

Ganga user interface for job definition and management

U. Egede^a, K. Harrison^b, R.W.L. Jones^c, A. Maier^d, J.T. Moscicki^d,
G.N. Patrick^e, A. Soroko^f, C.L. Tan^g

^a *Department of Physics, Imperial College London, SW7 2AZ, UK*

^b *Cavendish Laboratory, University of Cambridge, CB3 0HE, UK*

^c *Department of Physics, University of Lancaster, LA1 4YB, UK*

^d *CERN, CH-1211 Geneva 23, Switzerland*

^e *Rutherford Appleton Laboratory, Chilton, Didcot, OX11 0QX, UK*

^f *Department of Physics, University of Oxford, OX1 3RH, UK*

^g *School of Physics and Astronomy, University of Birmingham, B15 2TT, UK*

Abstract

GANGA is a frontend for job definition and management in a distributed environment. It is being developed in the context of two high-energy physics experiments, ATLAS and LHCb, but is a component-based system that can readily be extended to meet the needs of other user communities. GANGA allows the user to specify the software to run, which may include user code, to set values for any configurable parameters, to select data for processing, and to submit jobs to a variety of local batch queues and Grid-based systems, hiding Grid technicalities. After jobs are submitted, GANGA monitors their status, and takes care of saving any output. GANGA offers a Command-Line Interface in Python (CLIP), possibilities for scripting, and a Graphical User Interface (GUI), which significantly simplifies basic tasks. This paper describes the job model and architecture of GANGA, and illustrates GANGA's main features as a user interface.

1 Introduction

GANGA [1] is a component-based system providing a user interface for job definition and management in a distributed environment. It is being developed in the context of two high-energy physics experiments, ATLAS [2] and LHCb [3], whose specific needs it addresses, but offers possibilities for extension and customisation that make it potentially interesting for a wide range of user communities. It is implemented in Python [4].

ATLAS and LHCb will investigate various aspects of particle production and decay in high-energy proton-proton interactions at the Large Hadron Collider (LHC) [5], due to start operation at CERN, Geneva, in 2007. Both experiments will require processing of data volumes of the order of petabytes per year, and will rely on computing resources distributed across multiple locations. The experiments' data-processing applications, including simulation, reconstruction and physics analysis, are all based on the GAUDI/ATHENA C++ framework [6]. This provides core services, such as message logging, data access, histogramming; and allows runtime configuration via options files. These may be written in Python, or using a value-assignment syntax similar to that of C++, and can have considerable complexity. The two experiments have developed different solutions for cataloguing the available data: the ATLAS Metadata Interface (AMI) [7] and the LHCb Bookkeeping Database [8].

GAUDI/ATHENA jobs for simulation and reconstruction typically use software that results from a coordinated, experiment-wide effort, and is installed at many sites. The person submitting the jobs, pos-

sibly a production manager, performs the job configuration, which involves selecting the algorithms to be run, defining the algorithm properties and specifying inputs and outputs. The situation is similar for an analysis job, except that the physicists running a given analysis will usually want to load one or more algorithms that they have written themselves, and so use code that may be available only in an individual physicist's work area.

Several possibilities for exploiting distributed computing resources, and for processing distributed data, have been developed using Grid technologies. These include systems from national and international projects, such as LCG (LHC Computing Grid) [9], EGEE (Enabling Grids for e-Science) [10], OSG (Open Science Grid) [11], GridPP (UK Computing for Particle Physics) [12] and NorduGrid [13], and experiment-specific solutions, such as DIAL [14] in ATLAS and DIRAC [15] in LHCb. The situation for the user is confusing, in that different sites hosting data of interest may have adopted different systems, requiring different commands and job descriptions, and all of the systems continue to evolve. Also, a user will often wish to run test jobs on local resources, where debugging is simpler, before submitting to remote sites, and will prefer not to have to do different things in the two cases.

GANGA provides a single frontend for submission to multiple backends, shielding the user from the uncertainties and technical complications associated with data processing in a distributed environment. This paper describes the job model and architecture of GANGA, and illustrates GANGA's use.

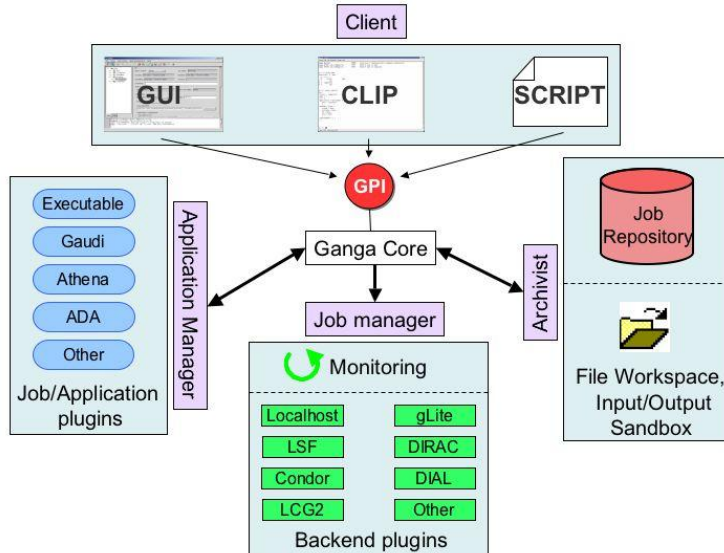


Figure 1: Schematic representation of the GANGA architecture. The main functionality is divided between Application Manager, Job Manager and Archivist, and is accessed by the Client through the GANGA Public Interface (GPI). The client may run the Graphical User Interface (GUI), the Command-Line Interface In Python (CLIP) or GPI scripts.

2 Ganga job model

Job-related information can be divided into three categories:

- Application information is information provided by the user to define the task to be performed within the job. This can include a specification of the software to be run, parameter values, settings for environment variables, and definitions of input and output files.
- Backend information is information provided by the user to define the computing requirements for the processing system (backend) to which the job is to be submitted. This can include indications of the site, cluster or node where the job is to run, the amount of memory needed, the processing time, and the priority that the job should be assigned.
- System information is information made available by the processing system to allow tracking of a job's progress. This can include the job's identifier within the system, and dynamic information on job status and resources used.

In GANGA, a job is represented by an object that holds all categories of information. A user directly sets values for application and backend properties, whereas the system information is read-only for the user and is updated periodically within a monitoring loop where GANGA queries the processing system(s).

In job descriptions for systems such as LCG [9] and Condor [16], the application information is specified in a fairly generic way, and typically amounts to giving the name of a script or binary executable, a list of values to be passed to the executable as command-line arguments, and lists of input and out-

put files. This kind of approach has the advantage that it allows a common description of all applications in terms of a small number of parameters, but with the disadvantage that the description contains no information on the roles of the different command-line arguments and files. It is the user's responsibility to ensure consistency between these and a job's script or executable. In the context of distributed computing, there is the added complication that a script may contain references that are locally valid, but are meaningless at a remote site.

In GANGA, rather than adopting a generic solution, each type of application is defined by its own schema. This places in evidence an application's configurable properties, and their meanings, and ensures that information is, or can be made, site-independent.

Each of the backends to which GANGA may submit is similarly defined in terms of a schema. This allows for differences between backends both in the user-configurable properties supported and in the amount, and type, of system information available.

In practice, different types of job, application and backend are supported in GANGA through plug-in classes. Classes in a given category must provide their own implementations of some specific methods – for example, submit and kill for backend classes – but can have different properties, defined through the relevant schema. Intermediary classes are implemented to take care of processing a job's application information and formatting it in a way appropriate for the backend to which the job is to be submitted. Users can easily add new plug-in classes, or suppress the loading of unwanted classes, so that GANGA can readily be extended or customised to meet the requirements of different user communities.

```

>>> j = Job( application = "DaVinci", backend = "DIRAC" )
>>> j.application.version = "v12r11"
>>> j.application.optsfile = "DVHLTCore.opts"
>>> j.submit()

```

Figure 2: Example job submission using CLIP. Here, the user wishes to run the LHCb analysis application, DaVinci [17], on the experiment’s own distributed processing system, DIRAC [15]. The user selects the version number for the application (v12r11) and the options file to be used (DVHLTCore.opts), and accepts the default values for the DIRAC backend. Submission is then a one-line command.

3 Ganga architecture

The main functionality of GANGA is divided among three core components, corresponding to Application Manager, Job Manager and Archivist (Fig. 1). These communicate with one another through the GANGA core, and their functionality is made available through the GANGA Public Interface (GPI). Users access the GPI through a fourth GANGA component, the Client, which provides a Graphical User Interface (GUI), a shell – the a Command-Line Interface In Python (CLIP) – and the possibility to run GPI scripts. The four components listed are defined in such a way that they could be placed on different hosts, although they can also all be installed on a single machine.

The Application Manager needs a host where the environment for running the user’s application is set up, and the required software is available. This component takes a user’s locally valid description of how an application should be run, and converts it into a site-independent job representation. This operation may include translating local path and environment settings, and identifying user files that must be shipped with the job.

The Job Manager needs a host that allows submission to one or more backends. This component accepts information provided by the Application Manager, creates a wrapper script for running the specified application on the chosen backend, and generates any additional job-description files that the backend requires. It executes job-related commands on behalf of the Client, and also performs monitoring, periodically querying backends for job status. The Job Manger automatically retrieves a job’s output files when the monitoring system indicates that the job is completed

The Archivist, which places no special requirements on the host machine, has two main functions. First, it provides a job repository, which allows job objects to be stored in a persistent form. Jobs created in one invocation of GANGA are accessible in subsequent invocations, until explicitly deleted, and the status of jobs in the repository is kept synchronised with the status reported by the monitoring system. As a second function, the Archivist manages file workspace, providing temporary storage for job output, and for files that may be shared at runtime by several jobs.

Details of the Client functionality are given in the next section.

4 Using Ganga

4.1 Configuration

GANGA has default parameters and behaviour that can be redefined at startup using one or more configuration files, which use the syntax understood by the standard Python [4] ConfigParser module. Configuration files can be introduced at the level of system, group or user, with each successive file processed able to override settings from preceding files.

One important feature of the configuration files is that they allow selection of the Python packages that should be initialised when GANGA starts, and consequently of the modules, classes, objects and functions that are made available through the GPI. The configuration files also allow modification of the default values to be used when creating objects from plug-in classes, and permit actions such as choosing the log level for messaging, specifying the location of the job repository, and changing certain visual aspects of GANGA.

4.2 GPI and CLIP

The GANGA Public Interface (GPI) is designed as a high-level, user-friendly Python API for job manipulation, and may readily be used in scripting. GANGA’s Command-Line Interface in Python (CLIP) is intended as an extension to the GPI, with enhanced support for interactive operations at the Python prompt. In the current implementation, CLIP and GPI are identical, but at a later stage CLIP may add features such as higher-level shortcuts and a property-aliasing mechanism.

From CLIP, or after GPI has been imported into a script, the user has access to functionality determined by the configuration file(s) used. For example, an LHCb user at CERN might have a setup that would load the plug-ins for the LHCb simulation, reconstruction and analysis applications, for the local LSF batch system, for LCG [9] and for DIRAC [15]. An ATLAS user might, instead, load the plug-ins for submitting jobs to DIAL [14], and objects for accessing the DIAL catalogues for dataset selection. CLIP and the GPI additionally make available an object for interacting with the job repository, allowing examination of the stored jobs.

With CLIP or GPI, a user needs give only a few commands to be able to set application properties and submit a job to run the application on a

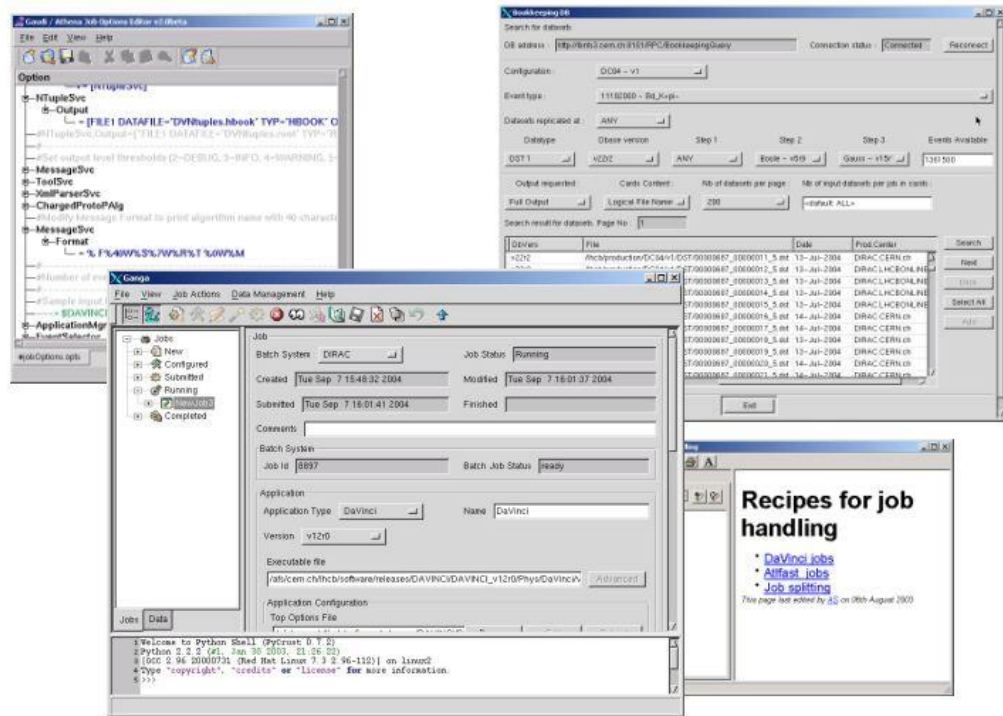


Figure 3: Screenshots showing components of the GANGA GUI prototype: main window (bottom left), window for Job-Options Editor (top left), window for querying the LHCb Bookkeeping Database [8] (top right), and help window (bottom right)

Grid backend (Fig. 2). The output, when ready, is retrieved automatically. The user is protected from the technicalities of the submission and output-retrieval operations, and is spared from working directly with job scripts and description files, which, even for fairly simple jobs, can be tens or hundreds of lines long.

4.3 GUI

Several components have been implemented for a prototype GUI (Fig. 3). The main window features three panels, all of which can be resized by the user, drop-down menus, and a toolbar, with the following functionality:

- The upper-left panel displays a hierarchy of folders, which group jobs by state (submitted, running, completed and so on). When monitoring is activated, job icons are moved automatically from one folder to another as the job state changes.
- The upper-right panel has content that is dependent on the active selection in the upper-left panel. When a folder is selected in the left panel, the right panel displays summary information for the jobs in the folder. When a folder is opened in the left panel, and a job inside the folder is selected, the right panel displays the job properties.
- The bottom panel hosts an embedded Python shell. This panel allows the user to interact with GANGA using CLIP commands; echoes

the command executed whenever an action is performed through the GUI; and displays any messages sent by GANGA to standard output and error.

- Most job-related operations can be performed using the toolbar, using the drop-down menus, or through appropriate selections in the right and left panels.

Job submission through the GUI involves creating a job of the required type (accomplished with a few mouse clicks), filling in application and backend properties in the form displayed in the upper-right panel, then clicking on the “Submit” button.

The main window in the prototype GUI is supplemented by other windows, depending on the task being performed. These additional windows allow data for analysis to be selected by querying the experiment catalogues; allow examination and modification of GAUDI/ATHENA options files using a specially developed Job-Options Editor (JOE), optimised for working with these kinds of file; and provide access to online help.

Further development of the GUI is being undertaken, to move beyond the prototype, taking into account feedback from trial users. The revised GUI will build on the functionality of the prototype, but will be both less cluttered and more dynamic, with use made of tabs and dockable windows.

5 Summary and outlook

GANGA has been developed in Python as a front-end for job definition and management, providing a single, easy-to-use interface for running and monitoring user-defined applications on a variety of local batch queues and Grid-based systems, hiding Grid technicalities. The main functionality is divided between an Application Manager, which deals with application setup, a Job Manager, which deals with job submission and monitoring, and an Archivist, which deals with job storage and workspace management. Functionality is made available through the GANGA Public Interface (GPI), and is accessed using the GANGA Client. The Client provides a Graphical User Interface (GUI), a Command-Line Interface in Python (CLIP), and the possibility to run GPI scripts.

GANGA specifically addresses the needs of the ATLAS and LHCb high-energy physics experiments for running applications performing large-scale data processing on globally distributed resources. As it handles different applications and backend processing systems using plug-in classes, which are easily added, and it offers significant configuration possibilities at startup, GANGA can readily be extended and customised to meet the requirements of other user communities.

Work in progress focuses on ensuring the availability of all plug-in classes required for the applications and backends of interest to ATLAS and LHCb, and on developing the GUI beyond the prototype stage. Trial users have been positive in their response to GANGA, which they have used successfully to submit jobs to both local and remote systems. Longer-term development efforts will be guided by feedback from an increasingly large user group.

Acknowledgments

We are pleased to acknowledge support for the work on GANGA from GridPP in the UK and from the ARDA group at CERN. GridPP is funded by the UK Particle Physics and Astronomy Research Council (PPARC). ARDA is part of the EGEE project, funded by the European Union under contract number INFSO-RI-508833.

References

- [1] <http://ganga.web.cern.ch/ganga/>
- [2] ATLAS Collaboration, *Atlas - Technical Proposal*, CERN/LHCC94-43 (1994); <http://atlas.web.cern.ch/Atlas/>
- [3] LHCb Collaboration, *LHCb - Technical Proposal*, CERN/LHCC98-4 (1998); <http://lhcb.web.cern.ch/lhcb/>
- [4] G. van Rossum and F. L. Drake, Jr. (eds.), *Python Reference Manual, Release 2.4.1* (Python Software Foundation, 2005); <http://www.python.org/>
- [5] LHC Study Group, *The LHC conceptual design report*, CERN/AC/95-05 (1995); <http://lhc-new-homepage.web.cern.ch/lhc-new-homepage/>
- [6] P. Mato, *GAUDI - Architecture design document*, LCHb-98-064 (1998); <http://proj-gaudi.web.cern.ch/proj-gaudi/welcome.html>
- [7] <http://atlasbkk1.in2p3.fr:8180/AMI/>
- [8] <http://lhcbdata.home.cern.ch/lhcbdata/bkk/>
- [9] <http://lcg.web.cern.ch/lcg/>
- [10] <http://egee-intranet.web.cern.ch/egee-intranet/gateway.html>
- [11] <http://www.opensciencegrid.org/>
- [12] <http://www.gridpp.ac.uk/>
- [13] <http://www.nordgrid.org/>
- [14] D. Adams et al., *DIAL - Distributed Interactive Analysis of Large datasets*, Proc. 2003 Conference for Computing in High Energy and Nuclear Physics, La Jolla, California, USA; <http://www.usatlas.bnl.gov/~dladams/dial/>
- [15] V. Garonne, A. Tsaregorodtsev and I. Stokes-Rees, *DIRAC: A Scalable Lightweight Architecture for High Throughput Computing*, Proc. 5th IEEE/ACM International Workshop on Grid Computing, Pittsburg, USA (2004); <http://lbnts2.cern.ch/>
- [16] <http://www.cs.wisc.edu/condor/>
- [17] <http://lhcb-comp.web.cern.ch/lhcb-comp/Analysis/>