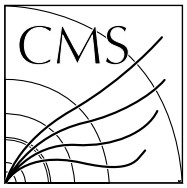
Available on CMS information server

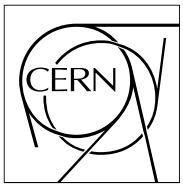
CMS NOTE 2003/005



The Compact Muon Solenoid Experiment

CMS Note

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland



March 12, 2003

Object Based System for Batch Job Submission and Monitoring (BOSS)

C. Grandi, A. Renzi

INFN-Bologna, Italy

Abstract

High Energy Physics experiments need to produce large amounts of data, either running event generators or processing existing data. In general the processes are composed of several steps and each step needs specific bookkeeping in order to assure the quality of the data produced. Furthermore, production is carried on at several production centers, each one with its own hardware and software setup. In this note we present BOSS (Batch Object Submission System). BOSS provides real time monitoring and bookkeeping of jobs. Different job types can be registered to the BOSS System, allowing information specific to the task performed by the job to be stored in a local database. BOSS is configured as an interface to local batch schedulers, providing a unique interface to the batch facilities. BOSS has been used by the CMS experiment at the CERN LHC collider for the production of large amounts of simulated data needed to provide an adequate statistic for Trigger System design. The use of BOSS as an interface to a GRID scheduler is also discussed.

1 Introduction

The management of a big Monte Carlo production or data analysis as well as the quality assurance of the results, require a careful monitoring and book-keeping of the batch jobs. This is normally achieved by parsing the *log file* of the job (i.e. the standard output and standard error of the job executable) with ad-hoc programs that can build summaries and/or check that the execution was as expected.

BOSS (Batch Object Submission System) has been developed to provide real-time monitoring and bookkeeping of jobs submitted to a compute farm system. The information is persistently stored in a relational database (MySQL in the current version) for further processing. In this way the information that was available in the *log file* in a free form is structured in a fixed-form that allows easy and efficient access. BOSS can monitor not only the typical information provided by the batch systems (e.g. executable name, time of submission and execution, return status, etc…), but also information specific to the job that is being executed (e.g. dataset that is being produced or analyzed, number of events done so far, number of events to be done, etc…).

BOSS extracts the specific job information to be monitored from the standard input, output and error streams of the job itself and stores it in the database. The fact that the information is extracted from the job rather than asserted by the user allows reliable bookkeeping. The user can provide BOSS with description of the parameters to be monitored and the way to access them by registering a *job type*. A *job type* is defined by:

- a *schema*, which describes the parameters to be monitored;
- a set of executables that contain the algorithms to determine the values of the parameters from the standard input, output and error of the job. The executables are the *pre-process*, *post-process* and *runtime-process* filters, which are executed before, after and during the job execution respectively.

Thus we can say that a user job is of type *job type* if the *pre-process*, *post-process* and *runtime-process* filters can parse its standard input, output and error and determine the values of the parameters contained in the *schema*.

When a job is submitted through BOSS the request details, including its *job type*, are stored in the database. Then BOSS builds a wrapper around the job itself, the *jobExecutor*, which is submitted to the farm batch system together with all the information needed to run the user job. When the execution starts a few processes are started on the execution host together with the real user executable. In particular the *dbUpdator* process that has the capability to submit queries to the relational database is used to update the job status in the database in real time. BOSS basic flow is shown in figure 1.

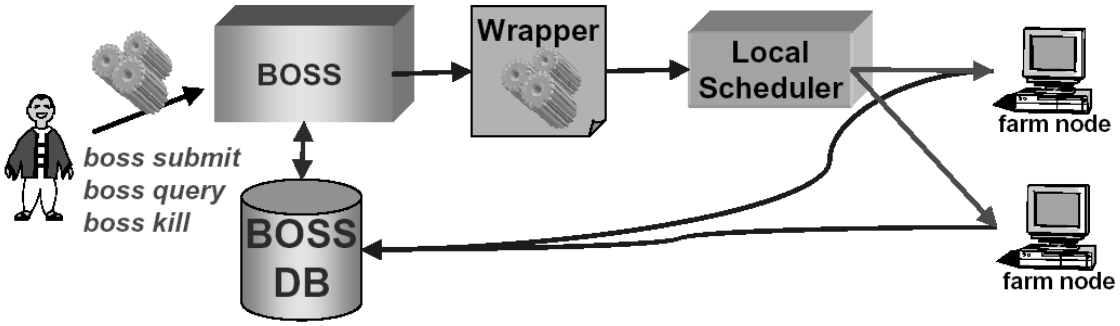


Figure 1: Basic flow of BOSS

BOSS is not a batch system. It interfaces to a local scheduler (e.g. LSF, PBS, Condor, etc…) through a set of scripts provided by the farm administrator, using a predefined syntax. The local scheduler job identification string is kept in the database together with the BOSS job identification number so that it is always possible to reach the running job via the local batch system through BOSS commands. In the current version BOSS provides an interface to the local scheduler for the operations of job submission, deletion and query.

BOSS provides a few commands for browsing the database. They can be used inside user scripts to access in an easy way the information about the jobs. In this way the user can track the state of the running jobs as well as of the finished jobs, which allows identifying anomalies and building summaries.

BOSS has a command-line interface. Detailed description on its use may be found at the BOSS web site [1]. A prototype of a web-based graphical interface has been developed using perl and php4 with SSL based access to the boss system [2]. A CMS-customized web-based interface to the BOSS database has also been developed and

deployed in the CMS production environment [3].

This note describes BOSS version 3.3.

BOSS can be downloaded from http://www.bo.infn.it/cms/computing/BOSS/.

2 Architecture

The main components of BOSS are:

- The `boss` executable: the BOSS interface to the user
- The database: where BOSS stores job information
- The `jobExecutor` executable: the BOSS wrapper around the user job
- The `dbUpdator` executable: the process that writes to the database as the job is running
- The local scheduler: the compute farm batch system

A BOSS installation consists of a `boss`, a `jobExecutor` and zero or more `dbUpdator` executables. Each installation is connected to a database on a DB server. The same database may be used by many BOSS installations. The `dbUpdator` executable is optional because runtime update of the database may be excluded. There may be more `dbUpdator` instances in a BOSS installation that use different means to update the database. By default a `dbUpdator` that connects directly to the database is provided. Each installation is connected to one or more local schedulers. This is summarized in figure 2.

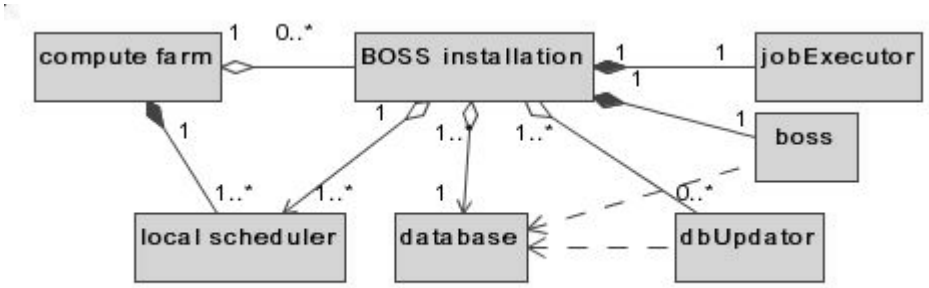


Figure 2: Cardinality relations among BOSS components

2.1 Interaction with the database

In the current release BOSS uses MySQL [4] as underlying database system. The MySQL contact information is:

- MySQL server host, port and UNIX socket
- MySQL client flag
- MySQL database name
- MySQL username and password
- MySQL username and password for direct database operations

At installation time the user may modify the default database contact information that is written in a configuration file in the BOSS top directory (`BOSSDBConfig.sh`). The file is read when compiling the BOSS executables. Since the executables are statically linked, the `boss` and the default `dbUpdator` executables are self-contained and connected to a single instance of a database. The `jobExecutor` executable is also statically linked but it doesn't depend on the database contact information.

The operations BOSS performs on the database are:

- `CREATE TABLE`, `DROP TABLE`
- `INSERT`, `DELETE`, `UPDATE`

- `SELECT`, `SHOW FIELDS`

The data types used are:

- `INT` (used for integers and time in UNIX format)
- `VCHAR` (used for BOSS-defined strings)
- `BLOB` (used for user-defined strings and for files, being them binaries or ASCII)

Any database system can be used in place of MySQL provided it supports these functionalities.

2.2 Interaction with the local batch system

From the architectural point of view it is possible to consider BOSS as the component that prepares the wrapper around the user job letting the user submit the job to the preferred batch manager. Nevertheless submitting the job through the BOSS commands allows the complete tracking of its status because at submission time BOSS stores information in the database. Furthermore the job identifier assigned by BOSS to any job it manages (`BOSSJobId`) is put in one-to-one correspondence with the job identifier assigned to the job by the local batch system (`nativeJobId`) allowing further interaction with the job.

The interaction through BOSS with the local batch system is simple so that it is portable to a wide variety of implementations. Basically only three operations are considered: job submission, job cancellation and job status retrieval. Any other operation has to go directly through the batch system user interface. Support to a given batch system is provided via a set of executables that are registered in the form of plug-in to BOSS. The interfaces of the plug-in are:

Operation	*Plug-in name*	*Interface*
Job submission	`submit`	`nativeJobId = submit(BOSSJobId,…);`
Job cancellation	`kill`	`success = kill(nativeJobId);`
Job status retrieval	`query`	`list<nativeJobId,jobStatus> = query();`

The job submission plug-in only has to submit the BOSS wrapper (`jobExecutor`) with a few arguments, the most important being the `BOSSJobId` and has to return the job identifier assigned by the batch system to the job. The `jobExecutor` doesn't need any standard input stream, and both its standard output and error streams are sent to a *log file*. The BOSS wrapper will then take care of executing the real job submitted by the user with the appropriate standard I/O streams and arguments.

Job cancellation requests are simply forwarded to the batch system.

The job status retrieval plug-in has to return a list of all the batch jobs known to it that are either running or queued, with a job status flag.

BOSS entirely relies on the batch system to execute the user job reliably. If the batch scheduler indicates that the job failed or could not be executed, it simply passes the error up, as it has no information useful for recovery.

The plug-in executables are stored in the BOSS database in a dedicated table.

BOSS also gives the possibility to associate a top working directory to a scheduler. If a top working directory is specified, the BOSS wrapper (`jobExecutor`) will create a unique directory under it, where the job will start its execution.

In case more schedulers are registered BOSS, one is defined as the default.

2.3 Job *polymorphism*

A "job" in BOSS architecture is characterized by a set of data variables that specify the job status and a set of methods to determine the status variables. Some of the data variables and the methods to determine them are common to any kind of job. These variables are typically those related to the batch system operation (submission time, executable name, standard I/O files, etc…) and are persistently stored in the BOSS database in a dedicated table that has the BOSS job identifier as primary key. Some other variables instead are typical of specific job types (e.g. number of simulated events produced by a given Monte Carlo job, non-standard I/O files used, etc…) and are determined by parsing the standard I/O streams of the job itself. In other words the user will deal with *concrete jobs* but BOSS will manage *abstract jobs*.

Even though BOSS is written in an Object Oriented language (C++), polymorphism is not implemented using

inheritance since the definition of a new concrete job type should not imply recompilation of the BOSS executables. The way that is chosen is similar to that used for batch scheduler management:

- Data variables are defined by means of a schema i.e. a comma separated list of `variable:type` pairs. Internally BOSS treats all variables as strings and keeps them in an STL map of `<variable,value>` pairs.
- Methods used to extract the variable values from the standard I/O streams are implemented as plug-in executables with defined interfaces that are invoked by BOSS (*filters*).

When a new job type is registered with BOSS, the schema and the plug-in executables are stored in the BOSS database in a dedicated table. Furthermore a new table is created in which the columns are the variables specified in its schema. In addition a column with the BOSS job identifier is added so that it is possible to link information stored in the specific job table to those in the standard job table.

The interface of the filters is the following:

Operation	*Plug-in name*	*Interface*
Job filtering	*XXX*`-process`	`list<`*variable*`=`*value*`> = XXX-process(`*stream*`);`

XXX can be `pre` (executed before user job execution), `runtime` (executed during user job execution) and `post` (executed after user job execution). *stream* can be the user job standard input, output or error. In detail: `pre-process` filters the standard input of the job, `runtime-process` and `post-process` filter both the standard output and the standard error of the job.

It is possible to associate to a single job more than one job type. If more job types are specified all the filter files of the different job types will parse the job standard I/O streams.

It is worth stressing that the only communication channels between the jobs and the BOSS system are the job I/O streams through the plug-in filters. In this view we can give this definition: *"a job is of a given type if the filters of that type are able to extract the value of the variables of that job-type from the job I/O streams"*.

When a job is submitted to BOSS, the user may optionally specify its *types*. BOSS retrieves from the database the filter files and stores them in an *archive file* together with the information needed to run the job (executable, standard I/O streams, arguments, etc…). The archive file must be made available to the executing job.

Figure 3 shows an example of the filtering procedure. The user job writes to its standard output a string "`counter` *N*" every second, where *N* is the number of seconds since the beginning. The filter script looks for lines in the standard output matching this kind of string and writes out the string "`COUNTER=`*N*" where `COUNTER` is the name of the variable in the job schema (i.e. in the database table). The `jobExecutor` reads the output of the filter and writes to a journal file. Optionally the `jobExecutor` may start the `dbUpdator` process that updates the database.

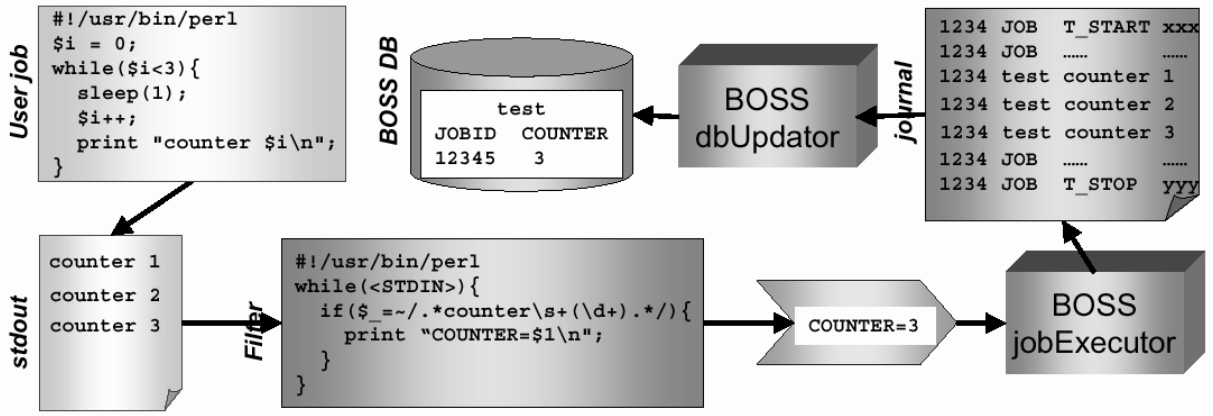


Figure 3: Simplified view of the BOSS filtering procedure

2.4 The job wrapper: `jobExecutor`

The farm scheduler starts the BOSS wrapper (`jobExecutor`) on a farm node using a few arguments:

- the job identifier assigned by BOSS to the user job;

- the BOSS *base directory*;
- the BOSS *top working directory*;
- an optional flag for local I/O.

The different steps performed by the `jobExecutor` are the following:

- If a *top working directory* is passed to the `jobExecutor`, a dedicated working directory is created under it.
- An empty *journal file* is created in the local working directory. It will contain all the updates to be done to the database.
- If real-time update of the database is requested the `dbUpdator` process is started. The `dbUpdator` is statically compiled with all the information needed to contact the database server hard-coded in it. The `dbUpdator` reads the journal file at regular intervals and updates the job information in the database.
- The details of the user job (executable name, standard I/O stream files, etc…) are retrieved from the archive file that is found in the *base directory*.
- Pre-processing step: the user specified standard input file is parsed by the `pre-process` filter files. The output of the filter is added by `jobExecutor` to the journal file.
- Run-time processing step: a number of temporary named pipes (FIFO's) and processes are created on the local farm node together with the real user executable. The standard output and error streams are intercepted by BOSS and processed using the job-type filters. The detailed schema of files and processes is shown in figure 4. The user job is started using as standard input the file specified by the user. The standard output and error streams are sent to two FIFO's that are read by a "splitter" (the UNIX `tee` executable). One stream from the splitters is sent to the user specified standard output and error files. A stream for each job type is created and sent to other FIFO's that are processed by the `runtime-process` filter files. The output of the filters is sent to another FIFO that is read by the `jobExecutor` itself that writes it to the journal file.
- Post-processing step: the standard output and error files, produced in the previous step are parsed by the `pre-process` filter files. The output of the filter is added by `jobExecutor` to the journal file.

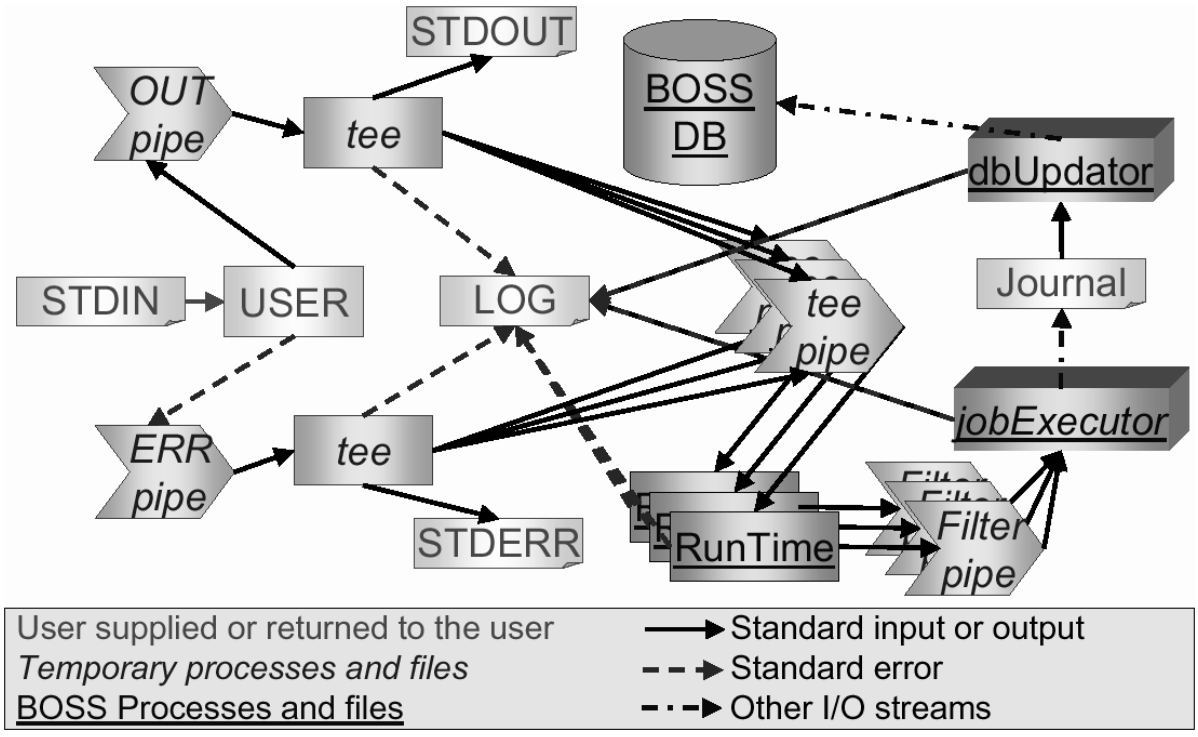


Figure 4: Schema of the processes and files created by `jobExecutor` at run-time

2.5 The user interface: `boss`

The `boss` executable is an interface to a set of commands. Each command is a C++ class that inherits from the `BossCommand` class that defines the interface. All commands are created at the beginning of the execution and are stored in a container. Adding a new command to boss implies creating a new class inheriting from `BossCommand` and instantiating it in `BossCommandContainer`. The commands implemented in version 3.3 are shown in table 1.

Command name	Desctiption
`BossRegisterScheduler`	Registers the interface programs to a batch system.
`BossShowSchedulers`	Lists the schedulers known by this BOSS installation.
`BossDeleteScheduler`	Delete a scheduler known by this BOSS installation.
`BossRegisterJob`	Registers the schema and filter executables for a job type.
`BossShowJobTypes`	Lists the job types known by this BOSS installation.
`BossDeleteJobType`	Delete a job type known by this BOSS installation.
`BossSubmit`	Submit a job to the BOSS system. Details are stored in the database and the wrapper (`jobExecutor`) is submitted to the requested batch system.
`BossDeclare`	As `BossSubmit`, but the job is not submitted to any batch system.
`BossQuery`	Displays a list of jobs managed by BOSS together with their status retrieved from the batch system.
`BossKill`	Asks the batch scheduler to terminate the execution of a job managed by BOSS. The job is marked as killed in the BOSS database and not deleted.
`BossDelete`	Delete a job entry in the BOSS database.
`BossPurge`	Delete job entries in BOSS database older than a specified date.
`BossSQL`	Place a normal SQL query to the BOSS database
`BossRecoverJob`	Updates job information in the database starting from the journal file or from the standard output and error of the job.
`BossVersion`	Displays the version of the BOSS executable.

Table 1: Description of BOSS commands

2.6 The BOSS job

Jobs are internally managed as objects of type `BossJob`. It also takes care of keeping the status of its persistent representation in the database consistent with the transient copy. All BOSS components only interact with `BossJob` to modify the status of a job.

2.7 Using ClassAd

ClassAd (Classified Advertisements) [5] is a syntax that allows specifying a mapping from *attribute names* to *expressions*. In the simplest cases, the expressions are simple constants (integer, floating point, or string). A ClassAd is thus a form of *property list*. In BOSS ClassAd may be used at submission time to specify the properties of a job in a file instead of specifying them as options on the `boss submit` command line. Internally ClassAd are managed by the `BossClassAd` class, which is a wrapper around the ClassAd class coming in the ClassAd distribution. The advantage in using ClassAd's is that it is possible to add attributes (i.e. `name=value` pairs) that are not standard BOSS submission options. These attributes are ignored by BOSS and passed as they are to the `submit` executable of the local batch system. Currently BOSS understands only the syntax of the Condor batch system and of the European DataGrid scheduler (JDL, see chapter 3). This means that a command file that is prepared following either Condor rules or of JDL can directly be submitted to BOSS. In principle it is possible to modify BOSS to understand any ClassAd based syntax.

3 Use in Grid environment

On the grid the submitting machine environment is completely disjoint from that of the executing machine. Since the MySQL server contact details are hard coded in the `dbUpdator` and that the `jobExecutor` and the `dbUpdator` itself are statically linked, it is possible to move it on any machine with the appropriate operating system and it will run. The only requirement in case real-time monitoring is required, is that the executing host has outbound connectivity and that the MySQL server accepts connections from all the grid nodes.

BOSS has been used with the scheduler provided by the (EDG) European DataGrid project [6]. The BOSS ClassAd parser has been modified to be able to understand the JDL (Job Description Language) syntax that is used by EDG to describe a job to be submitted. The BOSS archive file and all the needed files (i.e. the user specified executable file, the standard input file, the `jobExecutor`, the `dbUpdator`) are shipped to the execution host via the EDG *input sandbox* mechanism, and the BOSS journal file is shipped back in the *output sandbox* together with the specified standard output error and log files.

The `jobExecutor` has to be executed with the `localIO` flag. This means that the executables and standard input/output files may be found by BOSS in the current directory on the execution host (i.e. the worker node). This because *input sandbox* files are put in the execution directory. All the communication from the worker node is through MySQL. This is understood to be a weakness of the system and it will soon be replaced by a more robust mechanism.

4 Current use of BOSS

Previous versions of BOSS have been used as a job-monitoring tool by the CMS experiment during *Spring 2002* and *Summer 2002* data productions [7]. Every CMS Regional Center had a local MySQL server hosting the BOSS database. About 20 regional centers participated with about one thousand CPU's in total. Production ran continuously for about 4 months and over 100,000 jobs where executed using BOSS. At the time of writing, CMS is still producing simulated data and BOSS is also used for job monitoring.

A prerelease of BOSS version 3.3 has recently been used for monitoring CMS production jobs being submitted to the European DataGrid test-bed during the CMS-DataGrid stress test [8].

Acknowledgments

We would like to thank Julia Andreeva, Greg Graham, Veronique Lefebure, Tony Wildish and all the CMS production team for the suggestions on how to extend the functionalities of BOSS and for the patience they had especially with the first, buggy versions of BOSS. A special thank goes to Julia for the help she gave us in deploying BOSS to the CMS production environment: without her contribution it wouldn't be possible to reach a production-level quality. For the interesting discussions we had we would like to thank Rick Cavanaugh, Koen Holtman, Nikolai Smirnov of the CMS Grid Integration group and Peter Couvares, Miron Livny, Douglas Thain of the Condor team. For the recent porting of BOSS to the European DataGrid environment we would like to thank Daniele Bonacorsi , Alessandra Fanfani and Henry Nebrinski.

References

[1] BOSS Web Page: http://www.bo.infn.it/cms/computing/BOSS/

[2] A. Bassi (INFN-Bologna), private communication.

[3] A.Feline, *"BODE"*, http://filine.home.cern.ch/filine/

[4] MySQL web page: http://www.mysql.org/

[5] ClassAd web page at University of Wisconsin http://www.cs.wisc.edu/condor/classad/

[6] European DataGrid web page: http://www.eu-datagrid.org/

[7] **CMS Note 2002/034**, The CMS production Team: T.Wildish, V.Lefebure *et al., "The Spring 2002 DAQ TDR Production"*

[8] **CMS Note 2003/XXX**, The CMS/EDG Stress-Test Task Force, *"Report of the CMS* Stress Test *on the European DataGrid Testbed"* (in preparation)