Ganga
a job management and optimising tool for job submission to the grid

Andrew Maier (CERN)
for the Ganga Development team
Overview

• People – Sponsors

• Motivation

• Ganga introduction

• Architecture

• Ganga in action

• Current users of Ganga

• Summary
• Ganga is an ATLAS/LHCb joint project

• Development work supported by PPARC through GridPP and by EGEE through ARDA

• Core team:
  – U.Egede (Imperial), K.Harrison (Cambridge), D.Liko (CERN), A.Maier (CERN), J.T.Moscicki (CERN), A.Soroko (Oxford), CL.Tan (Birmingham)

• Contributions from many others, from summer students to senior researchers

• Valued contributions from our colleagues of the Academia Sinica
Motivation (1)

• To submit a job to the grid a number of “problems” have to be overcome:
  – Get a certificate
  – write a JDL with the resources needed
  – create a script to run the application on the worker-node
  – monitor the progress of your job
  – retrieve your output

• The grid may not be the only computing resource you want to use:
  – local machine - for debugging or short tests
  – a local batch system - for small or intermediate datasets
  – grid – for intermediate to large scale datasets
Motivation (2)

• Working on these resources can be different e.g.,
  – For job running locally - no need for a JDL
  – For a job running on a batch system - a shared file system may allow to retrieve results easily
  – One may have to monitor the job manually
  – The commands to submit a job have different syntax

• As a user you are probably not interested in these technicalities:
  ➔ Factor out these differences: use Ganga
Ganga Overview

- store & retrieve job definition
- submit, kill
- get output
- update status
- prepare, configure
- + split, merge, monitor
- Ganga4

- Athena
- Gaudi
- scripts

- LSF
- localhost
- pbs
- LCG2
- gLite
- DIRAC
- AtlasPROD

INFSO-RI-508833
Introduction to Ganga

• **What is Ganga**
  - Ganga is an application in python to help the user to
    - configure, prepare, submit and monitor applications to the local host, a batch or a grid system
  - The goal is to make submitting a job transparent to the batch system used
    - Configure once, run anywhere
User tasks are represented in Ganga in terms of a set of building blocks

- **Application**
  - Specification of the software to be run, including values for configurable parameters

- **Backend**
  - Specification of batch or Grid system to be used, including resource requirements (minimum memory, maximum CPU, etc)

- **Dataset**
  - Used to specify input and/or output, for example a collection of input files containing event data
• **Job**
  – Full task specification - input Dataset, Application, output Dataset, Backend - and bookkeeping information such as ID and status

• **Splitter**
  – Rule for dividing a Job into subjobs that can be run in parallel; rule may relate to Application parameters and/or to input Dataset

• **Merger**
  – Rule for merging outputs from subjobs
Ganga 4 is decomposed into 4 functional components.

These components also describe the components in a distributed model.

Strategy: Design each component so that it could be a separate service.

But allow to combine two or more components into a single service.
- Runs the Ganga interface (CLIP, GPI, GUI)
- The user interacts exclusively through the client
- With the client, the user creates, modifies, submits and monitors jobs
- Job configuration is kept in a registry which can be local (within the client) or remote.
Client

- The client is a **thin client** (pure python)
- The client can be a **command line client** or a **GUI**
- The client interacts with the **application manager** to configure applications
- It submits and monitors **jobs** via the **job manager**
- It keeps state by storing **persistent information** in the registry
Application Manager

- Prepares and configures the application
- Compiles user code
- Sets-up the necessary environment
- Provides information to the client on available applications, versions, platforms, etc.
Job Manager

- **Submits** the configured job to the submission backend
- A **submission handler** submits a job to a backend
  - Creates the starter script and the JDL
  - Performs the monitoring
- The **application runtime handler**
  - Prepares the application dependent wrapper script, depending on the backend.
  - E.g., DIRAC knows how to run LHCb applications with a different setup as LSF.
Remote Registry

- Keeps track of jobs
- Is a “passive” data store, typically using a database backend
- Keeps a roaming profile of the user jobs
- Ganga uses the AMGA metadata catalogue
- Keeps track of the job status
The key operations for a user running jobs are typically
- Job definition
- Job submission
- Job cancellation
- Job monitoring
- Output retrieval

These are performed in Ganga using simple GPI/User commands

Technicalities are hidden from the user
A job can be defined in Ganga starting from an instance of the Job class

Job properties can be passed as arguments to the constructor

- \( j = \text{Job}\left(\text{application} = \text{Executable}(), \text{backend} = \text{LCG}()\right) \)

Job properties and sub-properties can also be set through assignments

- \( j\.\text{application\.exe} = "\text{/bin/echo}" \)
- \( j\.\text{application\.args} = [ "\text{Hello World"} ] \)
- User command: `job.submit()`
- Outcome: job submitted, split into subjobs, command status returned

- Beneath the surface:

  - Job derived parameters
  - Job and derived parameters
  - Subjobs
  - Subjobs derived parameters
  - Subjobs and derived parameters
  - Status

- **Application Manager**
  - Perform application configuration for job

- **Job Manager**
  - Split job into subjobs
  - Create wrapper scripts, submit to backend

- **Archivist**
  - Register subjobs, allocate workspace
  - Register subjobs as submitted
• User action: none
• Outcome: changes in job status reported/updated periodically, output retrieved automatically when job completes

- Beneath the surface:

  Job Manager ➔ Control Monitoring thread ➔ Determine active jobs for each backend

  ⇒ For each backend with active jobs:

  Query status of active jobs

  ⇒ For each job with change of status:

  Report change of status to uses, retrieve output if job completed

  Archivist ➔ Register new job status
CLIP: simple job from 1st principles

Job defined in Cambridge

Job run in Amsterdam

INFSO-RI-508833

ISGC 2006, May 2006 - A. Maier (CERN)
Running ATLAS jobs on LCG

In [4]: status = Job(templates["SusyPlot"])[0]).submit()
Ganga.GPIDev.Lib.Job : INFO submitting job 1

In [5]: Ganga.Lib.LCG
   atlas - Wed Sep 28 13:56:38 2005
Ganga.Lib.LCG : INFO Job 1 Running at grid10.lal.in2p3.fr:2119/jobmanager-pbs-atlas -
Ganga.Lib.LCG : INFO Job 1 Done (Success) at grid10.lal.in2p3.fr:2119/jobmanager-pbs-atlas

In [5]: print jobs
Statistics: 1 jobs

---------
ID  status   name
# 1 completed SusyPlot grid10.lal.in2p3.fr:2119/jobmanager-pbs-atlas

In [6]: print os.listdir(jobs[1].outputdir)
['gangademo_susyplot_hist.root', 'stderr', 'stdout']

In [7]: []
- Ganga ship with a GUI
- Based on pyqt
- Completely dockable
- Dynamically built on the internal plugin architecture
- Includes a job builder wizard
Ganga GUI (top half)

List of Jobs

Job Details

Logical Job Folder list
Ganga Gui (bottom half)

A scriptlet

Saved scriptlets

Ganga/Python prompt
• Atlas uses the Athena framework for creating jobs

• Atlas DDM is being deployed on LCG sites
  – Dataset concept
  – Will be used to distribute data
  – DQ2 plugin for Ganga gives access to the catalog

• Site configuration
  – Short queues to support analysis jobs
  – Fast response for users

• New middleware components
  – gLite Resource Broker
  – Bulk submission
• Based on these developments Ganga plans for a new version by the end of May

• In addition Ganga will connect to Panda, the Production and Analysis of ATLAS on OSG

• Atlas will use Ganga as frontend to submit jobs to the various backends

• Atlas will use Application Monitoring based on MonaLisa to measure the actual performance of the various backends
Applications in LHCb are based on a common framework called Gaudi

- Users write code to be dynamically linked against the Gaudi framework

- These shared libraries, the user options and the dataset to be analysed will sent for execution

- Ganga has a plugin for Gaudi based applications and helps the user to configure the job
**Gaudi based applications:**

```python
In [3]: dv = DaVinci(version='v12r12')
In [4]: print dv
DaVinci {
    version = 'v12r12',
    extraopts = None,
    package = 'Phys',
    cmt_user_path = '/afs/cern.ch/user/u/uegede/cmtuser',
    masterpackage = None,
    optsfile = File {
        name = ''
    }
}
```

- Specify extra option file properties appended to the options file
- Specify the package you are working on
- Specify the options file to be used
Analysis access to the Grid

- **Analysis jobs: No direct submission to LCG**
  - Instead: Submission to the DIRAC workload management system

- **Advantages:**
  - Reduce the knowledge required to submit to the Grid
  - Provide transparent access to the LFC file catalogue for reading and writing data
  - Allow LHCb to set priorities and or restrictions for analysis jobs
Other users of Ganga

- **Gridproduction testbed**
  - Tests the functionality and availability of grid sites

- **Geant 4 simulation**
  - new versions are tested against result of earlier version

- **Biomed**
  - (see Hurng-Chun Lee's presentation yesterday)

- **ITU**
  - want to use Ganga in autumn, to aid the negotiation of new digital TV frequencies
Summary

- Ganga simplifies the submission of jobs to the grid
  - prepares
  - submits
  - monitors and retrieves results
- Ganga features a plugin architecture to easily enhance additional backends or applications
- Ganga factors out the technicalities to submit to different computing resources (local machine, batch system, grid)
- Demo session of Ganga, tomorrow
- Ganga motto:

  Configure once, run anywhere
Spare

• Spare
• **Ganga is a lightweight user tool**
  - easy to install (pure-python)
  - “designed and optimized” for users
• **But also: Ganga is a developer framework**
  - Plugin model
    - independent and rapid development of handlers (backends, applications)
  - Promote but not force common GPI abstractions
    - We do not require nor invent abstract base classes which are least common denominators between systems, example:
      - *you may implement very complex application (e.g. ADA) and enable submission to DIAL only if that’s your main case*
    - the design of framework does not attempt to match all possible applications with all possible backends
  - **But**: enable to build common tools on top of GPI: GUI, scripts,…
Status of the various backends

• **LCG**
  – Standard LCG Resource Broker
  – Currently available

• **CondorG**
  – Direct submission to CE
  – Currently available

• **gLite**
  – New resource Broker
  – BulkSubmission
  – Under development

• **PANDA**
  – Production and Analysis on OSG
  – First version

• **Prodsys**
  – Production and Distributed Analysis on LCG