THE GANGA USER INTERFACE FOR PHYSICS ANALYSIS ON DISTRIBUTED RESOURCES

A. Soroko, Department of Physics, University of Oxford, OX1 3RH, UK
C.L. Tan, School of Physics and Astronomy, University of Birmingham, B15 2TT, UK
D. Adams, BNL, Upton, NY 11973-5000, USA
K. Harrison, Cavendish Laboratory, University of Cambridge, CB3 0HE, UK
P. Charpentier, A. Maier, P. Mato, J.T. Moscicki, CERN, CH-1211 Geneva 23, Switzerland
U. Egede, J. Martyniak, Physics Department, Imperial College London, SW7 2AZ, UK
R. Jones, Department of Physics, University of Lancaster, LA1 4YB, UK
G.N. Patrick, Rutherford Appleton Laboratory, Chilton, Didcot, OX11 0QX, UK

Abstract

A physicist analysing data from the LHC experiments will have to deal with data and computing resources that are distributed across multiple locations and have different access methods. Ganga helps by providing a uniform high-level interface to the different low-level solutions for the required tasks, ranging from the specification of input data to the retrieval and post-processing of the output. For LHCb and ATLAS the goal is to assist in running jobs based on the Gaudi/Athena C++ framework. Ganga is written in python and presents the user with a single GUI rather than a set of different applications. It uses pluggable modules to interact with external tools for operations such as querying metadata catalogues, job configuration and job submission. At start-up, the user is presented with a list of templates for common analysis tasks, and information about ongoing tasks is stored from one invocation to the next. Ganga can also be used through a command line interface. This closely mirrors the functionality of the GUI, allowing easy transition from one to the other. This paper describes the Ganga design and functionality, and illustrates its use in the distributed analysis systems of the LHCb and ATLAS experiments in the context of their 2004 data challenges.

INTRODUCTION

Physics studies in ATLAS and LHCb will require analysis of the Petabytes of data that each will record at the Large Hadron Collider (LHC), under construction at CERN. The analyses will rely on computing resources distributed across many national and regional centres. These will be collected together and shared in a coordinated manner using grid services. The middleware for the grid services is provided by projects such as Globus [1] and LCG [2]. The experiments have developed their own services on top of the grid middleware, to extend the functionality, and to allow interaction with multiple distributed computing environments. Examples of these experiment-specific solutions include the DIRAC [3] workload-management system of LHCb and the DIAL [4] service that is a part of the ATLAS Distributed Analysis (ADA) system.

ATLAS and LHCb both use the Gaudi/Athena [5] C++ framework, designed to support all event-processing applications, including simulation, reconstruction and physics analysis. The similarity in the approaches of the two experiments has led to the setting up of Ganga [6] as a joint project aiming to provide a uniform front-end for managing framework-based jobs in a distributed environment.

Ganga assists the user in all phases of a job's life cycle. It helps in job creation, configuration, splitting, submission, monitoring, and output collection. It provides interfaces to experiment specific resources such as the metadata catalogue and bookkeeping database, which strongly facilitate physics analysis. Ganga does not provide its own grid services, but can use either generic grid services of LCG or the specialised services provided by DIRAC and DIAL. It can also use the computing resources of a local batch system or the user's PC.

In this paper we present the Ganga design and describe the functionality of different Ganga components and the user interface. We then explain how the user interacts with Ganga in order to execute an analysis job. In particular, we give an example of running an LHCb analysis job with the DIRAC system.

GANGA COMPONENTS

Ganga is implemented in python, an interpreted scripting language, using an object-oriented approach and following a component architecture, which benefits greatly from python's support for modular software development. The user has access to the functionality of Ganga components either through a Graphical User Interface (GUI) or through a Command-Line Interface (CLI) built upon a common Application-Programmer Interface (API).

The components of Ganga are connected via a so-called *software bus* [7], which in practical terms is a python module that follows a few non-intrusive conventions. The initial functionality of the software bus is provided by the python interpreter itself. In particular, the interpreter allows for the dynamic loading of modules, after which the bus can use introspection to bind method calls dynamically, and to manage components throughout their life cycle. The component-based approach has the advantages that it allows parallel code development, and that components from other systems with a similar

architecture can easily be incorporated into the main framework.

Ganga components fall into three categories: *core*, which define classes to support basic functionality; *specialised*, which address invocation of specific resources; and *external*, which heavily rely on third-party code and/or services. Both specialised and external components work like plug-ins and are not necessarily present in a given users Ganga installation. Below we describe the Ganga components in more detail.

Core Components

In the Ganga design, a distinction is made between a *job*, an *application*, and *data*. It allows running the same application on different computing systems with different types of file access. A job in Ganga terminology is what is actually submitted. An application is a computer program that the user wants to execute (the executable) together with any necessary configuration parameters, and data representing the input and output files for the application. Core Components provide generic interfaces for all of these objects, thus forming the Ganga design model.

A job-registry component provides the means to store and recover job information. It also contains important job metadata, like status, type of application, targeted batch system, timestamps etc. The job registry class has methods to initiate all job-related operations, including multithreaded monitoring. Essentially, these methods form an API used by the Ganga CLI and GUI.

Another component involved in the GUI construction is the abstract definition of a user interface. This component describes objects, which must be shown to the user, in terms of graphical widgets. As a result, the GUI for pluggable components can be built dynamically.

Core Components are not bound to any specific type of application, and in particular do not depend on the use of the Gaudi/Athena framework. They are self-consistent and may be used outside of Ganga to build other applications.

Specialised Components

In contrast to the core components, the specialised components serve the interests of various user groups and are very specific. Some of them such as Job, Application and File Handlers implement corresponding interfaces provided by the Core Components for different types of grid/batch systems, applications, and storage elements.

In particular, job-handling components manage job submission to LCG/EDG, DIRAC, DIAL, PBS, LSF, and local PCs. They perform job configuration by translating resource requests, for example minimum CPU time or minimum memory size, into the format expected by the target system, e.g., they create job description language (JDL) files for job submission to the grid. They also generate workflow scripts to be executed on the worker node, and issue specific commands as required for job monitoring, output retrieval and job termination.

Application-handling components provide configuration templates and cover specific tasks for a particular type of application. For example, the handler dealing with the LHCb analysis applications sets up the environment, discovers user DLLs and collects them for subsequent uploading to the worker node, creates configuration files, etc. Ganga has specialised support for ATLAS and LHCb applications based on the Gaudi/Athena framework. A handler for applications used in the BaBar experiment has also been implemented, and has been used successfully.

File-handling components transfer input and output files between worker nodes and different storage elements. They understand both logical and physical file names and support sandbox, gridftp, replica manager, and castor file-transfer commands.

Job splitters are scripts designed to create a collection of subjobs from a bulky initial job (macro job) for subsequent parallel submission and execution. Algorithms for job splitting may be different and vary among experiments. At the moment Ganga implements the simplest possibility, based on the splitting of a macro job's input dataset. In the future more complicated algorithms will be implemented and shipped with the Ganga distribution. Even now, though, experiments and/or experienced users may add their own job splitters to the script repository and the Ganga framework will automatically adopt them. In practice, to split a job, a user has to select the splitting script from the repository and possibly specify the desired number of subjobs. Ganga then does the rest. It creates and configures the subjobs and allows the user to submit all of them by submitting just the macro job. During monitoring, Ganga determines the status of the macro job according to the status of subjobs. Work is in progress to allow automatic merging, where desirable, of the outputs from the sub-jobs.

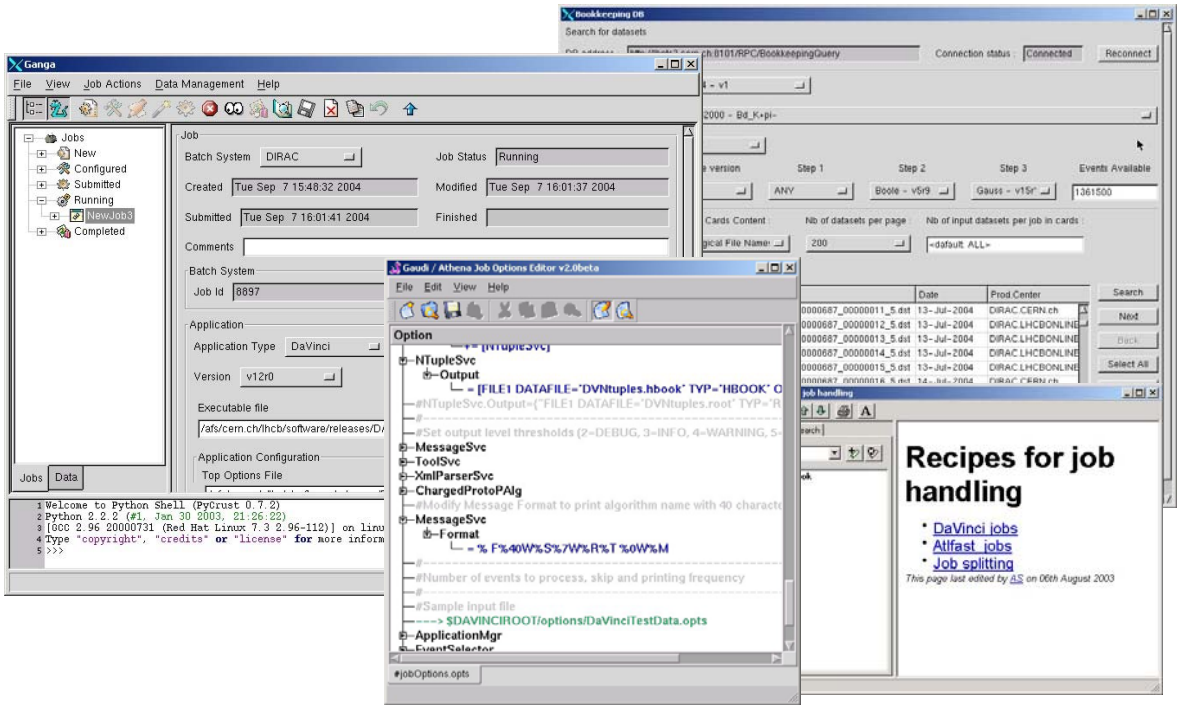

Figure 1: Screenshot showing the general layout of the Ganga GUI. The main window is to the left, the window for querying the LHCb metadata catalogue is to the right, the JOE window is in the centre, and the help window is shown to the bottom right.

A prominent role among the specialised components belongs to the graphical Job-Options Editor (JOE), which significantly decreases the difficulties in creating and modifying the options files used for application configuration within the Gaudi/Athena framework. A job-options file may be written as a text file, of may be in python, and may include any number of additional options files. JOE helps with the manipulation of options files by providing guidance on known options/values, using a database built by scanning and analysing the Gaudi/Athena release areas. It also reduces the likelihood of errors in syntax and spelling. The JOE GUI further helps the user by offering a collapsible tree representation, which allows options, attributes or included files of interest to be viewed on demand. Multiple job option files may be open for editing simultaneously using window tabsPreview of the job options file that is currently being edited in text format.

External Components

One of the first steps in a physics analysis is the selection of a dataset. ATLAS and LHCb both store information on datasets in metadata catalogues, although the catalogue contents, and the set of valid queries, for the two experiments are different. Ganga relies on external components to perform querying of these catalogues, and to allow the user-selected datasets to be added to an appropriate configuration file. An LHCb-specific component has been implemented, which additionally initiates automatic job splitting if the number of files in the selected dataset is larger than has been allowed per job. An ATLAS-specific component is at the prototype stage.

Another example of an external component is the AJDL job builder. AJDL is the Abstract Job Description Language used in the ADA system. The building blocks of an AJDL job are:

- Application, described by a name and a version number;
- Task, representing a collection of files required by the application;
- Dataset, which may be fetched from a catalogue;
- Preferences, which provide the means to fine-tune the way the job is processed.

The AJDL job builder has a prototype GUI, which allows the user to prepare an AJDL job easily. Work in progress will enable monitoring through Ganga of AJDL jobs submitted to ADA.

USER INTERFACES

Ganga provides both GUI and CLI interfaces for the main job-management tasks. Specialised and external components may bring their own GUI or such a GUI can be built dynamically based on the abstract definition of the user interface shipped with these components.

Ganga GUI

A general view of the Ganga GUI is presented in Fig. 1. The layout of the main window (on the left of the figure) consists of three parts: there is a tree control on the left that displays job folders, which themselves may be folders, in their respective states. There is a multi-purpose

panel on the right, which facilitates many displays, a list control in particular. Finally, there is an embedded python shell. With the advanced (expert) view option activated, the job folders, in the tree of job states, display a hierarchy of all job-related values and parameters. The most important values are brought to the top of the tree, and less important ones are hidden in the branches. The normal (user) view stops at the level of jobs and gives access to the most important parameters only. The user can also choose to hide the tree control completely. The list control displays the content of a selected folder in the job tree. The lower part of the frame is reserved for the python shell, which doubles as a log window. The shell allows for the execution of any python command, and also permits access to, and modification of, any GUI widget. The shell, too, can be hidden if desired.

Actions on jobs can be performed through a menu, using a toolbar, or via pop-up menus called by a right click in various locations.

When the monitoring service is switched on, jobs move automatically from one folder to another as their states evolve. To avoid delays in the program's response to the user input, the monitoring service runs in its own thread. It posts customised events whenever the state of a job changes and the display is to be updated.

Ganga CLI

An API provided by the job registry class may be used itself as a low-level command-line interface (CLI). However, a higher-level CLI is under development. This might be used to write splitting scripts, to record Ganga sessions, to build various test utilities, and by users who prefer a CLI to a GUI.

SUBMISSION OF ANALYSIS JOBS

The interaction between Ganga components may be illustrated by considering the handling of analysis jobs; submission of the LHCb analysis application, DaVinci, to the DIRAC workload management system is taken as an example. After having prepared and compiled the package(s) required for a given analysis, the user selects the package(s) and the DaVinci version number via the Ganga GUI or CLI. The algorithm workflow and algorithm parameters can then be edited using the Job-Options Editor. The component interfacing to the LHCb metadata catalogue can be invoked to select the dataset that is to be analysed. Depending on dataset selection and user settings, the job may be split automatically at this point, either by the standard splitting script or by a splitter supplied by the user. The user then presses the “Submit” button in the GUI main frame, and the monitoring component will update the job status as the job progresses. When the job completes, the user browses the job folder and collects the output.

A number of components are involved in the above process behind the scenes. It is the job-registry component that initiates creation of a new job object when the submission process starts. It then connects the appropriate application handler (DaVinci) and job handler (DIRAC) to this job object. These handlers do the bulk of the work. In particular, the application handler completes creation of the job options file and informs the job handler about any user-owned DLLs. The job handler in turn generates the JDL file and XML job-description file required by DIRAC. The job handler ships these files to the DIRAC system and, after receiving the job ID, periodically queries DIRAC for the job status. In the meantime, on arrival at the worker node, the workflow script may use the file-handling components to download missing files and to prepare the output sandbox. The job handler initiates the request for DIRAC services to download the job output automatically when the job status allows.

Submission of jobs to ADA is conceptually similar to the submission to DIRAC. The main differences are that the AJDL job builder is used for job configuration, and a different component is needed for metadata queries. There is also no need for the job splitters, as this is dealt with internally by the ADA system.

OUTLOOK

Ganga has been developed as a joint project between ATLAS and LHCb. Its modular design provides a uniform user interface to different distributed resources, and enables experiment customisation. The schedule of future releases includes plans for:

- High-level CLI (scripting);
- Interfaces to other analysis services (gLite);
- The building of remote Ganga services running under control of a local client;
- A GUI interface for the above client;
- Services to provide centralised persistency of user jobs;
- Ganga session persistency (history, GUI preferences);
- Roaming access to user profiles.

The Ganga project will then continue to keep pace with, and adapt to, the evolving grid middleware and experiment-specific services with the only intention to make life easier for users.

ACKNOWLEDGEMENTS

This work has been supported by the GridPP collaboration, and funding has been provided by the Particle Physics and Astronomy Research Council (PPARC).

REFERENCES

[1] http://www.globus.org
[2] http://lcg.web.cern.ch/LCG/
[3] http://dirac.cern.ch/
[4] http://www.usatlas.bnl.gov/~dladams/dial/
[5] http://cern.ch/Gaudi/
[6] http://ganga.web.cern.ch/ganga/
[7] http://wlav.home.cern.ch/wlav/pybus/