

Ganga 4 Architecture

D.L. Adams^a, U. Egede^b, K. Harrison^c, A. Maier^d, J.T. Mościcki^d, A. Soroko^e, C.L. Tan^f

^aBNL, ^bImperial College London, ^cUniversity of Cambridge, ^dCERN, ^eUniversity of Oxford, ^fUniversity of Birmingham

Created: January 10, 2005
Last modified: April 6, 2005

1 Introduction

Ganga is being developed for ATLAS and LHCb as a frontend for job definition and management. It needs to allow a user to specify the software to be run, which may include user-supplied algorithms; to set values for any configurable parameters; to select data for processing; and to submit jobs to a variety of backends, including local batch queues and Grid-based systems. After jobs are submitted, Ganga must monitor their status, and take care of saving any output. Ganga particularly aims to help with setting up jobs that run the main ATLAS and LHCb applications, all of which have been developed in the Gaudi C++ framework, and are configured by options files.

ATLAS and LHCb have fairly similar general requirements for Ganga, but there are significant differences in the details, for example:

- both run Gaudi-based applications, but ATLAS configuration is by Python options files whereas LHCb configuration is by options files with a value-assignment syntax similar to that of C++;
- the two experiments have different catalogues for dataset selection, with different possibilities for submitting queries;
- for ATLAS, Ganga must be able to submit jobs to the ATLAS Distributed Analysis (ADA) system, built around DIAL services, and must support the Abstract Job Definition Language (AJDL) that this system uses; for LHCb, Ganga needs to be able to submit to the DIRAC Workload Management System;
- being able to use Ganga to split a single large job into many smaller jobs has a high priority in LHCb, but not in ATLAS, where splitting is performed by DIAL services.

Work on Ganga has been in progress since 2002, and three major releases of Ganga software have been made (as well as a number of minor releases). Ganga release 3, made early in 2005, contains a lot of useful functionality: it allows definition of various types of ATLAS and LHCb jobs; it allows datasets to be selected via the ATLAS Metadata Interface (AMI) and the LHCb Bookkeeping Database; it allows submission to backends including PBS, LSF, LCG and DIRAC; it stores information about defined jobs, and monitors the progress of submitted jobs. Ganga 3 provides both a Command-Line Interface in Python (CLIP), which can be useful for scripting, and a Graphical User Interface (GUI), which simplifies basic tasks. There are, however, problems with Ganga 3: the way in which the user is allowed to interact with the system is sometimes cumbersome and overcomplicated; the system is a bit inflexible, with many code interdependencies, and extending the functionality, for example to be able to use ADA as a backend, tends to be difficult; there are various performance issues, leading to difficulties coping with large quantities of users, jobs and/or output data. To address these problems, while keeping the useful functionality, a revised architecture will be adopted for Ganga release 4, and much of the code will be rewritten.

This document presents the Ganga 4 architecture, and ideas for its implementation. General considerations are outlined in Section 2; the way in which the Ganga functionality is presented to the user is discussed in Section 3; then a detailed description of the architecture is given in Section 4.

It is noted that this document is intended first and foremost for the Ganga developers, and it should be viewed as a work in progress. Text that is greyed out relates to points that the developers feel are worth bearing in mind, but need further discussion.

2 General considerations

2.1 LHCb usage of Ganga

For LHCb the goal of Ganga is to assist in running jobs based on the Gaudi C++ framework. The physicist will use Ganga as a way to keep track of their analysis, much in the same way that we all use our email application to keep track our communication. The general way for a LHCb physicist to work with Ganga is illustrated in the following composite use case.

The user starts Ganga and as the first thing selects a small dataset for developing code on in the LHCb Bookkeeping database. The dataset is saved as a local template. The user creates a new job of type DaVinci and develops the C++ code outside Ganga. Using the Job Options Editor the job is configured and submitted as a local job. In a series of iterations the user copies the job and resubmits it first as a local job and then to the local batch system for slightly larger datasets.

The user is now ready to perform the analysis. The selection of data involves multiple selections in each of multiple large datasets to deal with differences in running conditions over time. All these selections are saved to ensure that jobs can be rerun over the same dataset. To keep things neat the user divides the analysis up into sub folders corresponding to global categories.

The analysis is now submitted by creating a small number of jobs corresponding to the global categories. The datasets to analyse are specified for the jobs, and a policy for splitting is defined. The jobs are then submitted to Dirac. Failed jobs are simply resubmitted as identical jobs again. Output files (typically in ROOT format) are merged when the complete analysis has finished.

The scale of LHCb analysis jobs are such that a master job is expected to get divided up into the order of 1000 sub jobs making a complete analysis contain less than 10000 individually submitted jobs to the distributed analysis system.

2.2 Some issues to be addressed

The proposed Architecture addresses the following issues (some of them arise from weak compatibility of the strategy of LHCb and ATLAS):

- management of state (native Ganga vs ADA Service);
- management of operations (direct vs ADA-based interaction with submission backends);
- semantics of certain abstractions: ADA-based applications have symbolic names, natively in Ganga applications have stronger connection to the physical executables;
- overlap of functionality: server/client side splitting, remote/local job registry;
- greater modularity including separation of client- server aspects;
- “special cases”: for example DIRAC is capable of handling DaVinci applications natively
- scalability:
 - number of simultaneous jobs per user: 1K typical, 10K max.

2.3 Strategy and Direction

- 2.3.1 Currently there is one-to-one mapping between handlers (implementation details) and the public interface (currently CLIP) objects which represent them (i.e. in the logical model). In Ganga 4 we need to have more fine-grained handler decomposition which will guarantee the required flexibility and some of the component of the handlers may be “invisible” in the public interface. For example application handler has two aspects: preparation step on the client and the script wrapper running on the WN. It may be desirable to separate them.
- 2.3.2 Limit the number of binary dependencies and select the pure-python packages if possible. Keep the Ganga client as lightweight to install as possible.
- 2.3.3 In case of certain remote Ganga components we may use the following strategy for the extension of the Ganga system by other people: we propose to implement the service which implements an existing protocol. If this is refused then we propose to implement a client-side component which should comply with internal APIs. It is not clear yet to which components this strategy may be applicable.

3 Ganga Public Interface – The Model

3.1 Complex Jobs / Splitting

- 3.1.1 Subjobs are objects which implement the job interface and they logically belong to the master job object:

```
j.subjobs[i]
```

This section is grayed out and kept for archival purposes. The current solution for client-side subjobs is given in the next section. Use case for client side subjobs must be reviewed and clarified with the experiments. Alternative solution involves the client splitting looking like server-side splitting.

Client side (Ganga) subjobs. A Ganga splitter generates a number of subjobs which are fully-functional Ganga jobs: they may be individually modified, copied (`j.subjobs[i].copy()` creates another subjob (outside of master job context), in the case of resubmission we use a special `resubmit()` method) etc. In other words the Master Job is a collection of regular jobs which live in the scope of the Master Job. Master Job is used to manipulate in bulk all the subjobs: for example `j.kill()` is a shortcut for “for `s` in `master.subjobs`: `s.kill()`”. Splitter takes a normal job and converts it into a Master Job by generating the subjobs. The output data directory hierarchy reflects the subjob structure. CLIP interface:

```
j = Job() # --> id = 1
```

```
len(j.subjobs) == 0
```

```
j.setMerger(merger) # pre-job merging
```

```
j.split(somesplitter)
```

```
len(j.subjobs) == N
```

```
j.subjobs[i] # --> id = 1.i
```

```
j.submit()
```

```
j2 = j.subjobs[i].copy # --> id = 2
```

one may also submit individual subjob without submitting the master job (for example for debugging purposes)

```
j.subjobs[i].submit()
```

```
j.subjobs[i].resubmit()
```

3.1.2 Server side (ADA, glite (optional)) subjobs. AJDL server or a submission backend generates a number of subjobs. These subjobs have no meaning outside of their Master Job context therefore Ganga will restrict a number of allowed operations on server side subjobs. It is not possible to configure the subjobs individually (because they start to exist only after the MasterJob has been submitted so they are read-only by definition), it may be not possible to copy the subjob (server however may resubmit the subjob so the subjob object is modified in place).

Job splitting may be done internally by Ganga (so called native splitting) which allows using subjobs even if the submission backend does not support subjobs directly. Such client-side Ganga subjobs appear as server side subjobs in the GPI.

```
j = Job()
# true server side splitting
j.backend.splitter = somesplitter
# -- OR -- Ganga native splitting
j.splitter = somesplitter
len(j.subjobs) == 0
j.submit()
len(j.subjobs) == N
j.subjobs[i].kill() # OK
j.subjobs[i].copy() # --> ERROR
j.subjobs[i].submit() # --> ERROR
j.subjobs[i].resubmit() # OK
```

Another possible syntax to interact with subjobs may be:

```
j[i]
j.kill(i)
j.resubmit(i)
```

3.2 Logical Schemas

Ganga logical model describes the objects and their properties/operations in the public interface (GPI). GPI is extensible in the sense that each type of application or backend may define their own set of properties visible to the user (aka GPI ContainersOfProperties).

3.2.1 Generic Application

3.2.1.1 executable : <string>, name of an executable

3.2.1.2 parameters : <list of strings>, arguments to the executable

3.2.2 Job

3.2.2.1 id : <string>, <readonly>, unique job identifier

3.2.2.2 status: <string>, <readonly>, current status of the job, getting value of this property may internally trigger a forced retrieval of the monitoring information (contacting the backend for latest job status), see Job Monitoring below

3.2.2.3 name : <string> optional (non-unique) name

3.2.2.4 folder : <string>, <readonly> a name of the folder (logical tree) in which the job will be visible. Alternative name for this property is 'path'.

3.2.2.5 application : <application>

3.2.2.6 backend: <backend>

3.2.2.7 inputsandbox : <list of File objects or filenames> – additional files to be copied to the input sandbox. Default semantics: '/x' is copied to WN

and accessible as 'x' (i.e. in cwd of a job), 'lfn:/y' is copied to WN and accessible as 'y' In case of splitting each subjob will receive identical copy of the sandbox.

3.2.2.8 `outputsandbox`: <list of File objects or filenames> – a declaration of which files are produced by the job and a specification where to copy them when the job is finished. If files are declared but not produced by the job it is an runtime error. Default semantics: 'x' <=> file x is produced on WN and copied to default output sandbox location of a job as file 'x'. '/y' <=> file '/y' is produced on the WN and copied to default output sandbox location of a job as file 'y'. 'lfn:/z' <=> job produced a file 'lfn:/z' in a file catalog and this file is copied to the default output sandbox location as file 'z'

3.2.2.9 `merge(merger)` – merge output if the job is finished, if a job is new then this method will enable automatic merging at the end of the job. To disable automatic merging: `merge(None)`

3.2.2.10 `splitter` – splitter for the job. Splitting will be done at job submission time and a number of subjobs generated.

3.2.2.11 `submit()`

3.2.2.12 `resubmit()`

3.2.2.13 `kill()`

3.2.3 Any Gaudi-based application (e.g. DaVinci)

3.2.3.1 `version`: <string>

3.2.3.2 `inputdata` : <dataset> , an input dataset. This may be a list of files (with their locations). In case of splitting the subjobs may receive parts of the input dataset. It also affect the brokering process on the GRID: since by default input data is not local the job may be sent to a CE with best access to the inputdata. Default semantics for datasets specified as list of files: '/x' <=> file '/x' is opened on the WN, 'lfn:/y' <=> file 'lfn:/y' is opened on the WN. It is not yet clear if this property belongs to the application or to the job.

3.2.3.3 `outputdata` : <dataset>, a directive for a job to produce certain files which are then typically left on some (L)SE and not copied to the output sandbox location. Default semantics: '/x' <=> produce file '/x' on the WN and leave it there, 'lfn:/x' <=> produce file 'lfn:/x' in the file catalog. Comment: in the context of certain predefined applications Ganga has control over the output data produced. However in case of generic application Ganga has no control over which files are produced so `outputdata` cannot be specified. It is not yet clear if this property belongs to the application or to the job.

3.2.3.4 `available_versions` : <list of strings>, <readonly>, <transient>, list of available versions of a given application, internally this is a call to the `ApplicationManager` and I have certain doubts if this is a way we want a user to see this. It is rather like a class property (method) then the object property (method).

3.2.3.5 `optsfile` : <string> : implicit sharing of files should perhaps be part of the semantics of this property; see page 9

3.2.3.6 `cmt.releasepath` : <string>, main cmt release area

3.2.3.7 `cmt.userpath` : <string>, user cmt area

3.2.3.8 `platform` : <string>

3.2.3.9 `masterpackage` : <string> package with cmt requirements file

3.2.4 Athena Application

3.2.5 LSF Submission Backend

3.2.5.1 queue

3.2.6 Glite Submission Backend

3.2.6.1 splitmode – server side splitting mode

3.2.7 DIRAC Submission Backend

3.2.8 LHCb Dataset (if decided that we need them)

3.2.9 ADA Dataset (if decided that we need them)

3.2.10 Metaproperties of any GPI object

3.2.10.1 `_schemaVersion` : <persistent>, <readonly>, <meta> version of the schema of the GPI object

3.2.11 INTERNAL OBJECT: installation

3.2.11.1.1 “default” (defined by a submission backend)

3.2.11.1.2 SharedFilesystem

3.2.11.1.2.1 path

3.2.11.1.3 ARDAInstaller

3.3 Job Registry and Job Identification

3.3.1 Job Identifiers

Job Id is a positive integer number represented by a string “j” (currently `sys.maxint` is 2147483647). A subjob id is represented by a string “j.i” where i is a positive integer number. Registries have GUID identifiers. A combination of (regid,jobid) is unique.

3.3.2 Backend Job Identifier

Each job ultimately is run on some submission backend and gets and gets an identifier specific to that backend (for example: process id for local jobs, LSFid for LSF, etc). The backend job identifier is DIFFERENT from Ganga job identifier and is not the primary mechanism for organizing and finding jobs in Ganga).

3.3.3 In the same Ganga session there may be more than one registry object (for example: local and remote job registry)

>>> `jobs()` # return the default job registry

3.3.4 JobTemplates are registered within their own registry.

>>> `templates()` # return the default template registry

3.3.5 Remote registry supports the concept of softlock (the same way PINE closes imap INBOX to avoid multiple writers)

3.3.6 It is well defined within a single Ganga session when Client is a writer (jobs in 'new' state) and when JobManager is a writer (jobs in other states).

3.3.7 Registry stores the job configuration and it does not store any files.

However there is a default LSE for the registry which may be used to store the input files.

3.3.8 Implementation remarks:

3.3.8.1 Local registry uses a local file system as the LSE. Probably local registry should be implemented as a lightweight server writing and reading files so that concurrent Ganga sessions may be run safely.

3.3.8.2 Remote registry should be implemented as a server with some database backend. Initial implementation may be done with ARDA's lightweight metadata server. Other possibilities: sqllite, anygui, berkeley DB. Question: how to implement a LSE? With the same server (binary db blob) or raw filesystem?

3.4 Job Monitoring

3.4.1 In order to make job status update scalable the periodic updated combined with caching of job status should be implemented. Example of automatic monitoring: every 30 s glite backend is queried for job status.

This may scale badly with the number of users so the period may be longer.

However automatic monitoring is not sufficient. An explicit GPI call `j.status` should give more up-to-date information than the possibly long automatic monitoring period. However it is not scalable either that each call of `j.status` forces a query to glite (this will not work for a GPI loop). The technique of minimal forced monitoring period will make sure that, if asked explicitly, the status information will be not older than, say 5 seconds. So a rapid series of `j.status` calls will not trigger the monitoring update more frequently than every 5 seconds. The exact length of the period must be chosen experimentally. Each backend may have individual monitoring period lengths.

- 3.4.2 Local/LSF monitoring techniques: worker node and client share the same filesystem. Ganga may take advantage of this and “communicate” the termination of jobs via the filesystem.
- 3.4.3 Client vs Remote Job Manager. Similar force monitoring period technique may be used by the Client to make sure that a remote call to Job Manager will scale for GPI loops.
- 3.4.4 An implementation possibility in case of Remote Job Manager: if job manager is a server then job wrappers may try to contact the job manager to notify about job termination.

3.5 Job Folders

- 3.5.1 Folders provide a way of organizing the jobs into hierarchical structure resembling the directories in the file system (on top of the flat list of jobs in the file registry):
`>>> folders() # give the root folder of the default registry`
- 3.5.2 This mechanism is fully optional: jobs do not have to belong to any folder.
- 3.5.3 User may easily navigate the folders and list their contents. There is also a shortcut to retrieve job objects directly via folders:
`>>> folders.cd('\myjobs')['large job']`

3.6 File Management

- 3.6.1 Job submission involves operations on files such as making input files available on a worker node and making output files available to the Ganga Client.
- 3.6.2 Files in the context of certain operation (input/output) is described as follows:
 - 3.6.2.1 source : `<string>` this is a source filename before performing the operation
 - 3.6.2.2 dest : `<string>` this is a destination filename after performing the operation
 - 3.6.2.3 other information, such as protocol or some optimization hints.
- 3.6.3 In the context of an operation there are some defaults if file is specified as a string. See input/output sandbox and input/output data in the logical schema description.
- 3.6.4 Local files
There is not implicit caching of the files (filenames behave as 'references')
`j.inputsandbox = "/a/b/c"`
`j.submit()`
`# modify the contents of 'a/b/c'`
`j2 = j.copy()`
`j2.submit() # use the MODIFIED file`
- 3.6.5 Remote files and LSEs
 - 3.6.5.1 Remote files live in remote LSEs (Lightweight Storage Elements). LSE

in this context is a *lightweight* implementation of a file catalog combined with a protocol of putting and getting files. NOT to be confused with Grid SE which is a complex infrastructure to store large amount of data. LSE is NOT intended to keep large data files.

3.6.5.2 Remote file may be easily shared because LSEs are by definition accessible from all places so files may be referenced with the syntax similar to "lse://sename/filepath". In case of WNs on the Grid which cannot access certain LSEs, Ganga will internally copy relevant files into the Grid-specific SEs or put them into local sandbox.

3.6.6 Sharing/Protecting local files.

If you want to make sure that you rerun a job using a local file with the same contents you must first copy the file to some LSE. For convenience Ganga provides a default LSE for each of the job registry and offers convenient shortcuts at the GPI level to do this. For example it may look like this:

```
f = File("/a/b/c",shared=1)
```

```
j.inputsandbox = f
```

If it does not already exist the file "/a/b/c" is copied to the default LSE for sharing. Each time object f is used it will use the remote copy. The remote copy may be easily accessed with a path as well:

```
j2.inputfile = "shared://a/b/c"
```

This also responds to the LHCb use cases 8.b and 8.c

3.6.7 Internally the input sandbox may be generated on the fly (for example flattened job options file) and is not stored on the LSE (at least it is not visible to the user; storing on the LSE may be done as an implementation detail for caching).

3.6.8 Retrieving output files. User may request to store the output files on some LSE or to transfer the output files automatically to the local filesystem (local LSE). The job output is stored in a directory hierarchy which reflects the organization of jobs as they are seen in GPI (for example subjob output is in the subdirectories. This might reflect the subfolders structure (LHCb use case 4h)).

3.6.9 Maybe the default behaviour for handling output files could be like in Condor or LSF: get all files back in the output sandbox, or register them in the catalog. Possibly this may be part of Ganga configuration file.

3.7 Plugins

Plugin contains: logical schema, internal handlers, GUI/CLIP extensions.

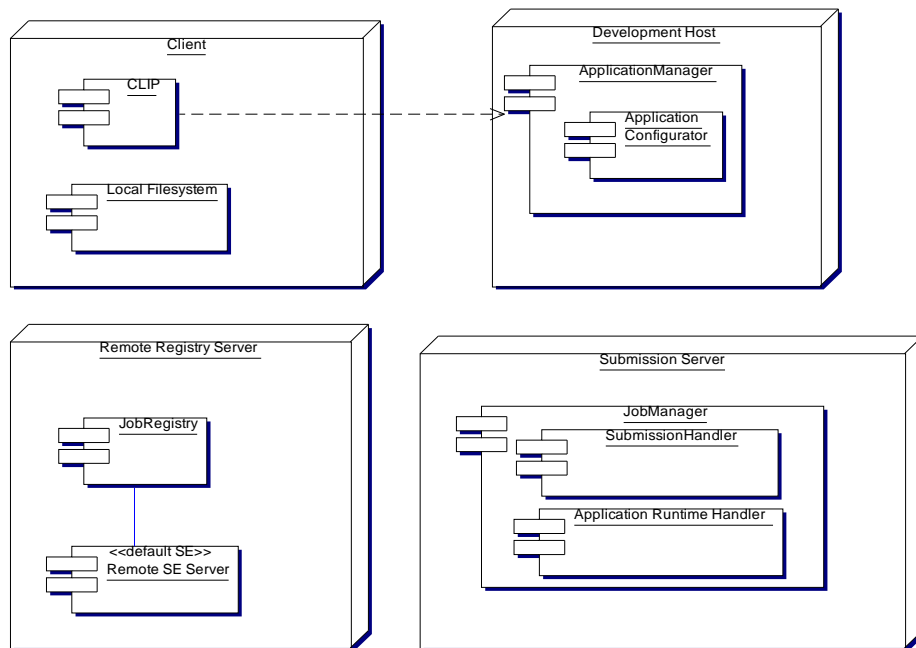
[More detail needed here.](#)

4 Internal Architecture

4.1 Overview of Functional Decomposition

This is the overview of the functional decomposition of Ganga which at the same time describes the maximal splitting of Ganga software components in a distributed environment. ApplicationManager and JobManager provide interface to be shared by both ATLAS and LHCb. In the two cases their internal implementation may be different. The section describing Core Domain Handlers is a proposal for LHCb implementation.

4.1.1 Client



Drawing 1 Fully Distributed Deployment Model

This is the host which runs Ganga interface (CLIP, GUI and GPI) and with which the user interacts directly. Job configuration is kept in the job registry which may be remote (Remote Registry Server with a default Remote LSE). Job submission is handled by JobManager, application preparation is handled by ApplicationManager.

4.1.2 Development Host (Application Host)

This is the host in which the application preparation happens. ApplicationManager is responsible for configuring the application on that host and generate additional files for input sandbox if necessary. For example the job options file gets flattened using environment produced by cmt. Some temporary files for the input sandbox may be produced by the Development Host and shipped to the Submission Host. Typically Development Host is identical to the Client. This host may run a ADA Analysis Service for ATLAS use cases.

4.1.3 Submission Host

This is the host which does the submission to the submission backends. The submission handlers are run here as well as the monitoring loops. Job Manager internally matches the correct combination of SubmissionHandler and ApplicationRuntimeHandler to correctly submit and monitor the job. Job Manager may have a LSE cache for the files of Client's localfilesystem and for the input sandbox files. This host may run an ADA Analysis Service for ATLAS use cases.

4.1.4 Remote Registry Server

Remote Registry Server is a "passive" data store (typically it has a database backend).

In case of ATLAS the ADA Analysis Service performs the function of the Remote Registry Server.

4.2 Vertical Overview – Layers

4.2.1 GUI

Uses GPI to access functionality. GUI registers as an observer of core objects

to monitor the changes of state which may happen outside of GUI framework (e.g. Monitoring thread or interactive python prompt)

4.2.3 CLIP – Command Line Interface in Python

This package contains extensions to GPI to better support the interactive command prompt. CLIP does not define any new functionality wrt the GPI. CLIP is kept as close to GPI as possible. CLIP may provide very high-level shortcuts, property aliasing mechanisms on top of GPI etc.

4.2.4 GPI – Ganga Public Interface

This is the primary scripting interface for public use. GPI provides a python representation of the logical Ganga model and ways to manipulate jobs, datasets etc. Certain fractions of GPI are automatically generated from logical description of the model (so called schema) which is provided in the plugins which define concrete applications, backends and datasets. GPI propagates the operations and state changes to the underlying Core Objects Layer. GPI objects internally support very high level state and behaviour management.

4.2.5.1 State Management: ContainersOfProperties

Job/Application configuration is persistently saved in a JobRegistry. ContainersOfProperties make sure that the state of the GPI objects is well-managed. If some other management of state is required this layer may be replaced without impact on the rest of the system. Its

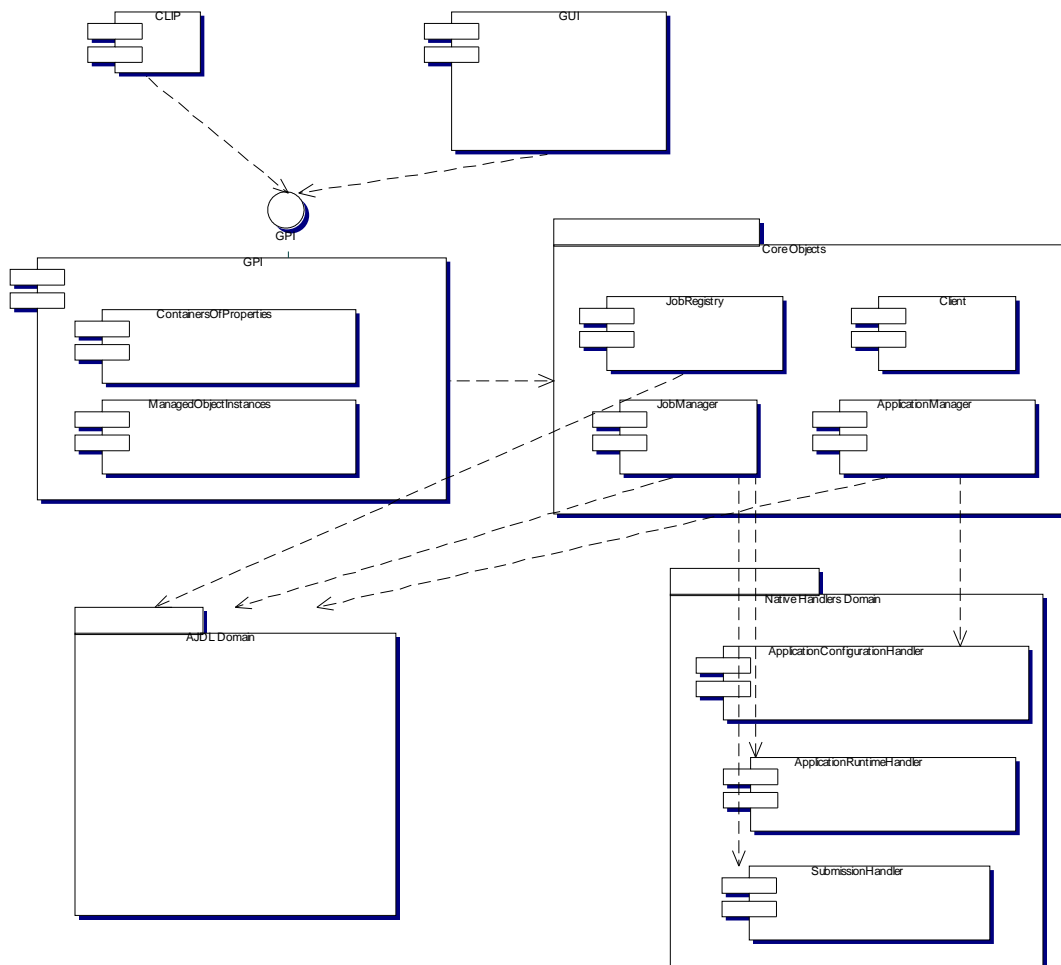


Illustration 1 Package dependency. Hopefully AJDL Domain will not need a separate implementation and ATLAS specific handlers may be implemented in terms of Native Handlers Domain.

implementation provides transaction safety and flushing of memory buffers.

4.2.5.2 Framework Behaviour Management: ManagedObjectInstances

Ganga framework is a state machine with well defined interactions between various Core Objects. ManagedObjectInstances are the 'driver' of this high level state machine.

4.2.6 Core Objects (NOT part of GPI)

Core Objects are not exposed in a public interface. Core Objects provide the internal interface for lower-level state and behaviour management in Ganga. Depending on the Ganga deployment scenario and experiment context (ATLAS/LHCb) there may be different implementations of Core Objects (for example local job manager versus a proxy to the remote job manager). Core Objects may be internally implemented in a completely different way (for example AJDL Services) but the basic state machine is always the same (as defined by GPI objects). Core Objects correspond to the basic deployment units. It does not make sense to distribute Ganga in more independent pieces than it is implied by the Core Objects.

4.2.6.1 JobManager – matches correctly instances of

ApplicationRuntimeHandler and SubmissionHandler and runs the monitoring loops. There is a callback mechanism which provides (optimized by buffering techniques) notification of certain events which are received by the Client. It also performs the validation of Job Configuration:

- 4.2.6.1.1 Before submission the job configuration is validated to detect disallowed combinations of handlers. Certain job configuration errors may be reported by the submission backend only

4.2.7 Handlers

Handlers are internal implementation details of the Core Objects. In certain cases the entire handler “domain” may be substituted by software developed completely outside of Ganga context (for example AJDL service may have a private internal mechanism for handling of the jobs). In this case the “AJDL Domain” could provide the full replacement for the default implementation provided by “Ganga Default Domain”. Hopefully this shall not be needed.

4.2.7.1 Internal implementation of the Core Objects. A typical example for LHCb: ApplicationConfigurator is used by ApplicationManager for configuration of the applications on the Development Host.

SubmissionHandler and ApplicationRuntimeHandler are used by JobManager to organize correctly the submission and monitoring of jobs.

4.3 Details of Functional Decomposition

Responsibility and functionality of main functional components of Ganga. The list is not in any particular order.

4.3.1 Ganga Client

4.3.1.1 Thin client, possibly pure Python.

4.3.1.2 Initially sets up the job.

4.3.1.3 Performs client side job splitting.

4.3.1.4 Client side split jobs are fully qualified jobs. They can be manipulated in the same way as a standard job

4.3.1.5 Receives information on server side splitting from the Job Manager

4.3.1.6 Then goes to the registry to get information on the subjobs

4.3.1.7 Server side subjobs are limited in manipulation depending on the features supported by the backend

- 4.3.1.8 Talks to the book keeping and/or replica manager to retrieve data sets.
- 4.3.1.9 Enables the editing of job options.
- 4.3.1.10 Registers job in a registry. The registry can be local or remote.
- 4.3.1.11 Communicates with the Application Manager to retrieve:
 - 4.3.1.11.1 Available applications.
 - 4.3.1.11.2 Available versions.
 - 4.3.1.11.3 Available platforms.
 - 4.3.1.11.4 Compiled user code.
 - 4.3.1.11.5 Pre-processed job- parameters (“flattened option files”).
- 4.3.1.12 Communicates with the Application manger to send:
 - 4.3.1.12.1 Application relevant parameters for manipulation (e.g., option files).
 - 4.3.1.12.2 User code to be compiled in the context of the chosen application (optional)
 - 4.3.1.12.3 Receives the information preprocessed run- options and compiled user code (shared libraries).
- 4.3.1.13 Communicates with a registry service:
 - 4.3.1.13.1 Several registry “accounts” are possible, which may be local or remote.
 - 4.3.1.13.2 Registers jobs in the remote registry.
 - 4.3.1.13.3 From creation of a job until submission. The client has write permission to the registry. Once the job is submitted, the job status is managed by the Job Manager.
- 4.3.1.14 Communicates with the Job Manger
 - 4.3.1.14.1 Submits a configured job to the Job Manager
 - 4.3.1.14.2 Sends kill and delete commands to the Job Manager
 - 4.3.1.14.3 Receives the the status from the Job Manager
 - 4.3.1.14.4 Once the job is submitted, the client is only the reader of the job status

4.3.2 Application Manger

- 4.3.2.1 Informs the client on available application it knows about. In case of DIRAC or DIAL it may be desirable to find out what versions/applications may be run on the backend. This functionality is probably best placed in the ApplicationRuntimeHandler (i.e. within JobManager).
- 4.3.2.2 Accepts job configuration requests from the client
- 4.3.2.3 Sets up the environment for configuring the application
- 4.3.2.4 Processes the user options
- 4.3.2.5 Compiles user code (optional)

4.3.3 Job Manager

- 4.3.3.1 Accepts a job from the client
- 4.3.3.2 Has knowledge of the supported backends.
- 4.3.3.3 Creates the wrapper script in order to run the job on the backend.
- 4.3.3.4 Modifies the job the information of the job output depending on the knowledge of the what the application supports and on what the backend supports.
- 4.3.3.5 E.g. the Job Manager may decide to change the output location of an output files specified in the output- data to be local and then copied to the final location.
- 4.3.3.6 Manipulates jobs on behalf of the client (kill, resubmit, delete)
- 4.3.3.7 Informs the client on available submission backends
- 4.3.3.8 Submits the job on behalf of the client

4.3.3.9 In case of server side splitting:

4.3.3.9.1 Informs the client on the success of server side splitting

4.3.3.9.2 Informs the registry on the number and status of the the sub jobs

4.3.3.10 Informs the client and the registry on the status of jobs

4.3.3.11 Informs the registry on the output- data (Ntuples, hbook) and the output location (all files in the sandbox. Keeps track of stdout and stderr.

4.3.4 Remote registry

4.3.4.1 Keeps track of Ganga jobs

4.3.4.2 Receives job configuration from client for new jobs

4.3.4.2.1 Job object

4.3.4.2.2 Input sandbox

4.3.4.2.3 Receives and stores information of the output location (sandbox) and the output- data.

4.3.4.2.4 For submitted job receives the job status from the Job Manager

4.3.4.2.5 Informs client on status of jobs

4.3.4.2.6 Allows client to receive output files and output data for finished jobs.

4.4 Core Objects Interfaces

4.4.1 Some supported concepts

4.4.1.1 Bulk operations

Core Objects should support bulk operations (like bulk job submission, bulk job data retrieval etc)...

4.4.1.2 Notifications

Event notifications are part of the Core Object Interface. Notifications may extended in some cases to cover additional requirements such as feedback on the dataset contents.

4.4.2 ApplicationManager

4.4.2.1 list_applications(): <list of strings>

Get a list of all available applications known to the ApplicationManager. This list corresponds to the list of available and loaded plugins in the DevelopmentHost.

4.4.2.2 list_defaults(appname) : < { proptime:value} >

Get a dictionary of default values of the all properties of a certain application.

4.4.2.3 list_choices(appname,proptime): <list of strings>

Get a list of all possible values of a certain property (for example list all available versions of DaVinci).

4.4.2.4 preprocess(jobdata,proptime): <PreprocessedPropertyValue>

Get an internal object which represents the preprocessed value of certain property. For example: expanded job options file is used by JOE on the client side.

4.4.2.5 configure(jobdata): <ExtraJobInfo>

Preprocess all values related to the application and put them into ExtraJobInfo structure. This ExtraJobInfo structure is application specific so the primary consumer of this information is

ApplicationRuntimeHandler (part of JobManager). Example: in case of DaVinci ExtraJobInfo contains extra input files to the sandbox (e.g.

flattened job options file, additional application wrapper script, etc). It is a matter of internal implementation of ExtraJobInfo if the files are copied by value via the network or if they are put on the shared filesystem (e.g. LSE).

4.4.3 JobManager

4.4.3.1 list_backends(): <list of strings>

4.4.3.2 list_choices(backendname,propname): <list of strings>

4.4.3.3 list_defaults(backendname) : < { propName:value} >

See above.

4.4.3.5 submit(jobdata,ExtraJobInfo,outputmode) : <OK status>

Submit the job together with any ExtraJobInfo produced by the ApplicationManager. When the job is finished the Client is notified by the client.notify() method. "outputmode" specifies what JobManager should do with the output of a job when it terminates:

"cache" -- cache the job output (in the local LSE), client must getoutput() explicitly at some later time

"send_with_notify" -- send the output_sandbox by value with the notify()

"upload" -- upload the output sandbox to the LSE as specified in the jobdata.output_location

4.4.3.6 kill(jobid)

4.4.3.7 resubmit(jobid)

4.4.3.8 getoutput(jobid) : <FileObject>

Get a previously cached output sandbox as a string. This possibility is reserved for a local client LSE which cannot be run in a server mode because it is just a plain filesystem.

4.4.4 Client

4.4.4.1 notify(jobid, what, value) : <void>

Notify that a job has changed state due to some event on the submission server. For example: job has finished or job has been split. Arguments may be:

what == 'status' or 'subjobs'

value == new status or list of subjob ids

4.4.5 RegistryServer

4.4.5.1 Concurrent r/w access to the registry

4.4.5.1.1 Client has write access to jobs which are in 'new' state.

4.4.5.1.2 JobManager has write access to all other jobs.

4.4.5.1.3 A transition from the 'new' state must be well defined because it implies the transfer of the write access between concurrent (distributed) processes (Client/SubmissionServer). Note that we have only ONE client and ONE submission server per Ganga Session. If you want to have multiple clients then they all must be serialized via one Client process which acts as a proxy for them.

4.4.5.2 Registry

4.4.5.2.1 createJob(jobdata)

4.4.5.2.2 removeJob(jobid)

4.4.5.2.3 updateJob(jobdata)

4.4.5.3 LSE

4.4.5.3.1 download_file(source,dest)

4.4.5.3.2 upload_file(source,dest)

4.4.5.3.3 upload_job_output(source,jobid,topdir=default)

4.4.5.3.4 download_job_output(jobid,dest,topdir=default)

4.5 Design of Core Objects / Servers

This section describes the internal design of CoreObjects from the perspective of processes, threads and the synchronization with distributed peers etc.

4.5.1 We identified the need of caching and emulation of distributed callbacks if this is imposed by the underlying networking protocol or transient

network problems. "Caching" problem: JobManager needs to upload the sandbox to the LSE when the job is completed. LSE server is down. The JobManager needs to cache the output and try to upload it later. "Emulating callbacks" problem: we would not like to brake the standard Client/Server networking model, i.e. Ganga Client should not listen on sockets and accept incoming connections. Our model assumes event notifications and talking back to the client (or local LSE on a Client machine). In some cases we can do the real distributed callback (for example reusing the TCP/IP connection initiated by the Client). In some cases the callback must be emulated with an internal "poll" in a Client to fetch the status of the server. We want to hide this (if possible) using some sort of proxy mechanism.

4.5.2 Client

Transfer of the writing rights between client and job manager: making sure that a multithreaded client and a submission server do not step each other's toes while writing to the same registry.

4.5.2.1 j.submit()

```
registry_writers_mutex.lock() # no flushing of registry cache to the DB
will happen until mutex released
try:
  if appmgr.configure(j) is not success or connection dropped:
    job_configuration_error(j), status 'new' (unchanged)

  if jobmgr.submit(j,extra) is not success:
    job_manager_did_not_accept_the_job(j), status 'new'
    (unchanged)

  if connection dropped:
    status 'unknown' (real status may be 'submitting' or 'new')
    will try to update the status of 'unknown' jobs later, for the
    moment the job is read-only

  if success:
    status = 'submitting' (read-only)
    return OK
finally:
  registry_writers_mutex.release()
# even if some thread modified the state of j in the cache
# the changes will not be flushed into the database because the job is
readonly, they will be simply lost (with a possible warning to the user)
```

4.5.2.8 cache flushing thread on the client

```
registry_writers_mutex.lock() # no submit possible during cache flush
try:
  jobs = filter(not SUBMITTED,all_jobs)
  # update needs to make sure IN THE SAME TRANSACTION that this
  client still has a softlock, otherwise rollback
  db.update(jobs)
  if rollback:
    read_only_registry = 1
    raise AccessToRegistryNowReadOnly()
```

```
finally:  
    registry_writers_mutex.release()
```

4.5.2.13 cache update thread on the client

```
read_only_jobs = db.query('SELECT jobdata FROM jobs WHERE status != "NEW"')  
all_jobs.update(read_only_jobs)
```

4.5.2.14 job manipulation (j.x = y):

```
if read_only_registry:  
    raise RegistryIsReadOnly()  
if j.state not new:  
    raise CannotModifyJobInAnotherState()  
else: set(j,x,y)
```

4.5.2.16 client startup

```
imported_jobs = db.query('SELECT jobdata from jobs')  
# create a cache  
all_jobs.update(imported_jobs)  
# take over a softlock from another client which may be running  
ok = db.update(softlock->THIS_CLIENT)  
if not ok: read_only_registry = 1
```

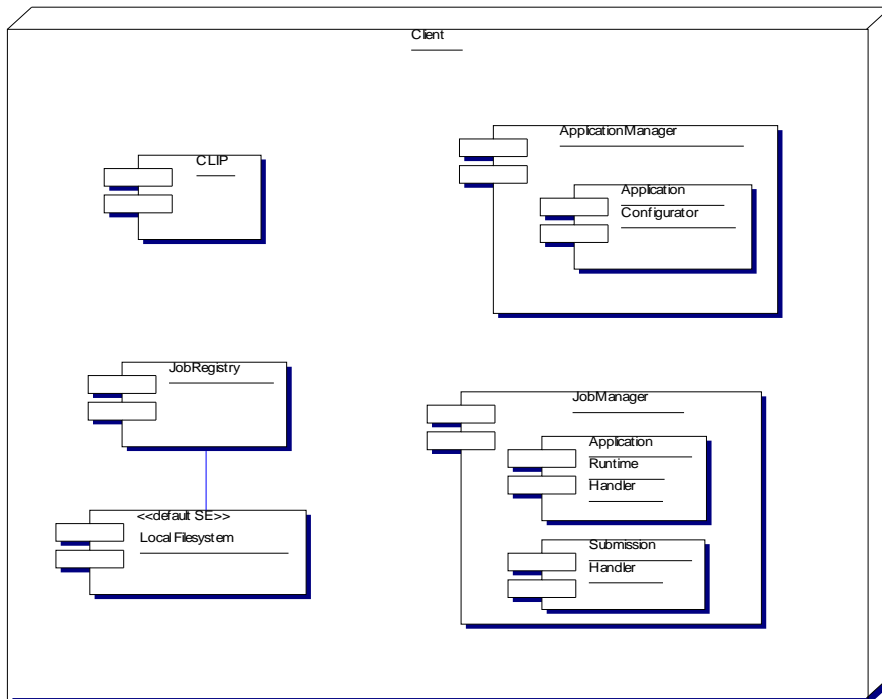
4.6 Handlers Details

Handlers are internal implementation details of Ganga. Their knowledge may be necessary to write a correct plugin for a new type of backend or application. This is a proposal for LHCb. In the case of ATLAS this may be internally implemented by ADA Service in another way.

4.6.1 Application related

4.6.1.1 ApplicationConfigurator – prepares the correct input sandbox in the SubmitterHost. For local and LSF submission the SubmitterHost is typically the client machine. For ADA the submitter host is the DIAL server.

4.6.1.2 ApplicationRuntimeHandler – generates a wrapper script which makes sure that the application is installed on a WN and the environment is setup correctly. It also does necessary corrections to input sandbox files (e.g. optsfile) to assure correct file management (output data and output sandbox).



Drawing 2 Standalone Ganga Application (all in one)

4.6.1.3 SubmissionHandler – provides an interface to submit, kill and monitor jobs in a given submission backend. It also generates the main job wrapper script if necessary (job wrapper embeds the application wrapper in the most general case).

4.7 Details of GPI Internal Design

This section lays down the internal design of the GPI/Ganga4 based on CLIP/Ganga3. This section needs the revision of the vocabulary to avoid misunderstandings.

In order to reach the desired level of flexibility the Architecture consists of the following layers (drawing 3 shows the vertical layout based on current CLIP implementation):

4.7.1 Ganga Public Interface

GPI represents the logical Ganga job model and provides a public interface to interact with Ganga Core. Internally it uses a generic tree-based mechanism with ContainerOfProperties and ManagedObjectInstances to manage the state of the logical job model as well as the very high-level framework behaviour from the point of view of the Client (by proxying the desired behaviour to the underlying Client Core Object).

4.7.2 Core Objects

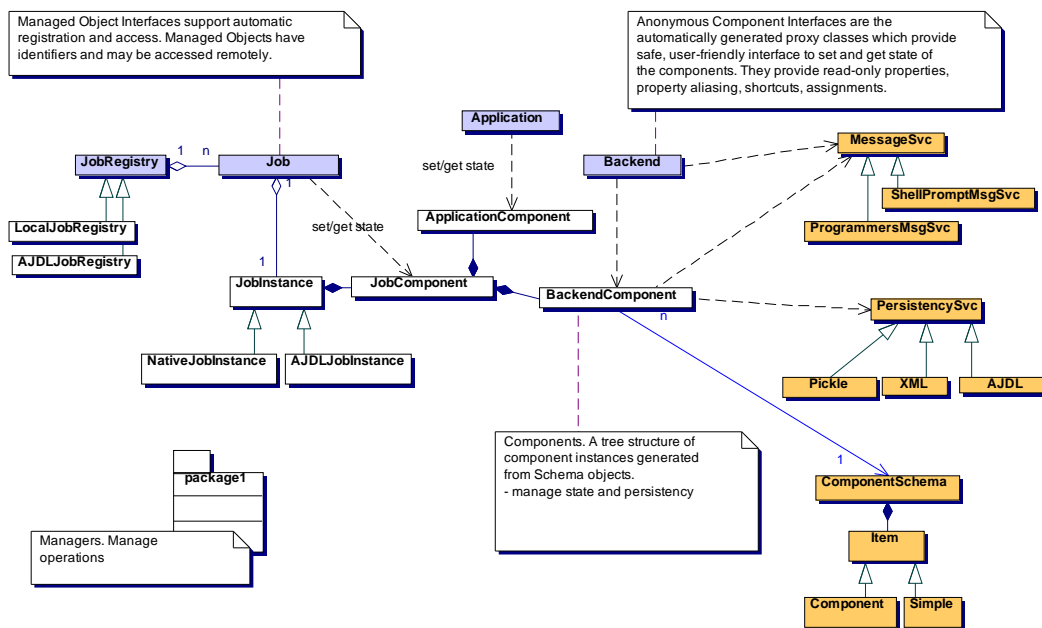
Core Objects are responsible for high-level framework behaviour of the Client, DevelopmentHost (ApplicationManager), Registry Server (Registry) and SubmissionHost (JobManager).

4.7.3 Handlers

Handlers may be logically grouped into “domains”. A domain is a collection of internal handlers which contain the principal functionality of Ganga and which are regarded as implementation details of the (high-level) Core Objects. For example NativeJobManager does the job submission and monitoring within the Ganga process whereas AJDLJobManager uses a remote AJDL service to do so). The domain is a “flavour” of Ganga should there be problems to overcome conflicting requirements from LHCb and ATLAS.

4.7.4 Utility Packages

Packages of common use, such as message, persistency or plugin service.



Drawing 3Vertical Layout of Ganga (based on existing CLIP implementation)

4.8 GPI Objects

Technically the **GPI** is a set of python object *proxies* (no state) which give access to the underlying tree of ContainersOfProperties and ObjectInstances (like JobInstance). GPI allow to manipulate the properties of the Jobs, Applications and Backends but not their internal state (read-only attributes defined in the Schema). GPI also provides automatic registration of the Managed Object Instances in their respective Registries (for example the Job [which is a GPI class] constructor automatically creates a JobInstance and registers it in a JobRegistry). Access to the nested nodes of the tree (like j.app) is provided at the GPI level by appropriate proxy classes (DaVinci is a proxy object to the node which represents DaVinci application in the job tree). These GPI proxies classes are dynamically created from Ganga Plugins (for example: DaVinci plugin generates a DaVinci GPI class which is automatically available inside the Ganga.GPI and Ganga.CLIP package).

4.9 GPI Internals

- ContainersOfProperties do not have identity but they manage the internal state (either local or foreign) and they interact with the Persistency Service to do persistent checkpoints.
- Object Instances are object which have identity and they are registered in the Registries. Each Object Instance has a corresponding ContainerOfProperty object which manages the state. The corresponding GPI proxies for the Instances must be programmed manually to handle the registration (for example Job and Dataset classes in the GPI).

4.10 Core Objects and Handlers

4.10.1 Refer to previous sections for details.

4.11 Utility Packages

4.11.1 Schema and Plugin Mechanism

Ganga may be extended using the Plugin mechanism. A Plugin is a python package which defines a logical schema for a GPI Object i.e.: the names and initial values of the properties. The GPI proxy class is generated

automatically and inserted into the public Ganga.CLIP package unless the Plugin defines an Object Instance. In this case Ganga Core must be modified manually to add a corresponding CLIP proxy (there are no automatic ways for doing it because such a generic mechanism would be prohibitively hard to develop).

4.11.2 Utility Services

4.11.2.1 Messaging

4.11.2.2 Persistency – it will be possible to use different persistency mechanisms to save jobs (for example via XML streamer, pickle streamer etc). Hence it will be also possible to extend Ganga to support some “manual” export/import of jobs to foreign formats (such as AJDL/XML).

4.12 Management of state

4.12.1 Anonymous ContainersOfProperties

For example: `b = LSF()` creates a transient object without a job context. Anonymous ContainersOfProperties are not automatically persistent and cannot be accessed in other way than via the variable name. Anonymous objects always have local state and may not be accessed remotely.

4.12.3 Managed (Registered) ContainersOfProperties

For example: `j = Job()` creates a managed object. Managed objects have an identifier (e.g. job id) which may be used to find them in a Object Registry (e.g. JobRegistry). Managed objects may have an automatic persistency mechanism and may be referenced and accessed remotely. Another examples of potential Managed objects include JobTemplate or Dataset. Note: `j.backend = LSF()` places an anonymous ContainerOfProperties (LSF()) in the managed context which means that automatic persistency will apply to `j.backend`. Internally, a Managed Object Instance has a corresponding ContainerOfProperties which holds its state.

4.13 Management of operations

4.13.1 If schema declares an public operation then this operation must be defined in a plugin as a (stateless) function. Implementation of the operations may require knowledge of Core Interfaces and implementation details of GPI.