulation, reconstruction and final analysis for the experiments, are based on the C++ Gaudi/Athena [34] framework. This provides core services, such as message logging, data access, histogramming, and a run-time configuration system.

The data from the experiments will be distributed at computing facilities around the world. Users performing data analysis need an on-demand access mechanism to allow rapid pre-filtering of data based on certain selection criteria so as to identify data of specific interest.

The role of Ganga within ATLAS and LHC*b* is to act as the interface for data analysis by a large number of individual physicists. Ganga also allows for the easy exchange of jobs between users, something that can otherwise be difficult because of the complex configuration of analysis jobs.

6.1. *The LHC*b *experiment*

The LHC*b* experiment is dedicated to studying the properties of *B* mesons (particles containing the *b* quark) and in this section we describe the way in which Ganga interacts with the application and backend plugins specific to LHC*b*.

In a typical analysis, users supply their own shared libraries, containing user-written classes, and these are loaded at run-time. The LHC*b* applications are driven by a configuration file, which includes definitions of the libraries to load, non-default values for object parameters, the input data to be read, and the output to be created.

Ganga includes an application component for Gaudi-based applications to simplify the task of performing an analysis. During the configuration stage, and before job submission, the application component undertakes the following tasks:

- it locally sets up the environment for the chosen application;
- it determines the user-owned shared libraries required to run the job;
- it parses the configuration file supplied, including all its dependencies;
- it uses information obtained from the configuration file to determine the input data required and the outputs expected;
- it registers the inputs and outputs with the submission backend.

The user, then, only needs to specify the name and version of the application to run, and the configuration file to be used.

Code under development by a user may contain bugs that cause runtime errors during job execution. The transparent switching between processing systems when using Ganga means that debugging can be performed locally, with quick response time, before launching a large-scale analysis on the Grid, where response times tend to be longer.

Some studies in LHC*b*, rather than being based on Gaudi, are performed using the RooFit [37] framework, most notably studies that make use of simplified event simulations. Jobs for these studies require large amounts of processing power, but do not require input data and produce only small amounts of output. This makes them very easy to deploy on the Grid, with support in Ganga provided by a generic ROOT [22] application component.

In the LHC*b* computing model [29], Grid jobs are routed through the DIRAC [35] workload management system (WMS). DIRAC is a pilot-based system where user jobs are queued in the WMS server and the server submits generic pilot scripts to the Grid. Each pilot queries the WMS for a job with resource requirements satisfied by the machine where the pilot script is running. If a compatible job is available, it is pulled from the WMS and started. Otherwise, the pilot terminates and the WMS sends a new pilot to the Grid. This system improves the reliability of the Grid system as seen by the user. Ganga provides a DIRAC backend component that supports submission of jobs to the DIRAC WMS, making use internally of DIRAC's Python API [36].

A *splitter* component implemented specifically for LHC*b* is able to divide the analysis of a large dataset into many smaller subjobs. During the splitting, a file catalogue is queried to ensure that all data associated with an individual subjob is available in its entirety at a minimum of one location on the Grid. This gives significant optimisation, as it avoids subjobs having to copy data across the network before an analysis can start.

In total, above 300k user jobs finished successfully in 2008 with a total CPU consumption of 87 CPU years. The jobs ran at a total of 140 Grid sites across the globe. The system was responsive to a highly irregular usage pattern and spikes of several thousand simultaneous jobs were observed during the year. This usage is expected to rise dramatically after the start of the LHC*b* data taking.

The `Robot` in Ganga is used within LHC*b* for *end-to-end* testing of the distributed analysis model. It submits a representative set of analysis jobs on a daily basis, monitors their progress, and checks the results produced. The overall success rate and the time to obtain the results is recorded and published on the web. The `Robot` monitors this information, producing statistics on the long-term system performance.

6.2. The ATLAS experiment

ATLAS is a general-purpose experiment, designed to allow observation of new phenomena in high-energy proton–proton collisions.

The distributed analysis model is part of the ATLAS computing model [38] which requires that data are distributed at various computing sites, and user jobs are sent to the data.

An ATLAS analysis job typically consists of a Python or shell script that configures and runs user algorithms in the Athena framework [38], reads and writes event files, and fills histograms/n-tuples. More-interactive analysis may be performed on large datasets stored as n-tuples.

There are several scenarios relevant for a user analysis. Some analyses require a fast response time and a high level of user interaction, for which the parallel Root facility PROOF [41] is well suited. Other analyses require a low level of user interaction, with long response times acceptable, and in these cases Ganga and Grid processing are ideal.

Analysis jobs can produce large amounts of data, which may initially be stored at a single Grid site, and may subsequently need to be transferred to other machines. This is supported in ATLAS by the Distributed Data Management system DQ2 [39]. This provides a set of services for moving data between Grid-enabled computing facilities, and maintains a series of databases that track the data movements. The vast amounts of data involved are grouped into datasets, based on various criteria, for example physics characteristics, to make queries and retrievals more efficient.

6.2.1. ATLAS Grid infrastructures

The ATLAS experiment employs three Grid infrastructures for user analysis and for collaboration-wide event simulation and reconstruction. These are the Grid developed in the context of Enabling Grids for e-Science (EGEE, mainly Europe) [42], accessed using gLite middleware [8], the Open Science Grid (OSG, mainly North America) [21], accessed using the PanDA system [40], and NorduGrid (mainly Nordic countries) [43], accessed using the ARC middleware [20]. Ganga seamlessly submits jobs to all three Grid flavours.

6.2.2. ATLAS user analysis

A typical ATLAS user analysis consists of an event-selection algorithm developed in the Athena framework. Large amounts of data are filtered to identify events that meet certain selection criteria. The events of interest are stored in files grouped together as datasets in the DQ2 system. The Ganga components for Athena jobs include the following functionality:

- During job submission, DQ2 is queried for the file content and location of the dataset to be analysed. The number of possible Grid sites is then restricted to the dataset locations.
- A job can be divided into several subjobs, each processing a given number of files from the full dataset.
- In a Grid job, after the Athena application has completed, the user output is stored on the storage element of the site where the job was run, and is registered in DQ2.

In the second half of 2008, more than 4×10^5 Grid jobs were submitted through Ganga by ATLAS users. Following a procedure similar to that of LHC*b*, the Ganga `Robot` submits test jobs daily to ATLAS Grid sites. Test results are used to guide users to sites that are performing well, avoiding job failures on temporarily misconfigured sites.

6.2.3. ATLAS small-scale event simulations

In addition to data analysis, users sometimes need to simulate event samples of the order of a few tens of thousands of events. The *AthenaMC* application component has been developed to integrate software used in the official ATLAS system for event simulation. This

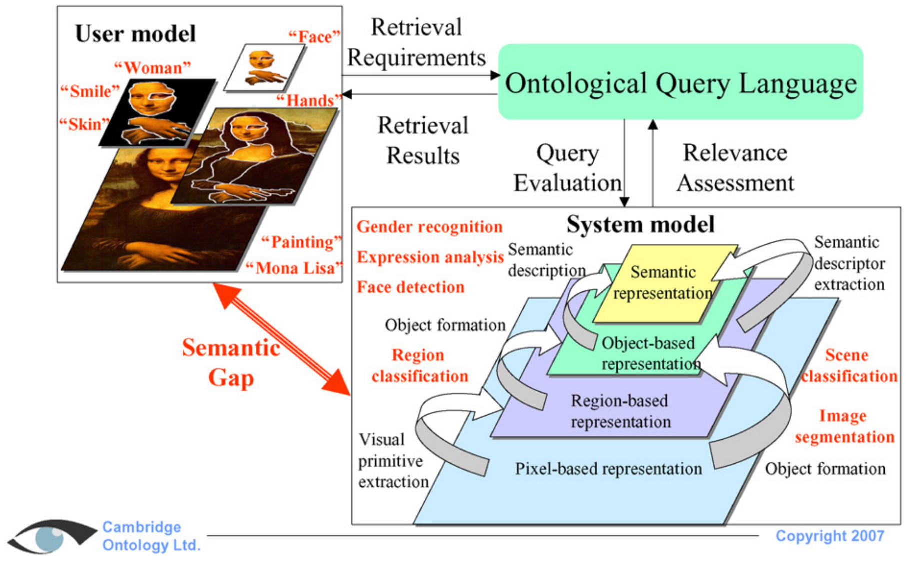


Fig. 6. Schematic representation of the image-retrieval system developed by Imense Ltd. Image characteristics are determined by applying feature-extraction algorithms, and an ontological query language bridges the semantic gap between terms that might be employed in a user query and terms understood by the processing system.

component consists of a set of PYTHON classes that together handle input parameters, input datasets and output datasets for the three production steps: event generation, detector simulation, and event reconstruction. As in the case of user analysis, datasets are managed by the DQ2 system.

7. Other usage areas

GANGA offers a flexible and extensible interface that make it useful beyond the original scope of particle-physics applications in the ATLAS and LHC*b* experiments. Here we provide details of just a few of the other contexts in which GANGA has been used.

7.1. Enabling industrial-scale image retrieval

Imense Ltd.,[5] a Cambridge-based startup company, has implemented a novel image retrieval-system (Fig. 6), featuring automated analysis and recognition of image content, and an ontological query language. The proprietary image analysis, developed from published research [44], includes recognition of visual properties, such as colour, texture and shape; recognition of materials, such as grass or sky; detection of objects, such as human faces, and determination of their characteristics; and classification of scenes by content, for example, beach, forest or sunset. The system uses semantic and linguistic relationships between terms to interpret user queries and retrieve relevant images on the basis of the analysis results. Moreover, the system is extensible, so that additional image classification modules or image context and metadata can easily be integrated into the index.

By using the GANGA framework for job submission and management, it has been possible to port and deploy a large part of Imense's image-analysis technology to the Grid and build a searchable index for more than twenty-million high-resolution photographic images.

The processing stages for the image-search system – image analysis and indexing – are intrinsically sequential. Analysis has been parallelised at the level of single images or small subsets of images. Each image can therefore be processed in isolation on the Grid, with this processing usually taking a few to ten seconds. In order to minimise overheads, images are grouped in sets of a few hundred per job submitted through GANGA. Results of the image processing and analysis are passed back to the submission server once a job has successfully completed.

Support for Imense has been added to GANGA through the implementation of two specialised components: an application component that deals with running the image-processing software, and a dataset component for taking care of the output. As usual with GANGA, the jobs can run both locally and on the Grid, giving maximum flexibility.

At runtime, images are retrieved and segmented one at a time, all of the images are classified, and finally an archive is created of the output files (several per input image). The archive is returned using the sandbox mechanism in GANGA when using the `Local` backend, and is uploaded to a storage element when using the Grid `LCG` backend.

The specialised dataset component provides methods for downloading a results archive from a storage element, and for unpacking an archive to a destination directory. These methods are invoked automatically by GANGA when an image-processing job completes: the effect for the user is that a list of images is submitted for processing and results are placed in the requested output location independently of the backend used.

7.2. Smaller collaborations in High Energy Physics

Large user communities, such as ATLAS and LHC*b*, profit from encapsulating shared use cases as specialised applications in GANGA. In contrast, individual researchers or developers in the context of rapid prototyping activities may opt to use generic application components.

[5] http://imense.com.

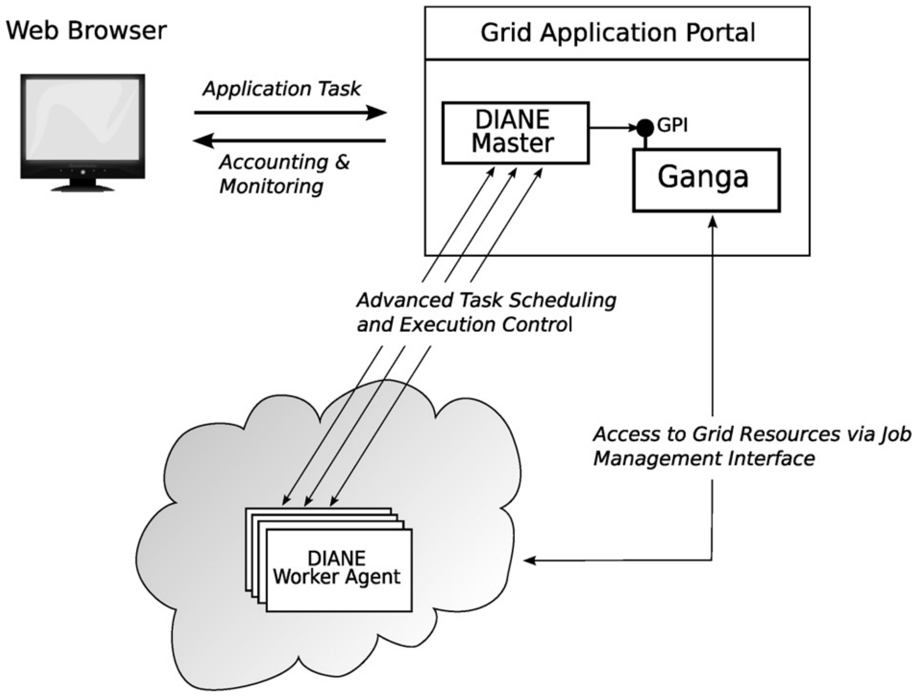


Fig. 7. GANGA as a job management component embedded in DIANE, with an application portal.

In such cases, GANGA still provides the benefits of bookkeeping and a programmatic interface for job submission. As an example of this way of working, a small community of experts in the design of gaseous detectors use GANGA to run the GARFIELD [45] simulation program on the Grid. A GANGA script has been written that generates a chain of simulation jobs using the GARFIELD generator of macro files and GANGA's `Executable` application component. The GARFIELD executables, and a few small input files, are placed in the input sandbox of each job. Histograms and text output are then returned in the output sandbox. This simple approach allowed integration of GARFIELD jobs in GANGA in just a few hours.

7.3. GANGA *integrated with lightweight Grid middleware*

The open-plugin architecture of GANGA allows easy integration of additional Grid middleware, as has been achieved, for example, with the ARC (Advanced Resource Connector) Grid middleware [20]. This is a product of the NorduGrid project [43], and is used by many academic institutions in the Nordic countries and elsewhere.

ARC jobs are accepted and brokered by a Grid manager, running at site level, and resource lookup is done through load balancing and runtime environments advertised by individual sites. File storage and access is 'cloudy', meaning that all files registered in Grid-wide catalogues are accessible to all worker nodes. File transfers are handled by the Grid manager, between job acceptance and execution. ARC-connected resources are used, e.g., by researchers in bioinformatics, genomics, meteorology, in addition to High Energy Physics.

GANGA has been interfaced to ARC through a backend, which translates GANGA input into ARC-readable xRSL language. The ARC user client is lightweight, and binaries are provided as an external library at GANGA install time. The main user of this integration is the ATLAS experiment (see Section 6.2), where it is the main user access portal to one of the experiment's three main computing grids. Further collaboration between ARC and GANGA is envisaged, to employ GANGA as a fully featured frontend to ARC.

8. Interfacing to other frameworks

The GANGA Public Interface constitutes an API for generic job submission and management. As a result, GANGA may be programmatically interfaced to other frameworks, and used as a convenient abstraction layer for job management. GANGA has been used in combination with DIANE [46], a lightweight agent-based scheduling layer on top of the Grid, in a number of scientific activities. These have included: dosimetry-related simulation studies in medical physics [47]; regression testing of the Geant 4 [48] detector-simulation toolkit; in-silico molecular docking in searches for new drugs against potential variants of an influenza virus [49]; telecommunication applications [50]; and theoretical physics [51]. The DIANE worker agents are executed as GANGA jobs, so that resource usage may be controlled by the user from the GANGA interface. This approach allows the efficiency of the DIANE overlay scheduling system to be combined with the well-structured job management offered by GANGA, as well as combining Grid and non-Grid resources under a uniform interface. Also, this allows the efficient implementation of low-latency access to Grid resources and improvements to responsiveness when supporting on-demand computing and interactivity [52].

GANGA may be embedded in web-based services such as the bio-informatics portal developed by ASGC, Taipei. The portal is fully customized for analysis of candidate drugs against avian flu. The portal engine delegates job management to the embedded DIANE/GANGA framework, as shown in Fig. 7. Following this approach, users can switch between different resources, or access heterogeneous computing environments through a single same web interface.

9. Conclusion

Ganga has been presented as a tool for job management in an environment of heterogeneous resources and is particularly suited to the Grid paradigm that has emerged in large-scale distributed computing. Ganga makes it easy to define a computational task that can be executed locally for debugging, and subsequently be run on the Grid, for large scale data mining. We have shown how Ganga simplifies task specification, takes care of job submission, monitoring and output retrieval, and provides an intuitive bookkeeping system.

We have demonstrated the advantages of having a well-defined API, which can be used interactively at the Python prompt, through a GUI or programmatically in scripts. By virtue of its plugin system, Ganga is readily extended and customised to meet the requirements of new user communities. Examples of Ganga usage have been provided in particle physics, medical physics and image processing.

Existing command-line submission interfaces, such as gLite, tend to include only limited usability features. Some higher level tools, for example GridWay [53], present jobs as if they were Unix processes and corresponding command line utilities. Interfaces based on Condor job-submission scripts have also been developed [54]. A distinctive feature of Ganga is that it may easily be adapted to different styles of working, allowing simultaneous use of three different interfaces. Ganga also provides a higher level of abstraction than most job-management tools, and allows a user to focus on solving the domain-specific problems, rather than changing their way of working each time they switch to a new processing system.

Ganga has a large user base and is in active development. Ganga is a tool which may easily be used to support new scientific or commercial projects on a wide range of distributed infrastructures.

Acknowledgements

The development of Ganga has been supported by the GridPP project in the United Kingdom [2], with funding from the Science and Technology Facilities Council (STFC) and its predecessor, the Particle Physics and Astronomy Research Council (PPARC); by the D-Grid project in Germany, with funding from the Bundesministerium für Bildung und Forschung (BMBF); and by the project for Enabling Grids for E-sciencE (EGEE), co-funded by the European Commission (contract number INFSO-RI-031688) through the Sixth Framework Programme.

Ganga has received contributions over the years from a number of individuals. Particular thanks are due to David Adams, Marcello Barisonzi, Mike Kenyon, Wim Lavrijsen, Janusz Martyniak, Pere Mato, Caitriana Nicholson, Rebecca Ronke, Vladimir Romanovski, David Tuckett, Ruth Dixon del Tufo, Craig Tull.

The developers would also like to thank the large number of users, from both within and outside particle physics, for their valuable suggestions for improving Ganga, and for their help in debugging problems.

Appendix A. Examples

Below we give a set of examples of working with Ganga. For ease of reading, Python keywords are in bold. First we look at a complete Ganga session.

```
~ \% ganga
*** Welcome to Ganga ***
Version: Ganga-5-1-0
Documentation and support: http://cern.ch/ganga
Type help() or help('index') for online help.

This is free software (GPL), and you are welcome to redistribute
it under certain conditions; type license() for details.

[1]: j=Job(name='MyJob')          # Create a default job
[2]: j.submit()                   # Submit the job

# wait for the monitoring

[3]: j.peek('stdout')             # Look at the output
[4]: j=j.copy(name='GridJob')     # Make a copy of the job
[5]: j.backend=LCG()              # Change backend to the Grid
[6]: j.submit()                   # Submit the job
[7]: jobs                         # List jobs

...job listing...
[8]: Exit                         # Quit Ganga.
```

In the next example, we create a job for analysis of LHC*b* data. A splitter is used to divide the analysis between subjobs. Data are assigned using logical identifiers, and the DIRAC WMS ensures that subjobs are sent to locations where the required data are available.

```
[1]: j=Job(application=DaVinci(),backend=Dirac())
[2]: j.inputdata=LHCbDataset(files=[  # Data to read
...     'LFN:/foo.dst',
...     'LFN:/bar.dst',
...     many more data files])
[3]: j.splitter=DiracSplitter()  # We want subjobs
[4]: j.submit()
```

```
Job submission output
```

Here, we use the fact that standard Python commands are available at the Ganga prompt, and print information on subjobs.

```
# Status of jobs and where they ran
[5]: for subjob in j.subjobs:
···        print subjob.status, subjob.actualCE

42

# Find backend identifier of all failed jobs
[6]: for j in jobs.select(status='failed'):
···        print j.backend.id

42
```

Groups of jobs may be accessed and manipulated using simple methods:

```
[1]: jobs.select(status='failed').resubmit()
[2]: jobs.select(name='testjob').kill()
[3]: newjobs = jobs.select(status='new')
[4]: newjobs.select(name='urgent').submit()
```

References

[1] G. van Rossum, F.L. Drake Jr., The Python Language Reference Manual: Revised and Updated for Version 2.5, Network Theory Limited, Bristol, ISBN 0-9541617-8-5, 2006.
[2] P.J.W. Faulkner, et al., GridPP Collaboration, GridPP: Development of the UK computing Grid for particle physics, J. Phys. G: Nucl. Part. Phys. 32 (1).
[3] G. Aad, et al., ATLAS Collaboration, The ATLAS experiment at the CERN Large Hadron Collider, JINST 3 (2008) S08003.
[4] A.A. Alves Jr., et al., LHCb Collaboration, The LHC*b* detector at the LHC, JINST 3 (2008) S08005.
[5] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the Grid: Enabling scalable virtual organizations, International J. Supercomputer Applications 15 (3) (2001).
[6] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: The Condor experience, Concurrency and Computation: Practice and Experience 17 (2–4) (2005) 323–356.
[7] A. Streit, et al., UNICORE – From Project Results to Production Grids, Grid Computing: The New Frontiers of High Performance Processing Advances in Parallel Computing, vol. 14, Elsevier, 2005, pp. 357–376.
[8] P. Andreett, et al., The gLite workload management system, J. Phys.: Conf. Ser. 119 (2008) 062007.
[9] M.P. Thomas, et al., Grid portal architectures for scientific applications, J. Phys.: Conf. Ser. 16 (2005) 596.
[10] M. Li, M. Baker, A review of Grid Portal technology, in: J.C. Cunha, O.F. Rana (Eds.), Grid Computing: Software Environments and Tools, Springer-Verlag London Ltd., 2006, pp. 126–156.
[11] M. Snir, S. Otto, MPI – The Complete Reference: The MPI Core, MIT Press, ISBN 0262692155, 1998.
[12] R. Alfieri, et al., From gridmap-file to VOMS: Managing authorization in a Grid environment, Future Generation Computer Systems 21 (2005) 549.
[13] B.C. Neumann, T. Ts'o, Kerberos: An authentication service for computer networks, IEEE Communications Magazine 32 (9) (1994) 33.
[14] J.H. Morris, et al., Andrew: A distributed personal computing environment, Commun. ACM 29 (3) (1986) 184.
[15] I. Foster, et al., A security architecture for computational Grids, in: Proc. 5th ACM Conference on Computer and Communications Security Conference, 1998, pp. 83–92.
[16] R.L. Henderson, Job scheduling under the Portable Batch System, in: D.G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, in: Lecture Notes in Computer Science, vol. 949, Springer, Berlin, 1995, pp. 279–294.
[17] U. Schwickerath, V. Lefebure, Usage of LSF for batch farms at CERN, J. Phys.: Conf. Ser. 119 (2008) 042025.
[18] W. Gentzsch, Sun Grid Engine: Towards creating a compute power Grid, in: R. Buyya, G. Mohay, P. Roe (Eds.), Proc. First IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Los Alamitos, CA, 2001, pp. 35–36.
[19] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: The Condor experience, Concurrency Computat.: Pract. Exper. 17 (2005) 323.
[20] M. Ellert, et al., Advanced Resource Connector middleware for lightweight computational Grids, Future Generation Computer Systems 23 (2007) 219.
[21] R. Pordes, et al., The Open Science Grid, J. Phys.: Conf. Ser. 78 (2007) 012057.
[22] R. Brun, F. Rademakers, ROOT – an object oriented data analysis framework, Nucl. Instrum. Methods A 389 (1997) 81.
[23] E. Gamma, et al., Design Patterns: Elements of Reusable Object-Orientated Software, Addison-Wesley, 1995.
[24] Job Submission Description Language (JSDL) Specification, Version 1.0, http://www.gridforum.org.
[25] OGSA Basic Execution Service, http://www.ogf.org.
[26] A Simple API for Grid Applications (SAGA), http://www.ogf.org.
[27] F. Perez, B.E. Granger, IPython: A system for interactive scientific computing, Computing in Science and Engineering 9 (3) (2007) 21.
[28] B. Koblitz, N. Santos, V. Pose, The AMGA metadata service, J. Grid Computing 6 (2008) 61.
[29] R. Antunes-Nobrega, et al., LHCb Collaboration, LHC*b* computing, Technical Design Report CERN/LHCC 2005-019 LHCb TDR-11, 2005.
[30] E.J. Whitehead Jr., World Wide Web Distributed Authoring and Versioning (WebDAV): An introduction, StandardView 5 (1997) 3.
[31] P. Heinlein, P. Hartleban, The Book of IMAP: Building a Mail Server with Courier and Cyrus, No Startch Press, San Francisco, CA, 2008.
[32] J. Andreeva, et al., Dashboard for the LHC experiments, J. Phys.: Conf. Ser. 119 (2008) 062008.
[33] B. Rempt, GUI Programming with Python: QT Edition, Command Prompt Inc., White Salmon, WA, 2001.
[34] G. Barrand, et al., GAUDI – a software architecture and framework for building HEP data processing applications, Computer Physics Communications 140 (2001) 45.
[35] A. Tsaregorodtsev, et al., DIRAC: A community Grid solution, J. Phys.: Conf. Ser. 119 (2008) 062048.
[36] S. Paterson, LHC*b* distributed data analysis on the computing Grid, PhD Thesis, University of Glasgow, 2006 [CERN-THESIS-2006-053].
[37] W. Verkerke, D. Kirkby, The RooFit toolkit for data modeling, in: Contribution MOLT007, Proc. 2003 Conference for Computing in High Energy and Nuclear Physics, La Jolla, CA [SLAC eConf C0303241].
[38] G. Duckeck, et al. (Eds.), ATLAS computing, Technical Design Report CERN/LHCC 2005-022 ATLAS TDR-017, 2005.
[39] M. Branco, et al., Managing ATLAS data on a petabyte-scale with DQ2, J. Phys.: Conf. Ser. 119 (2008) 062017.
[40] T. Maeno, PanDA: Distributed production and distributed analysis system for ATLAS, J. Phys.: Conf. Ser. 119 (2008) 062036.
[41] M. Ballintijn, et al., Parallel interactive data analysis with PROOF, Nucl. Instrum. Methods 559 (2006) 13.
[42] R. Jones, An overview of the EGEE project, in: C. Türker, M. Agosti, H.-J. Schek (Eds.), Peer-to-Peer, Grid, and Service-Orientation in Digital Library Architectures, in: Lecture Notes in Computer Science, vol. 3664, Springer, Berlin, 2005, pp. 1–8.
[43] M. Ellert, et al., The NorduGrid project: Using Globus toolkit for building Grid infrastructure, Nucl. Instrum. Methods A 502 (2003) 407.
[44] C. Town, D. Sinclair, Language-based querying of image collections on the basis of an extensible ontology, Image and Vision Computing 22 (2004) 251.

[45] R. Veenhof, Garfield – simulation of gaseous detectors, CERN Program Library User Guide W5050 (1984 et seq.).
[46] J.T. Mościcki, Distributed analysis environment for HEP and interdisciplinary applications, Nucl. Instrum. Methods A 502 (2003) 426.
[47] J.T. Mościcki, et al., Distributed Geant4 Simulation in Medical and Space Science Applications using DIANE framework and the GRID, Nucl. Phys. B (Proc. Suppl.) 125 (2003) 327–331.
[48] J. Allison, et al., Geant 4 – a simulation toolkit, Nucl. Instrum. Methods A 506 (2003) 250.
[49] H.-C. Lee, et al., Grid-enabled high-throughput in silico screening against influenza A neuraminidase, IEEE Trans. NanoBioscience 5 (4) (2006) 288.
[50] J.T. Mościcki, et al., Dependable distributed computing for the International Telecommunication Union Regional Radio Conference RRC06, IEEE Computer, submitted for publication.
[51] J.T. Mościcki, et al., Lattice QCD thermodynamics on the Grid, Computer Physics Communications, submitted for publication.
[52] C. Germain-Renaud, C. Loomis, J.T. Mościcki, R. Texier, Scheduling for responsive Grids, J. Grid Computing 6 (2008) 15–27.
[53] J. Herrera, et al., Porting of scientific applications to Grid computing on GridWay, Scientific Programming 13 (4) (2005) 317–331.
[54] R.P. Bruin, et al., Job submission to Grid computing environments, Concurrency and Computation: Pract. and Exper. 20 (11) (2008) 1329–1340.