LEXOR, the LCG-2 Executor for the ATLAS DC2 Production System

A. De Salvo (INFN Roma);
G. Negri (CNAF Bologna);
D. Rebatto, L. Vaccarossa (INFN Milano).

Abstract

In this paper we present an overview of the implementation of the LCG interface for the ATLAS production system. In order to take profit of the features provided by DataGRID software, on wich LCG is based, we implemented a Python module, seamless integrated into the Workload Management System, which can be used as an object-oriented API to the submission services. On top of it we implemented Lexor, an executor component conforming to the pull/push model designed by the DC2 production system team. It pulls job descriptions from the supervisor component and uses them to create job objects, which in turn are submitted to the Grid. The interactions with the production database and ATLAS metadata catalog, and the staging of input and output files, are granted by the integration with the Don Quijote client module and via XML messages to the production supervisor (Windmill).

ATLAS PRODUCTION SYSTEM

The ATLAS Data Challenge

Starting on Summer '04, the ATLAS experiments [1] undertake its second Data Challenge (DC2) in order to validate its computing and data model. The phase I of DC2 consists in a massive production of simulated data (about 107 events). Events are generated using different generators (mainly Pythia and Herwig), while the simulation (tracking of particles through the detector and recording their interactions with sensitive elements of the detector) is made using Geant4. Phase I will then proceed with pile-up (superposition of background events with the signal ones) and digitization, that will finally produce a suitable output for reconstruction, which will be one of the main tasks in phase II.

The Production System

The DC2 production was planned to rely on the Grid technology for distributed execution, while keeping the local batch systems as backup solution.

For this purpose, a modular production system has been designed and implemented. Its architecture consists in one central database holding grid-neutral information about all the jobs. A *supervisor* agent (Windmill, [3]) pick up unprocessed jobs from this database, distribute them to the *executors* components, and verify them after execution. The executors (one implementation per grid flavor) offer to the supervisor a common, grid independent interface to the underlying middleware. For a detailed description of the Production System, see [2].

Three executors have been developed: *Capone* for GRID3, *Dulcinea* for NorduGrid and *Lexor* for LCG-2.

LEXOR IMPLEMENTATION

Lexor has been fully coded in Python. This language encourage a modular, object oriented approach to the application developement, so we adopted this philosophy throughout Lexor design.

The interface to LCG-2

As first, we created a module implementing a generic LCG job class. Our code is based on the SWIG API to the workload management system, developed by the Work Package 1 (WP1) of the European DataGrid (EDG) project for their User Interface.

Through this API, our class offers methods for defining, submitting, and managing the job on the grid.

For job definition, a set of methods allows the manipulation of the underlying JDL description, either by direct access to the classAd attributes or via higher level functions (e.g. `addRequirements()`, used to add an AND-ed condition to the requirements for the CE). Another few methods are used to tune and perform the job submission, to monitor the job status (and extract single pieces of information from the status), and finally to retrieve the output sandbox of the job. A job cancellation method is also provided, but not yet used in the production system.

If imported in an interactive Python session, the LCG job class can also be used as a programmable, object oriented replacement for the standard LCG User Interface when working with standard jobs (checkpointable and interactive jobs are not supported).

The XML-to-JDL translator

The encoding and decoding of the XML control messages between supervisor and executor are demanded to a dedicated module coming with the supervisor.

Only for parsing the job description, we implemented a custom module with a different approach from the default one. Instead of looking for a fixed set of attributes in the XML job description, we recursively navigate the whole XML tree. For each node, a corresponding method of the parser class is invoked, which is responsible to either do further navigation in its subnodes, or to produce the relative JDL attribute. This approach has proven to be quickly and easily extensible: changes and improvements in the XML schema for job description were usually coped by just adding few lines of code.

Most of the common attributes (i.e the ones not depending on the single job, and thus not present in the XML job description) are taken from Lexor configuration file. Here, by example, a user can select a subset of sites where the job can run, with a semantic similar to `hosts.allow` and `hosts.deny`. By specifying regular expressions in the allowedSites and forbiddenSites fields of the configuration file, a set of `RegExp(<the_regexp>, other.GlueCEUniqueID)` expressions are added to the `Requirements` attribute of the JDL.

Resource availability estimation

When it is in submission mode, the supervisor periodically asks for the number of jobs the executor is willing to manage. To answer this question, we issue an ldap query to the LCG Information System to retrieve the number of free CPUs in the sites matching the requirements expressed in the supervisor request.

Computing this number revaled to be a tricky task, due to the aggregated nature of the information published by the LCG Information System. Often we overestimate the real amount of available CPUs, because some of the advertised ones are not accessible by ATLAS jobs. This problem was temporarily addressed by defining one queue for each VO, but a refining of the information schema has to be considered.

The Job Wrapper

Once submitted, a job is dispatched to the Computing Element (CE) selected by the broker and enqueued until the local batch system start executing it on a Worker Node (WN). Anyway, before the required transformation can be started, a lot of actions have to be performed on the WN, so we had to wrap the job in a pretty thick script. To have an idea of all the tasks accomplished by the job wrapper, see Figure 1. Here are shown also the interactions whith external services like the PACMAN [6] repository where the tranformation packages are hosted, or the Don Quijote server [4], [5].

In the wrapper we also embedded some mechanisms to cope with temporary problems in the underlaying grid. The biggest one was the occasional unavailability of the ATLAS software, due to NFS problems between the CE exporting the software direcory and the WN. Jobs being executed on such hosts failed at the very beginning, freeing the WN which attracted more and more jobs. To avoid this “black-hole” effect, we modified the wrapper to have the job sending an e-mail message to the submitter about the problem, and then sleeping for some hours. When it wakes up, the process is repeated until the software is found or the job is killed. This keeps the WN busy, preventing further attempts to run jobs on it.

Also the stage-in and the stage-out phases of the wrapper were enclosed in a *sleep and retry* cycle. The LCG commands used to stage files are in fact very sensible to failures or big delays in the services they rely on. The heavy workloadload, imposed on these services by the high number of concurrent requests from the running jobs, sometimes caused them to become unresponsive for a while or even to crash. This affected all the running jobs, which after having successfully executed the transformation failed to stage the produced files to the Storage Element. The retry mechanism didn’t completely solved the problem, but it allowed to recover many jobs where the LCG commands hung. In fact, killing by hand the command process, a subsequent attempt was triggered, and usually succefully accomplished.

RUNNING LEXOR

In three months of activity, Lexor has managed more than 100.000 jobs, including generation, simulation, digitization and pile-up. More than 30 LCG sites were involved, providing overall about 3.000 CPUs.

The job management was shared, on average, among six Lexor instances owned by different users, every instance taking care of a maximum of about 800 jobs at a time. This threshold was decided in order to limit the size of the queries issued by supervisors to the central production database.

Troughs in the day by day production plot (Figure 2) reflect the various issues encountered during this first phase of the Data Challenge. In fact, thanks to the heavy load we imposed on all the components, we were able to hunt undiscovered bugs in all the components of the production system and in LCG middleware.

REFERENCES

[1] http://atlas.web.cern.ch/Atlas/

[2] Luc Goossens, “ATLAS production System in ATLAS Data Challenge 2”, [501] CHEP 2004.

[3] http://heppc12.uta.edu/windmill/

[4] Miguel Branco, “Don Quijote - Data Management for the ATLAS Automatic Production System”, [142] CHEP 2004

[5] http://mbranco.home.cern.ch/mbranco/cern/donquijote/

[6] http://physics.bu.edu/ youssef/pacman/

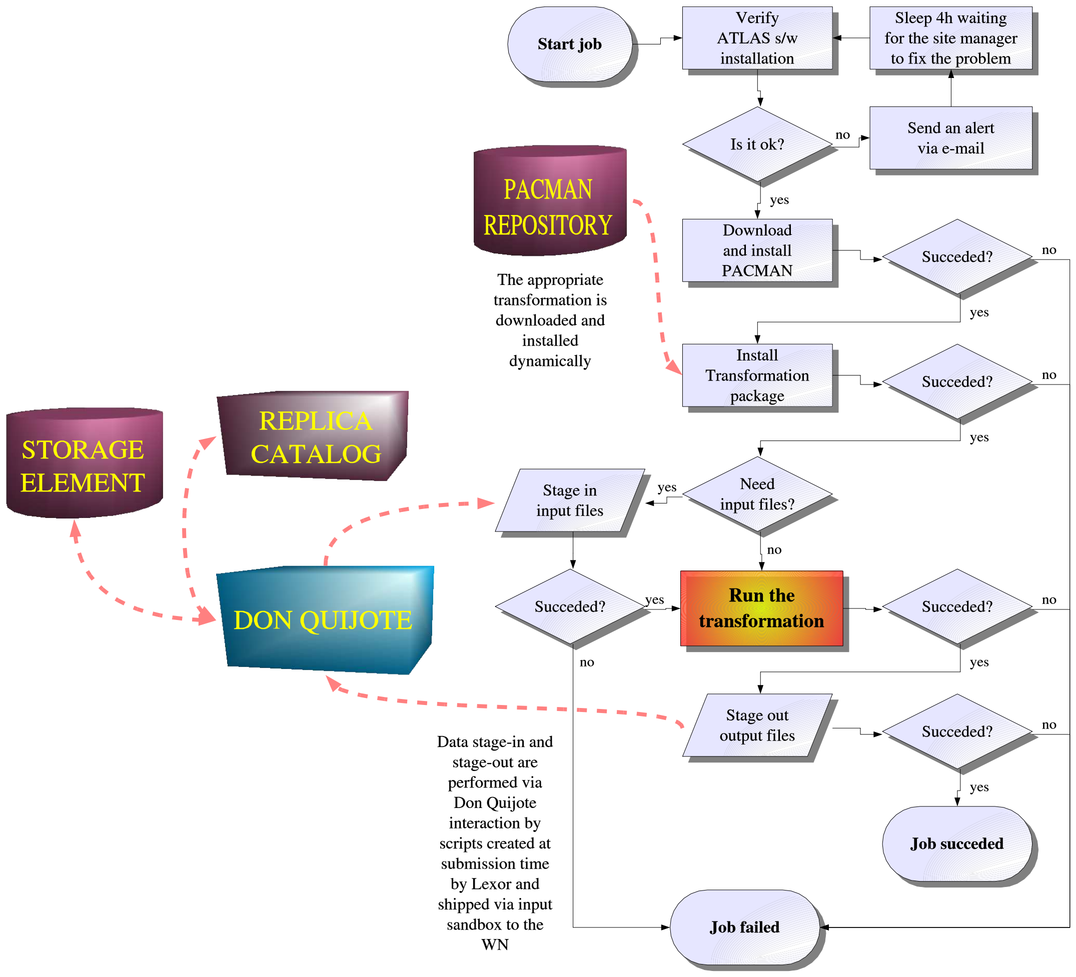
Start job
Verify ATLAS s/w installation
Sleep 4h waiting for the site manager to fix the problem
Is it ok?
no
Send an alert via e-mail
yes
PACMAN REPOSITORY
The appropriate transformation is downloaded and installed dynamically
Download and install PACMAN
Succeded?
no
yes
Install Transformation package
Succeded?
no
yes
STORAGE ELEMENT
REPLICA CATALOG
Stage in input files
yes
Need input files?
no
DON QUIJOTE
Succeded?
yes
Run the transformation
Succeded?
no
no
yes
Stage out output files
Succeded?
no
yes
Data stage-in and stage-out are performed via Don Quijote interaction by scripts created at submission time by Lexor and shipped via input sandbox to the WN
Job succeded
Job failed

Figure 1: A lot of operations are carried out by the job wrapper before and after the actual transformation.

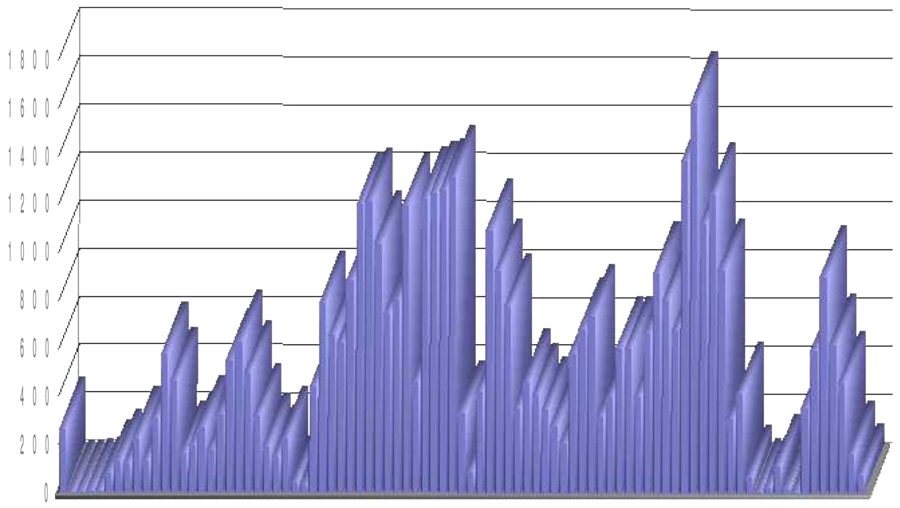

Figure 2: Jobs processed by Lexor day by day.