LHC Grid Computing Project

COMMON USE CASES FOR A HEP COMMON APPLICATION LAYER

HEPCAL

Document identifier:	**HEPCAL RTAG Report**
Date:	**23/03/2004**
Authors:	**F.Carminati (CERN), P.Cerello (INFN Torino), C.Grandi (INFN Bologna), E.Van Herwijnen (CERN), O.Smirnova (Lund University), J.Templon (NIKHEF)**
Editors:	**F.Carminati (CERN), J.Templon (NIKHEF)***
Document status:	**DRAFT v3.1**

Abstract: This document identifies common use cases for LHC applications to use Grid services. It is addressed to the Software and Computing Committee (SC2) of the LHC Grid Computing Project (LCG) and to the various Grid MiddleWare projects in which the LHC experiments are involved.

Issue	Date	Comment
1.0	10/04/2002	First version including the feedback of the first RTAG meeting on April 3-4 2002.
1.1	24/4/2002	Version taking into account the feedback of the RTAG meeting of April 16-18 2002.
1.2	28/4/2002	Draft report delivered to SC2
1.4	08/5/2002	First public draft for comment.
1.5	15/5/2002	Version prepared for the meeting of May 15-17, 2002
1.6	17/5/2002	Version elaborated at the meeting of May 15-17,2002
1.7	24/5/2002	Final version.
1.8	12/06/2002	Modified according to LCG request, recommendations moved to a memo.
2.0	11/7/03	Version modified by GAG.
2.1	9/9/03	Comments from CS11 included (as far as possible…)
2.2	30/11/03	Added the ATLAS DC table
3.0	19/3/04	Final revision by the editors
3.1	23/3/04	New table from CMS on Data Challenge

CONTENT

1 INTRODUCTION

Q: Any suggestions you've got for appropriate books for pretty bright kids at a good liberal arts school would be appreciated.

A: You can throw in Voltaire just about anywhere. Make sure, though, if you do, to say, "*in this, the best of all possible worlds, it is simple to see that* ***we'll just have to rage in and adb the bejesus out of this hideously broken program****. And adb, of course, is the best of all possible debuggers.*" The kids'll love this.

High-energy physics has made a strong commitment to Grid technology in planning for the LHC era of experiments. There are several national and international Grid projects with strong HEP components that are actively building middleware, installing testbeds, and inviting physicists to use them.

HEP requirements for Grid middleware have been previously reported. Some experiments (*e.g.,* CMS) have made thorough studies resulting in quite detailed requirements. There has been very little work done on *common* requirements for HEP Grids, however. Such work is important in the context of the LHC Computing Grid project, since the LHC Grid will have to serve all four LHC experiments. A Grid satisfying experiment-common requirements yields maximum benefit to the LHC community.

This document reports our quest for use cases common to the four LHC experiments. "Use Cases" (as defined by the UML) refer to typical sequences of interactions between the system being considered (in our case, the Grid) and objects *outside* the system (such as physicists, programmers, or programs). These interactions are focused on *goals* (what should the interaction accomplish?) and not on *implementations* (how will the goal be accomplished?). Given a fairly complete set of use cases that are sufficiently free of implementation details, Grid-project architects can extract the real requirements for the system. We see this document, therefore, as a first step to telling Grid middleware projects how to build the Grid we really want.

Chapter 2 provides a quick overview of a typical HEP computing activity. In Chapter 3, we discuss and define the basic components of our use cases, that is, the basic building blocks that our users can interact with. Chapter 4 lists some basic assumptions used in our use case analysis (if these don't hold, the use cases may need revision). Chapter 5 presents a high-level discussion of the Use Cases, which are then formally presented in Chapter 10, while Chapter 9 contains the Glossary and references and Chapter 8 presents the conclusions and recommendations issued from our work.

1.1 MANDATE AND OBJECTIVES

The mandate and objectives of this working group are to:

- Identify and describe a set of high-level use cases of Grid technology common to the four LHC experiments;
- Possibly identify and describe which use cases will be specific for the different experiments;
- Identify a set of common requirements for Grid middleware;

The result should be the detailed description of use cases that must be executable in the distributed "Grid" environment. We do not address in this document the question of whether the needed functionality comes from the Grid middleware or the experiments' frameworks. This boundary will depend on which middleware is used, as well as on when one asks the question; as the middleware projects are actively adding functionality. These use cases should serve to the middleware developers (both in US and in Europe) to guide their work and to the experiments as a platform to perform Grid interoperability studies.

The end product should help in the development of a common set of services for the four LHC experiments to be used on the timescale of the LHC exploitation.

1.2 PREFACE TO HEPCAL "PRIME"

This version of HEPCAL (known as the "prime" version) contains clarifications and corrections. A large part of these are due to the interaction with EU and US Grid projects and in particular during discussions between the HEPCAL team and the European DataGrid Architecture Task Force (ATF). The EDG ATF used our use cases to understand whether the DataGrid architecture was suitable for its intended purpose, and that the Grid-service interactions generated when executing a use case were consistent.

The main additions have been done in these areas:

- In all the use cases where this was applicable, portions of pseudocode have been added. These are to guide the implementers indicating them the kind of interface the user is expecting.
- Devious flow has been added to all the use cases.
- Quantitative requirements have been added based on the Data Challenges figures.

1.3 APPLICATION AREA

The area of application of this document is the offline computing of the four LHC experiments. In particular the way in which the experiments want to access and exploit their distributed computing resources for processing and analysing the data coming from the LHC detectors.

1.4 ACKNOWLEDGEMENTS

The starting point for this document was the work done within the HEP Application WorkPackage of the EU-funded DataGRID project (WP8). We are therefore strongly indebted to the WP8 personnel and in particular to the EU funded people (I.Augustin, J-J.Blaising, S.Burke, M.Reale and J.Templon) for their groundbreaking work (R1) in discussing with the LHC experiments their needs and requirements.

For the description of the use cases we started from the D0 templates (http://www-d0.fnal.gov/computing/grid/use-cases.html). We thank D0 for having kindly allowed us to make use of them.

During the discussions leading to the present document, we have greatly profited from the collaboration of a group of computer scientists from LAPP (Laboratoire d'Annecy-le-Vieux de Physique des Particules): S.Lieunard, T.Leflour and N.Neyroud. Their translation of our use cases in UML diagram gave us a great insight of our work and helped us to converge.

1.5 EDITOR'S REMARKS

It is important to note that while extensive mention was made of Storage Elephants (SE), and Computing Elephants (CE), no animal was hurt or mistreated in the preparation of this document.

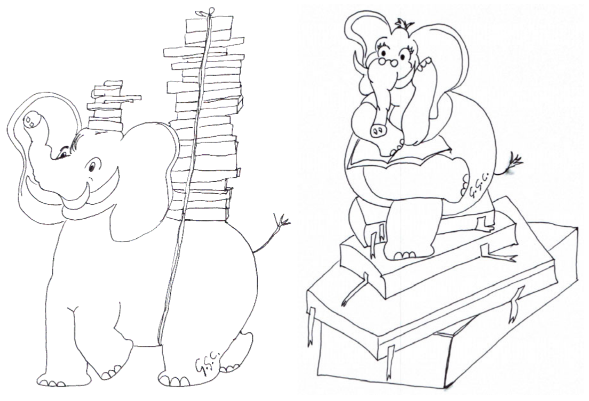


Figure 1: Storage and Computing Elephants

2 THE PHYSICS DATA PROCESSING MODEL

The following sections are an attempt to summarize the experiment independent aspects of data processing. Later, scenarios will be formulated in these contexts, from which use cases will be derived. We first describe the dataflow in the four processes classified as organised (data collection, Monte Carlo simulation, detector calibration and final state reconstruction). Finally we describe the dataflow in the user analysis, which may have a chaotic nature.

2.1 FROM THE DETECTOR TO OFFLINE DATA

In a LHC experiment, data gets collected from the detector's data acquisition system and stored offline (see Figure 2) after processing and selection by the trigger system. Small samples of rejected events are kept for efficiency studies. Collected raw data from the experiments are used to calibrate the detector, i.e. to correlate its response to the actual value of the physics parameters it is supposed to measure. After that, the reconstruction process (see 2.4) determines raw physical quantities such as energy in a calorimeter, assignment of hits to tracks, etc. Reconstruction is repeated a number of times during the running of the experiment to accommodate changes in algorithms, calibration and alignment.

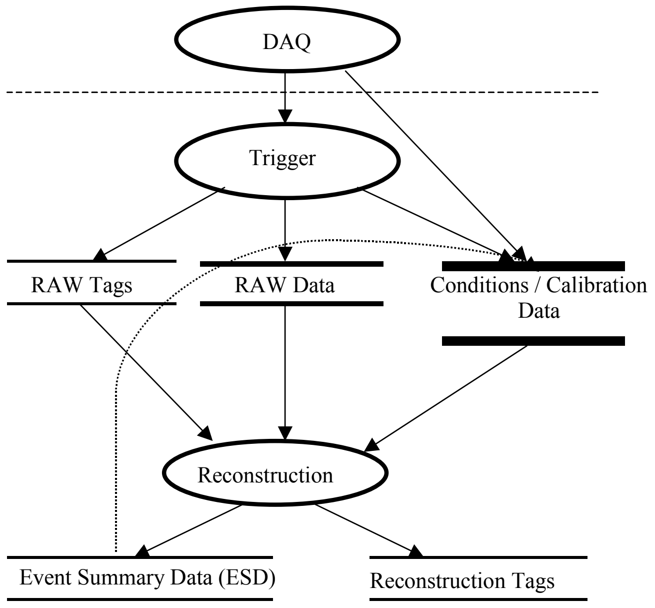


Figure 2 From DAQ to offline

2.2 EVENT SIMULATION

During the LHC preparation phase, all the experiments have large needs for simulated data, to design and optimise the detectors. This "Monte Carlo" simulation is done in the following steps (see Figure 3):

- Particles emerging from the collisions (called collision final state or simply final state) are generated using programs usually based on physics theories and phenomenology (called generators);
- The particles of the generated final state are transported through the virtual detector according to the known physics laws governing the passage of particles through matter;
- The resulting interactions with the sensitive elements of the detector are converted into rates of electronic counters (digitisation) similar to those produced by the real detector;
- The trigger is applied and the events are reconstructed (see 2.4).

- The (Monte Carlo) generated information (sometimes called *truth*) is saved for comparison with the reconstructed information.

Simulation is repeated many times until the detector reaches its optimal design. During the production phase large samples of simulated data are needed to estimate the detector acceptance, the trigger efficiency and extract the physics results and verify physics theories or guide detector upgrades.

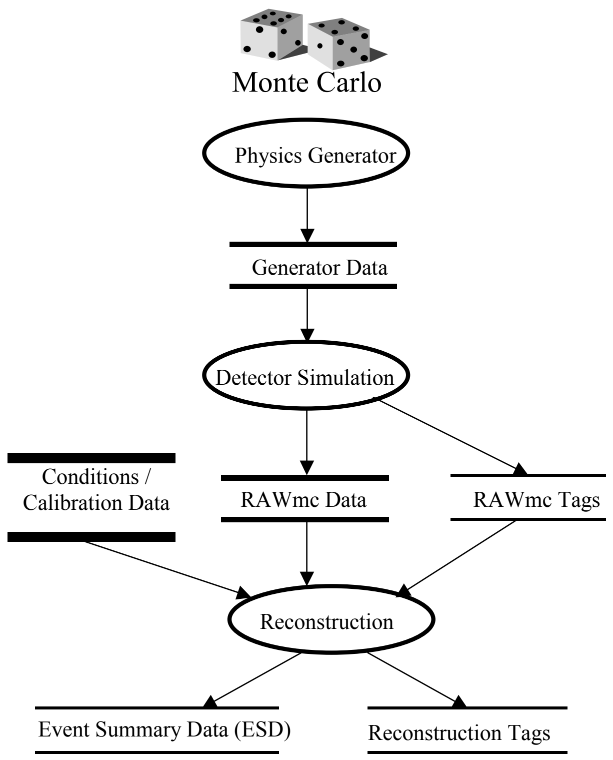


Figure 3 Event Simulation

2.3 DETECTOR CALIBRATION

The procedure that establishes a correspondence between the readings of the electronic counters associated to each detector and the physical quantity measured by the detector is called calibration. It is usually performed using special “events” where a known signal is injected in the detectors and the resulting reading is used to “calibrate” it. The way to inject a known signal varies with the nature of the detector. Laser beams, light and radioactive sources are common ways to generate signals of known intensity and position. Very important is also the intercalibration of the different detectors that measure the same or correlated quantities.

2.4 RECONSTRUCTION

The final state reconstruction consists of the following steps (see Figure 4):

- Reconstruct the positions where particles left signals (space points), and possibly the energy they released or the time of their passage, according to the nature of the sensitive element;
- Perform a pattern recognition to identify particle trajectories;
- Determine vertices, 4-momenta of the measured particles, their invariant masses and identities;
- Run tagging algorithms to characterise events (usually by the presence of a given particle candidate) and generate Tag Collections;
- Analyse Tag Collections to see whether the reconstruction process can be optimised in order to increase the physics content of the reconstructed events;

- Repeat this procedure (typically 3-4 times/year) on the complete input sample as algorithms evolve;

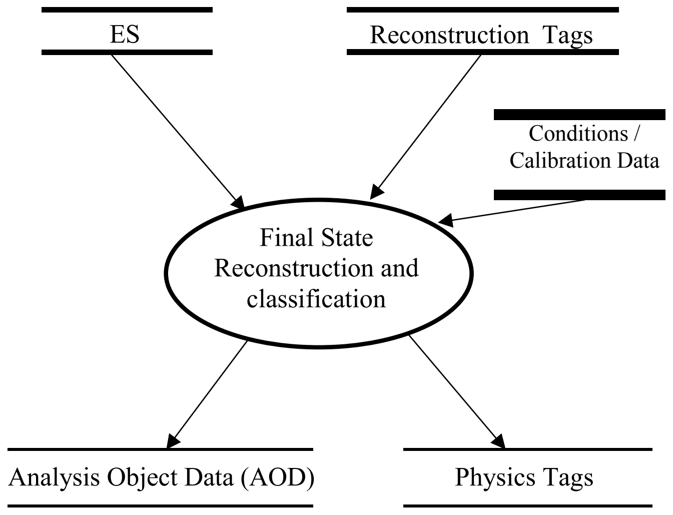

ES
Reconstruction Tags
Conditions / Calibration Data
Final State Reconstruction and classification
Analysis Object Data (AOD)
Physics Tags

Figure 4 Final State Reconstruction

2.5 PHYSICS ANALYSIS

The resource access patterns used in physics analysis (see Figure 5) tend to be less predictable than the one of the processes described in the previous three sections. This comes from the fact that jobs are initiated from almost any HEP site in the world, as well as from the large variation in the "sparseness" of the data access.

In this class of use cases, a physicist runs analysis jobs. She may either execute an inclusive analysis, using all the collected data, or select interesting events using Tags. A set of event containing similar physics is sometime called a "channel". Channels of interest are analysed starting from AOD, accessing parts of the ESD, or even of the raw data, if necessary. The need to access different portions of the data increases sparseness. The generated data may be private to the physicists, possibly with links to full events or other objects located in the official datasets. These data can be stored on private storage or they can be registered on the Grid in a private area accessible to the owner. Systematic effects are studied by looking at the ESD for small event samples. Access to complete individual events (~100) may be required and these are studied in detail e.g. with an event display.

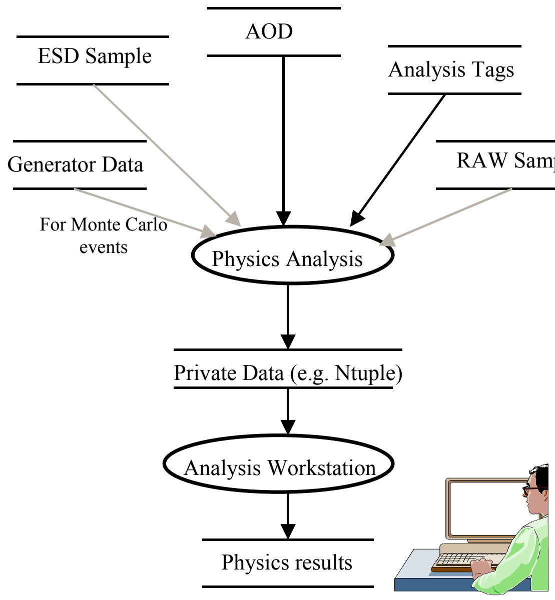


Figure 5 User analysis

2.6 CATEGORISATION OF JOBS

A typical job will perform some calculation on a specified input dataset and will produce some output. It can be interactive or batch and is part of the dataflow explained above. We consider two main cases of HEP jobs:

- Organized jobs. These jobs are planned in advance and perform a homogenous set of tasks. The input is a pre-determined set of events accessed sequentially, processed and then written out, in a different format, suitable for calculations to be performed in a subsequent phase. A production team manages the data processing; simultaneous requests to the same input dataset are minimised by a proper organisation of the production.
- Chaotic jobs. These jobs are submitted by many users acting more or less independently, and encompass a wide variety of tasks. The input is typically a selection/analysis algorithm to be applied to a very large dataset. Users can submit jobs of this kind at any time, simultaneously asking for the common input datasets.

This division is in part based on the differences in data-access patterns of the two types of jobs. HEP jobs typically access, process, and create large quantities of data possibly performing nontrivial processing to take place for each event.

The Grid Workload Management System (WMS) can obviously benefit from being told which datasets will be accessed by a job. Sometimes this is possible, and in other cases the list of input datasets (typically, a set of events) is determined dynamically by the job itself. How datasets are used can also make a difference. While events are recorded sequentially by the data acquisition system of the experiment, jobs do not necessarily access them sequentially, nor do jobs necessarily access all the events in a dataset. The access pattern of events, whether sequential or otherwise, and the fraction of events processed in a file can be used to distinguish the different jobs. Usually a sequential access pattern, where all events are processed in the order in which they are stored, minimises the overhead of file handling with respect to a sparse and random access pattern. However, a job requiring sequential event access spanning many datasets may suffer substantial idle periods waiting for events; relaxing the sequential constraint allows opportunistic processing (“give me whatever you’ve got ready”).

2.6.1 Organised jobs

Organised activities generally involve some transformation of a significant fraction (perhaps approaching 1) of a set of events. Examples are reconstruction (track fitting in the detector), creation of Event Summary Data (ESD) from raw data via reconstruction, production of AOD from the ESD. Sometimes these steps will be repeated several times, but with improved calibration constants or algorithms. Such large organised operations are called productions and are managed by a *production team* and supervised by a *production manager*.

In these cases, the production team knows beforehand which data should be processed. It is also likely that many of these activities could be phrased as "perform reconstruction algorithm C on all events with IDs between 107567 and 2098343, saving the results in the SE at RAL." We note here that specifying a given SE may be useful, since the Grid knows only about the optimisation of the currently submitted jobs (e.g. it does not know the future plans of a collaboration or a production team). The case of simulation production is usually simpler, since there is minimal input data.

The crux of this data locality issue is that the experiments do not need to worry particularly about intelligent job submission, since jobs will actually need to inspect most, if not all, of each input file. Since this is the case, it is reasonable for the Grid middleware to send the jobs to sites containing a large amount of the data needed, or to transfer entire files on behalf of a job. The distribution (among grid sites) of the input data (presumably done according to VO policy) has been shown to have a large impact on the efficiency of the production. As of this writing, this appears to place more requirements on the foresight of the production manager than on the grid middleware. It seems therefore evident that special attention has to be devoted to the middleware optimisation of the location of the input files. With the present middleware, this still requires manual intervention.

2.6.2 Chaotic jobs

Teams of physicists (or even single researchers) process events to search for specific patterns (called *signatures*) that reveal some interesting physics effects. The data access varies from completely unpredictable and sparse to predictable and sequential. Sparseness can even be of the order of one event out of a million. In this case it is not feasible to organize the input data in an efficient fashion unless new files are constructed containing only the selected events. These activities are also uncoordinated (meaning not planned in advance) and often iterative, further hindering the possibility to organize the input data. It is possible that either the Grid middleware, or the users, will need to develop methods for efficient single-event access to avoid inefficiency in data access.

Users will wish some form of interactive access for analysis activities. At the lowest level, this involves access to job output and progress status while the job is running. A higher level of interactivity might be, *e.g.,* dynamic access to the current state of histograms being filled by the job. Full interactivity means having an event display-like session on the screen interfacing with the experiment framework, with the underlying program running across the Grid.

The requirements expressed for interactivity were application-specific and we did not have the time to extract from them common use cases. Merely due to the lack of time and not because this set of use cases is of lesser importance of the ones treated, this document does not address the use cases specifically related to running interactive jobs on the Grid, nor the definition of what a Grid interactive job really is. This issue should be addressed by a continuation of this working group.

During the discussion of "interactive Grid functionality" we realised that applications could implement much of it if the Grid provided a way for a parent process to start other processes under the control of the Workload Management System and open a communication channel (e.g. a Unix socket) with them. In this case it will be WMS responsibility to choose the location where to run the child process depending on the parameters provided by the parent (e.g. location of input data). We realise that this requirement needs further elaboration.

2.7 RELATIONSHIP BETWEEN THE GRID AND THE EXPERIMENT FRAMEWORKS

Experiments want to access and exploit distributed computing resources (the Grid) that may be shared with other Virtual Organisations (VO), for processing and analysing the data coming from the detectors. In this distributed environment they need access to:

- Code and applications;
- Real or simulated data;
- Condition data;
- Metadata of various nature.

The data will be accessed by the experiment specific code, as their format is likely to be application dependent. The relationship between the experiment's framework and the Grid is schematically shown in Figure 6.

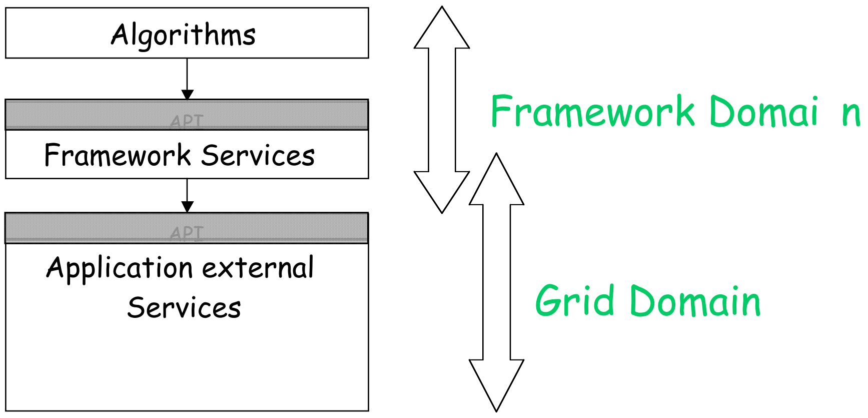


Figure 6 Interaction between experimental frameworks and the Grid

Although the experimental frameworks are different, the objective of this document is to describe common use cases with respect to the functionality and the interaction with the Grid world. This will allow a more precise definition of the boundary between what is common and what is experiment specific.

3 BASIC CONCEPTS

3.1 THE GRID

By the name *Grid,* we understand a widely distributed computing infrastructure, including hardware resources and the corresponding software tools and services, which allow optimal execution of computational tasks, with appropriate access to the distributed data. The implementation of such an infrastructure is beyond the scope of this document. The Grid is assumed to provide proper authentication and authorisation, transparent access to resources, and overall management of the necessary databases.

3.2 USERS

A user is any individual associated with a VO using the Grid services in the process of performing computational work. A user typically submits jobs to the Grid, i.e. requests of work to be done or actions to be taken on his behalf. Sometimes this kind of user is called "end user" to indicate that she is not part of the service-providing infrastructure, but rather at the "end" of the service providing chain. We will simply use the word "user" in this document. Typically, in the scope of HEP users are physicists, engineers and computer scientists working for the ultimate goal of extracting the physics information from the collected data. Other kinds of actors accessing Grid resources are described in the following section.

3.3 ACTORS

To identify a set of common use cases, it is necessary to start with the definition of the actors that are involved in the computing activity of a LHC experiment. The same physical person may play more than one role depending on her activity at a given moment. We identify the following actors:

- Users (physics analysis, production analysis, data quality checking, detector optimisation and calibration);
- Production managers: actors possessing the privileges necessary to submit and control large sets of jobs requiring substantial resources, and to update certain official VO-wide databases;
- Experiment database managers;
- Experiment resource managers;
- Managers (spokesperson, computing coordinator, physics coordinator): actors with highest responsibilities, at the top of the decision-making chain;
- Developers of experiment-specific software tools;
- Software maintainers and distributors (librarians).

The actors mentioned above are Grid users involved with the applications; others, such as computer centre managers and owners of resources, are not mentioned here as they go beyond the scope of this document. We discussed at length whether a Grid program or job could be an actor, or if the actor is always the user who initiates it. We concluded that probably it would be better to distinguish between the user and the program or job initiated by the user, but we did not implement the distinction in this document. Therefore when a use case says "Actor: user" it usually can be taken to mean "user, user job, or user program running on a Worker Node (WN)."

We do not discuss explicitly authorisation issues and the related use cases. In particular we imagine that there could be a way for a user to assume a "role" and become a specific actor.

3.4 USE CASE OUTLINES

This is a basic set of high-level sequences of interactions between users and the Grid from which we derived our use cases, organised by main actor:

- Users:
  1. Basic physics analysis task: a user wants to run some algorithms on a selection of the data (either real or simulated), build private collections of data, fill histograms, possibly inspect some events looking at an event display, apply cuts, refill histograms and possibly look at the event display again;
  2. A user submits a job to generate a collection of Monte Carlo events of a certain class, with given versions of the simulation code and of the detector description;
- Production managers:
  3. A user doing production analysis systematically processes all the events of a certain kind, creating new data samples;
  4. A user doing data quality control runs appropriate algorithms over all produced data and fills histograms, checking for deviations from reference data;
  5. A production manager submits a set of jobs according to some criteria and monitors the production for progress and errors, updating the bookkeeping database;
- Database managers:
  6. The person responsible for the conditions database publishes a new official version of the conditions data;
  7. The person responsible for the conditions database verifies that the correct conditions data are available to jobs;
  8. Event database manager after reconstruction publishes a new version of production ESD and AOD;
- Experiment resource managers:
  9. A resource manager decides how resources (disk space, queues) are allocated inside an experiment;
- Managers:
  10. A manager, for instance the physics coordinator, monitors the progress of a given production looking at the statistics of data produced and processed on the Grid;
  11. A manager, for instance the computing coordinator, performs supervisory functions and modifies priorities or other resource allocations;
  12. A manager, for instance a physics coordinator, approves the outstanding production requests and their allocation of priorities;
- Software developers:
  13. A software developer needs to test that the software works on the Grid;
- Software distributors:
  14. A software distributor, as the program librarian, releases and registers a new version of a package, making it available on the Grid;

3.5 FILES, DATASETS AND CATALOGUES

We introduce a rational naming scheme to avoid clashes in understanding with members of the various experiments. For example, while to some people the word *database* invokes a vision of a large table, each row having a key and various associated attributes (like a classical relational database), for others a database might be a file containing event data (a ROOT "tree" is a database).

We distinguish two logical entities containing data: *catalogues* and *datasets*. A catalogue is a collection of data that is updateable and transactional. A dataset is a read-only collection of data. A special case of the dataset is the *Virtual Dataset* described below, which is associated with all the information (algorithm and input data) needed to produce it. *Datasets* or *catalogues* might be implemented as one or more files; however they might also be implemented otherwise, such as Objectivity or Oracle databases.

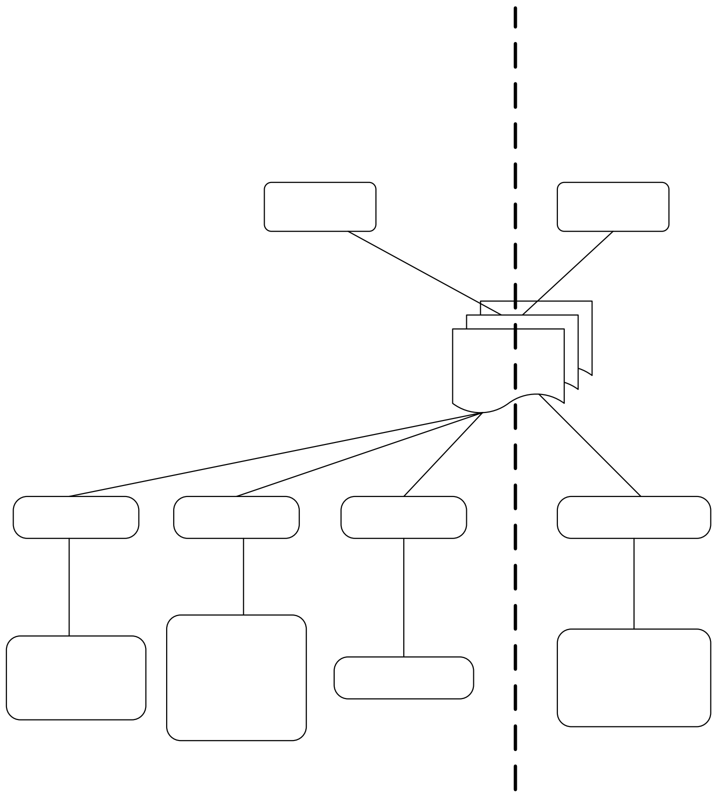


Figure 7: Catalogues and Datasets classification

3.5.1 Datasets

A *dataset (DS)* can be any sort of collection of information. Examples are a set of histograms, a file containing a number of raw events or processed events or some other type of data useful to an experiment such as the list of conditions used in a particular analysis, or a collection of pointers to events of a certain type. In certain circumstances, a software package used by an experiment can be considered a dataset as well, if the Grid manages it. Datasets in our context have one important property: *if they are registered in the Grid, they are write-once, read-many (WORM) objects.* A dataset can be written to only at its creation, and thereafter can only be accessed for reading. It lives forever on the Grid unless explicitly deleted.

When a dataset is registered on the Grid, it is registered with a unique logical dataset name (LDN) that users or programs will use when referring to the dataset. The Data Management System (DMS) keeps the association between a dataset and the files that compose it. This information can be stored and retrieved by the user. Multiple copies (physical data sets) may reside at various locations, managed by the DMS, and these are guaranteed to be identical to each other. A LDN must be unique within a given VO, and it must be unique forever to make sure that dependencies among DSs are not erroneously satisfied. The naming rules are VO policy, however to ensure uniqueness we realize that the DMS may need to define part of the name. An example of VO naming policy might be to construct part of the name by a combination of the data acquisition date and of the chain of analysis that produced the dataset being named.

The way in which LDN uniqueness is ensured is partly implementation dependent. In line with the HEPCAL guidelines we will not discuss this item any further, apart from noting that LDNs should be, as far as possible, *humanly readable.* Whatever solution is adopted, the problem of the erroneously resolved references should be seriously considered.

It is assumed to be possible to associate a default access protocol (to be used for remote access, *e.g.* root daemon for ROOT files, AMS for Objectivity/DB files, etc) to a dataset entry in the DMS. The Grid takes care of replicating the dataset only to a SE that supports that access protocol if remote access is required.

The DS shall not disappear from the Grid if not explicitly deleted by a user with appropriate Grid privileges. The rights of the local system manager could be possibly restricted when dealing with Grid DS. As an example, uncontrolled dataset deletion could result in loss of data, if a system manager were to delete a dataset's sole physical instance (or master copy, if implemented). This would entail the disappearance of the DS from the system. Possible exceptions to this are Virtual DS (see below). It should be possible to delete data from the Grid, but a job can crash if data is deleted after it was requested to the DMS but before the execution finished and there should be a way of avoiding such situations.

The Grid manages all the files composing a Dataset as a single entity, e.g. it is not possible to replicate a fraction of a dataset.

An additional requirement is that a dataset may be composed by other datasets. This means that the list of files composing a dataset is the same as it would be found by recursively resolving the composition of all datasets composing the target dataset. We include more details on this topic in a following subsection.

A dataset may contain references to objects belonging to other datasets (e.g. the AOD depend on the corresponding ESD to allow navigation to the unprocessed information), therefore it is not possible to predict the complete list of datasets a job will need to access. So a small dataset can depend on a large number of datasets. A typical example is a *user-defined* tag DS that has links to selected events of a given kind out of the whole period of the experiment data taking.

Files belonging to a dataset should be made available for opening via a POSIX call or an application specific remote access protocol. The Grid should provide a mechanism whereby a user can present a LDN and receive in return a list of physical file names (and possibly the protocols by which they can be opened) that can be mapped to the original files that were uploaded to the Grid.

Datasets specified at job submission time in the Resource Specification Language (RSL) have to be considered an optimisation hint to the WMS. Applications should be able to open any Grid dataset independently of whether it has been specified in the RSL, or requested directly by a running task.

A dataset may depend on other datasets in two ways:

1. A dataset is composed of one or more datasets;
2. A dataset is required by a virtual dataset (see next section) for its materialisation;

The Grid should be able to check these dependencies. A third dependency mentioned above is when an object in a dataset has a link to a portion of another dataset. We do not require the Grid to know about this dependency.

3.5.2 Virtual Datasets

An extension of the dataset concept is the *Virtual Dataset*. A virtual dataset is different from a "normal" dataset in two respects:

1. The procedure (here an algorithm) to produce it is known by the Grid and can be retrieved and executed. The algorithm includes any input datasets needed, as well as the program (including version information) through which these data must be run.
2. Since the Grid knows the complete specification of how to create the dataset, an actual physical copy is not required to exist. If the last such copy is deleted, a program can still access the dataset by regenerating it.

We assume that the Data Management System (DMS) will provide a method to calculate access costs for a dataset, possibly based on the information provided by the user together with the materialisation instructions. The idea behind this is that it may be more advantageous to generate a local physical copy of a dataset than copying it from elsewhere. The DMS must then provide information on the relative costs of the various options. This cost might be expressed in terms of CPU, bandwidth, and elapsed time. This is in fact an implementation issue, beyond the scope of this document, however if this information is present, users should be able to browse it.

Before any application tries to access it, a virtual dataset is "completely virtual". After it has been materialised, there exist actual physical copies. The algorithms for providing access to the dataset in these two cases should be straightforward. However consider the case where a user submits several (say 20) jobs within a short interval (compared to the expected materialisation period). One option is that the Grid creates 20 copies of the material dataset (perhaps with errors if more than one job goes to a given site). Another is that the materialisation request is logged in the DS metadata catalogue, essentially locking subsequent jobs out until the materialisation is complete. Whatever the implementation is, it has to satisfy the following requirements:

1. The materialised LDN is the VDS name;
2. DMS should store accurate information after the materialisation finishes;
3. Jobs should not fail when trying to access a DS being materialised.

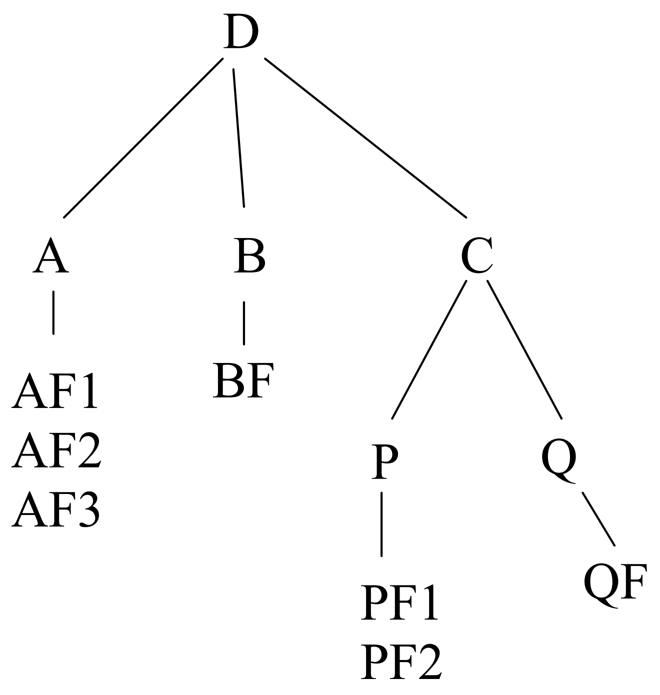
3.5.3 Compound Datasets

A dataset is logically a single entity, but may be composed of multiple lower-level objects. There are three cases; we list them here. Here we list how a *single physical instance* of a dataset can be constructed. We therefore try to explicitly avoid confusion with including all the various instances of the dataset. We revisit this point after listing the possibilities.

1. A dataset can be composed of a single physical file.
2. A dataset can be composed of a collection of several physical files.
3. A dataset can be composed of a collection of several datasets.

In item 1, since there may be several replicas of the dataset scattered over the grid, one might conceivably state that the dataset is composed of multiple physical files, but each *instance* of the dataset will comprise a single physical file, which contains exactly the same data as the other physical files corresponding to other replicas of this dataset.

A dataset has the semantics of a homogeneous list. In case 1, the list has one element that is a particular physical file on the Grid storage location that "hosts" the replica in question. In case two, the list has several elements, each of which is a particular physical file on the hosting storage location. In case three, the list has several elements, each of which is a particular physical instance of another dataset on the hosting storage location. *In particular, for a DS that does not contain other DSs, this implies a requirement that all its elements be located on the same storage element.*



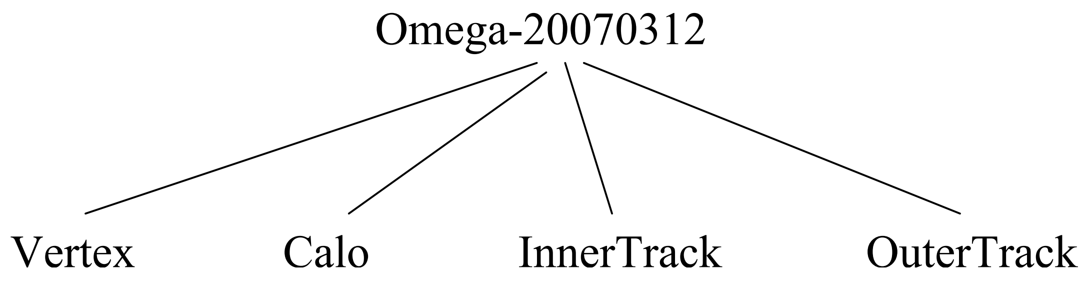
The figure shows examples of several of these cases. Case 1 is illustrated by dataset "B" which is composed of a single file "BF" (the "F"s in this picture refer to actual physical files), and by dataset Q. Case 2 is illustrated by datasets A and P which each consist of multiple physical files. Case 3 is illustrated by datasets C and D which each consist of multiple datasets.

We said above that "the list of files composing a dataset is the same as it would be found by recursively resolving the composition of all datasets composing the target dataset." In the picture above, this means that if we requested a copy of dataset C, we would receive copies of PF1, PF2, and QF. The question of how the different components are identified by the requesting application is still open. Usually it is assumed that the names originally given to the files composing the dataset are preserved, and that it is up to the application to deal with them.

NOTE: A compound dataset can include Virtual Datasets. A compound dataset is considered virtual if any of the component dataset is virtual.

3.5.4 Dataset File Naming Issues

Part of the motivation behind the dataset structure is the expected multi-file nature of HEP event data. For example, events for a given run might be partially resident in several files. A generic HEP experiment may have a vertex detector, a calorimeter, and inner and outer trackers. The "vertex" part of each event's data will be in the "Vertex" file, that of the outer tracker in the "OuterTrack" file, *etc.* This structure of a dataset is illustrated in the figure below.



If a physicist requests the dataset Omega-20070312 and wants to read vertex events, he wants to be sure to open the file "Vertex" and not something else. This means it must be possible to "label" the various components for identification later.

One could take a unix-like approach and make the DS name like a “directory”, and the component files be like the “files” in that directory. We were not able to decide during HEPCAL if this approach was good, since the problem of files/directories is not exactly equivalent to DS/components and making it look like Unix might create more problems than it solves. We are eager to see proposals on how to address this situation.

3.5.5 Catalogues

A *catalogue* on the Grid is a collection of data that may be updated. Several users may be simultaneously updating a catalogue; therefore catalogues must be fully transactional in nature. The Grid chooses the technology used to implement a catalogue. In general, a catalogue on the Grid has the same connotation as a catalogue in real life: it contains information *about* objects, but doesn’t contain the objects themselves. An example is the Replica Catalogue of the EDG project. It contains a list of all the files registered for the VO owning the catalogue, and for each file it contains a list of its physical locations along with other information useful for replica management purposes, but it does not contain a copy of the file itself. There may be instances of a Grid catalogue for which this doesn’t hold, but we hope this guideline is useful in most situations. Most HEP jobs will need to read and write data from/to one or more catalogues during their lifetime. Temporary network interruptions must not cause job failures due to inaccessible catalogues.

We do not address in this document the issue whether a catalogue is replicated on the Grid, as this is an implementation issue. From the user point of view, replication is not so interesting for performance (access bandwidth) since the amount of data being transferred is probably relatively small. Something like replication might help, however, to prevent jobs failing because network problems render the central catalogue inaccessible. We assume that if a catalogue is replicated, the Grid itself maintains consistency between the replicas, possibly using a “loose” synchronization method that prevents user jobs from crashing because of synchronization problems. We identify two kinds of catalogues.

Grid-managed catalogues. These catalogues are required to be part of a VO. They are not created or deleted by the users, and the Grid, as the result of some user-initiated operation, updates them, or the user updates them directly. We consider these logically unique within a VO, even if they can be implemented in several parts. One question that we realise is very important but that we have not treated is who has the authorisation to change the information in the catalogues, which are the *level* of authorisation and how these are granted or revoked. One very general remark is that we do not see any requirement for members of one VO to access the catalogue of another VO[1]. Some experience with Grid catalogues will be necessary to express meaningful requirements and use cases in this area. In this document we identify the following Grid managed catalogues:

- **DS metadata catalogue** (see 4.4). Contains meta-information about datasets. Two parts logically compose this catalogue:
    1. A user-defined part that contains meta-information describing the content of each dataset or information to materialise it in the case of Virtual Datasets.
    2. A Grid-specific (middleware-defined) part that contains, for instance, information about the replicas of each dataset. This second part is sometimes referred to as *Replica Catalogue.*

  We make no assumption on how this is implemented.
- **Job catalogue** (see 3.6). Contains information about Grid jobs. When a job is submitted, the WMS adds the corresponding entry in this catalogue, indexed via the job identifier.
- **Software catalogue** (see 3.7). Contains a list of all officially installed software belonging to a VO.

[1] In case of multiple cooperating grids a reasonable use case would be to access the catalogue for the same VO in a different grid, however this is outside the scope of the current document.

- **Catalogue of Grid users** (see 3.2). Contains Grid relevant information about each Grid user, such as privileges, accounting information, authorisation and authentication credentials.

User-defined catalogues. The user can create, update or delete them. Again, the Grid should make sure that these catalogues are fully transactional. These catalogues are identified by a "logical name", which is location-independent in the same sense as the logical dataset name described earlier. It should be possible to download a catalogue from the Grid (i.e. make a copy of it outside the Grid). This can be seen as an *export* operation, where the content of the catalogue appears outside the Grid with a user-specified technology. An example could be to *export* the content of a user-defined catalogue into an Oracle RDBMS outside the Grid. Similarly, an *import* operation could be useful. The main limitation with this approach is that the Grid controls the technology that implements a vital part of the data of an experiment.

We realise the relative incompleteness of our description of the requirements for catalogues. We think that more discussion is needed both within the experiments and with the middleware developers on this issue. In view of this we have only introduced very basic use cases for user-defined catalogues.

3.5.6 Read-write datasets

As already said, for the purpose of this document datasets are read-only. We have considered the need of read-write datasets to implement some of the user-defined catalogues discussed above. The important advantages of such an implementation are:

- The implementation technology is chosen by the application and not by the Grid;
- The catalogue can be directly uploaded to the Grid, rather than having a forced reliance on creating/updating/deleting the catalogue via special Grid commands or API's;
- The catalogue can be replicated on user request as a dataset;

We realise that this would mean to ask for a fully transactional dataset, implemented with any possible technology indicated by the user and that can be replicated on the Grid. We leave this issue open and in the following we only address the use of read-only datasets.

3.6 JOBS

By a job we understand a single invocation of the Grid submission use case. A basic job definition consists of a set of input data, executable(s) to process them, and a set of output data. There can be other parameters, specified via the RSL or otherwise, indicating requirements to the WMS.

The simplest example of a Grid job is a request for Grid resources that results in a single instance of some program running on a single computer. A "single instance of some program" includes the case of a shell script that might, during its execution, run several programs itself. The essence of the basic job is that the WMS is managing only one object.

These basic jobs can be combined to form more complex jobs. Such job systems are referred to variously as "processing chains", "workflows", or "processing pipelines". The basic idea is to split the computational task into various steps. Each step becomes a job, and these jobs can be run sequentially or possibly in parallel on more than one node according to given workflow dependencies. These dependencies are typically expressed as Directed Acyclic Graphs (DAG) where each node represents a job. An example of a sequential chain of dependencies is a Monte Carlo chain: "generate events" → "propagate events through detector model" → "generate hits" → "reconstruct hits". The Grid should be able to understand these dependencies, allowing a user to submit such a processing chain as a single job. The Grid then handles the overall optimisation of the chain, and manages the execution of the component jobs.

The events that are processed in HEP are, as alluded in section 4.7, generally independent of each other. This makes it possible to process them independently. When the number of events being processed is large enough, a processing step could run faster by splitting the task. Several independent instantiations of the processing program can each process some fraction of the total event sample. This feature of HEP jobs is one of the primary motivations for using Grids, and we term the Grid version of this operation *job splitting.* Whether the WMS or the user is going to split them in sub-jobs is, at the moment, an open question.

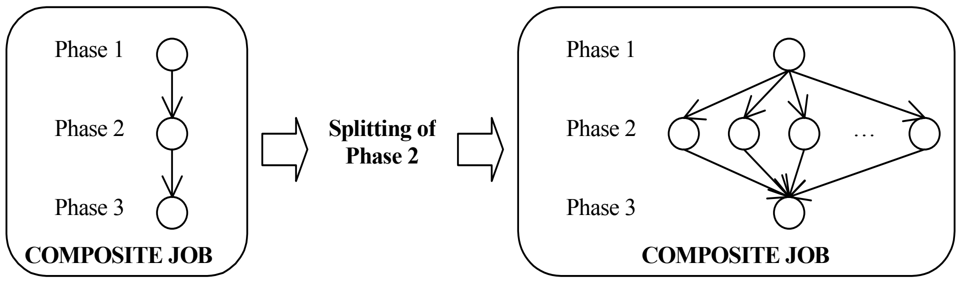


Figure 8: An example illustrating both processing chains (left panel) and job splitting (right panel).

Splitting may be indicated to the Grid by specifying explicitly the dependencies (for instance via a DAG) in the job request. Alternatively the Grid can derive the splitting opportunities from the job input, via an appropriate syntax. A very important subset of job splitting, and likely the dominant case, involves splitting based on the location of the input DSs, processing these independently and concatenating results, e.g. merging the output datasets from each job. In case this is handled by Grid middleware, the user will logically submit a single job to the Grid; the WMS will arrange the partitioning of the job into sub-jobs and manage them as a logical whole. We have identified three models for job splitting:

1. The Grid performs job splitting without user assistance. This can be based on the location of the replicas of the input datasets indicated in the job. Join of the results of the different sub-jobs is in general not possible because the join procedure is application dependent.
2. The Grid performs job splitting with user assistance. A plugin is provided by the application to help the Grid decide how to split the job. In this case more splitting options are available (e.g. distribution of analysis of events from a single input dataset to several sub-jobs). Join of the result is also possible, provided a join plugin for the application results is available.
3. A more effective result could be achieved if a process can spawn other processes on the Grid to be run in parallel under control of the WMS. Upon completion the spawned processes can communicate to the parent process the results to be joined. This implies that the application does dynamic splitting of data processing and that the Grid handles inter-process communications.

As of this writing the requirements for job splitting have not been thoroughly investigated, and more time should be devoted to this issue.

A job is expected to be able to access any of the required datasets at any given time during its execution, and to register the output datasets into a DMS, thus materializing new data on the Grid.

An important case is that of the *production job.* A production job is not different in nature from a normal job, apart from the fact that it is supposed to produce results that are in some sense *official* within a VO, possibly identified by a special identifier (the “production ID”) assigned by the experiments. Production jobs are usually large chunks of work, and therefore they make excellent candidates for job splitting. We expect that most production jobs will be split into many chunks, but this is certainly not necessary – production jobs should not be seen as a special case of job splitting. Given the large resource consumption involved, *production jobs* are usually submitted by a user with special privileges, called *production manager*. Production jobs are very important in HEP data processing; however, their difference being mostly organisational, we will only mention them when their specificity generates requirements for the Grid.

3.7 SOFTWARE

The software mentioned in this document refers to that developed by the experiments, unless explicitly specified. The software can be considered a dataset, since software is essentially "executable data." Just as the job requires some specific input files in order to run, the job will also require some specific executable (or perhaps a certain version of the complete software distribution for an experiment). Jobs must migrate to a computing resource that satisfies user requirements. These can be to minimise the cost of data access and program execution, or to minimise the time to obtain the results.

Our vision for the Data Management System is that automatic replication in support of job execution will result in dataset replica distributions which follow usage patterns. We expect this to result in "popular files" (those which are requested by many jobs) being replicated to many sites, while "unpopular files" will be physically present on only a few storage resources. This should be applied also to software, to reach a similar situation where "popular versions" of a software distribution will be present on many sites, while "unpopular versions" will be installed on only a few sites. The system should also take into account multiple versions of software packages. When the Grid is heavily used, it will not be sufficient to have e.g. a single ATLAS package distributed to all Grid sites!

Recent experience has shown a real need for disconnecting the "system" software, "middleware" and "application" realms. A proposal for how to do this should be combined with a proposal to use DMS functionality for management of "software datasets" and some work on authorization for installation of software and its implications for permissions on pooled accounts.

Software packages may have dependencies among them, so that the installation of one package may trigger other installation in a chain. The issue here is who is responsible for determining these dependencies and satisfy them, the Grid or the experiment/VO. We expect the user to provide dependency checking and solving packages, and to instruct the DMS about which DS to replicate where. There also must be some method of publishing information on which packages are installed where. We expect different binaries to be published for different hardware types. A standard software-publishing job is expected to move binaries, shared libraries, scripts, configuration files and so on (in general whatever we need installed to get a working program with some pre and post installation operations of course). If compilation from sources is needed, then an ordinary job should be sent.

We assume that the Grid will not act at the object level but only at the dataset level. The data persistency layer used by the application manages the access to objects. However the data persistency layer and the Grid need to collaborate in the navigation to objects stored in different datasets. As stated in the "Persistency Management RTAG report to the LCG project", given an object reference, the persistency layer provides the LDN where the object is stored, while the Grid provides the access to the physical files given the LDN. The combination of the two allows the persistency layer to access the relevant data via a POSIX open or via the appropriate remote access protocol.

4 BASIC ASSUMPTIONS

The use cases described in this document are *common* between LHC experiments. In this section we explain the characteristics that we assume to be true for the software and data structures of all the experiments.

4.1 DATA STRUCTURES

The dataflow described in 2 discusses how the data structures are created. The common data structures are listed in Table 1 and depicted in Figure 9.

Event object group	Description
Raw data	Information coming out of the experiment data acquisition system
Monte Carlo data	Simulated information, logically equivalent to the Raw data
Event summary data (ESD)	Reconstruction information detailed enough to display events, generate analysis objects, and redo most of the reconstruction.
Analysis object data (AOD)	Physics objects, e.g. electrons, muons, etc., used for analysis
Event tag	Brief information allowing a rapid first-pass selection to find events of interest. It may contain pointers to events, including the DS where they are stored

Table 1 Data structures for each physics event used in the offline environment

The data definition and the data content of these groups will be specific to each experiment. These structures apply both to real data and Monte Carlo simulations.

4.2 EVENT IDENTIFIERS

Experiments will have a way of unambiguously identifying events via an Event Identifier (EvtId). In some cases this EvtId is assigned by the data acquisition system or by the simulation production tools. In others it can be constructed by a specification of the combination of event number and run number (Evt #, Run #). This identifier will be left unmodified by any possible processing of the event. The EvtId therefore identifies all instances of data structures corresponding to a given acquired (or simulated) event. Said differently, this EvtId will be present in all derived products for that event.

4.3 MAPPING OF EVENTS ONTO DATASETS

An event is composed of objects contained inside one or more datasets. There will be an experiment dependent way to navigate from events to objects stored in other datasets. This depends on the persistency solution used and for some experiments this is still undefined.

4.4 IDENTIFYING DATASETS

Experiments have the following requirements on naming of LDNs:

1. Users must be assured that once their LDN has been registered, another user cannot assign this same LDN to another dataset;
2. The user should have as much freedom as possible to specify the LDN (may not be complete freedom because of 1);

The user is also going to locate data by using high level queries like: "give me all the datasets corresponding to events acquired during the period 22/11/2007 through 18/07/2008 using the XYZ trigger configuration". Datasets are located via a special catalogue that contains information about data sets: the *DS Metadata Catalogue*.

The DS metadata catalogue is accessible by the user in read/write mode and is indexed by the logical dataset name. We assume that the DS metadata catalogue will be part of the Grid Data Management System. When a data set is created or deleted, the DMS always adds or removes the corresponding entry in the catalogue.

Users need to add meta-information about datasets to the catalogue. For each LDN in the catalogue there will be a list of attributes in the form of *key=value* pairs. Users can modify the value of each attribute, possibly subject to authorisation. We foresee three solutions, increasing in both flexibility for the user and potential complexity:

1. Attribute name and type are predefined. Each attribute has then a meaning that is potentially different for each VO.
2. The schema of the catalogue (the list of attributes) can be defined at the VO level.
3. Users are allowed to add and remove attributes. The question of privileges or permissions in this regard requires more thought. Whether the DMS implements this functionality adding fields modifiable by the users to an existing catalogue or creating a new catalogue is considered an implementation issue.

A query on the attributes will return a list of matching LDNs.

Special keys in the DS metadata catalogue are foreseen to be understandable by the Grid component responsible for virtual data set materialization. The values of these special keys will contain the materialization instructions in a specified format (e.g. Executable=… StdIn=…, etc…).

4.5 EVENT METADATA

If a dataset is going to contain more than one event, we assume that the experiment will develop event metadata collections (also called event tag collections). Such collections will help selecting events with queries like: "give me a list of all EvtIds corresponding to bbar events acquired during the period 22/11/2007 through 18/07/2008". Since we do not foresee in this document that the Grid will operate at the object-level, the implementation of these collections is application dependent and will not be discussed further. This issue is explored in much greater detail in the HEPCAL II document.

4.6 CONDITIONS DATA

The conditions database contains calibration and conditions data for an experiment. These are the conditions that vary with time, temperature and pressure and are necessary for reconstruction and analysis. These conditions are stored with a validity range (typically time or run number) and several versions for a condition can exist at the same time (maybe with a different validity range) as a result of new evaluations. A label identifies each version of the database and different data items can have different labels to allow selecting the correct version of each data item valid at each time.

Implementing condition databases or event collections as catalogues would offer transactional facilities, i.e. Grid users would be able to update the data concurrently. However some users may want full control of replication of data accessed by their jobs, including making private copies of these data or exporting them to non-Grid environments. Our definition of catalogues does not address these questions, as we did not tackle the issue of replica synchronization. For a more complete discussion see 3.5.

4.7 EVENT INDEPENDENCE

For the purpose of processing, we assume that events are independent. This implies that the order of event processing is irrelevant, and that the processing of a given event does not require information from a previous event. Simulated events violate this assumption, as each event depends on the status of the random number generator at the end of the preceding event. However, once they are produced, also simulated events can be processed independently.

4.8 DATA ACCESS PERMISSIONS

Confidentiality requirements are weak for production datasets. Unauthorised modification or deletion of data must be controlled, and read-only data access is subject to the experiment policy. Users will want to keep more private data sets on the Grid, which should not by default be readable to other collaborators, even collaborators in the same experiment.

4.9 JOB INFORMATION

Often a user submits many jobs. This is particularly true in case of official productions. The Grid assigns a unique identifier to each job. We have identified three classes of job identifiers.

1. *Basic job identifiers* are assigned to the simplest type of job as described in section 3.6.
2. *Composite job identifiers* are assigned both to composite jobs (corresponding to processing chains) and to any basic jobs that have been split by cooperation with the Grid. The only requirement in this case is that each of the components of a composite job carries a basic job identifier, and it must be possible to use a composite identifier to refer to the complete set of component jobs. For example, if we provide a composite identifier to a Grid job-monitoring tool, it must return information for all the component jobs.
3. *Production job identifiers* are logically different than the first two. The four experiments see productions as the collective submission of many jobs by production teams. The jobs can be tagged as belonging to a given production by reserving a "Production ID" field in the job catalogue (described below) and filling it with a common tag for all jobs in a given production. This will simplify for the experiments the management of production jobs as a group.

We have not investigated the "Grid-process spawning" option in enough detail to determine whether a sort of identifier is required.

Users will often wish to retrieve information about the jobs they have submitted. The Grid should provide a *job catalogue* allowing the user to place queries like "give me the status of all the jobs I submitted that are analysing this dataset". When a job is submitted the WMS always adds the corresponding entry in this catalogue, indexed via the job identifier.

The user should have a unique entry point into the job catalogue. When a user wants to query the job catalogue for a list of all jobs of a certain type, she should not need to know information such as the hostnames of the resource brokers or the sort. There is a weak preference (not a requirement) to have a VO-specific job catalogue. Later it might make sense to partition the job catalogue within a VO according to VO subgroups, although the jobids should still be unique across the entire VO. It does not seem to make sense to have the catalogues accessible across VOs from an experiment point of view.

Similarly to the dataset metadata catalogue, the Grid should allow the user to add private information to this catalogue for each job. For each job identifier in the catalogue there will be a list of attributes in the form of *key=value* pairs. Users can modify the value of each attribute, possibly subject to authorisation. The possible access policies for the users are similar to those of the DS Metadata Catalogue. A query on the attributes will return a list of matching job identifiers.

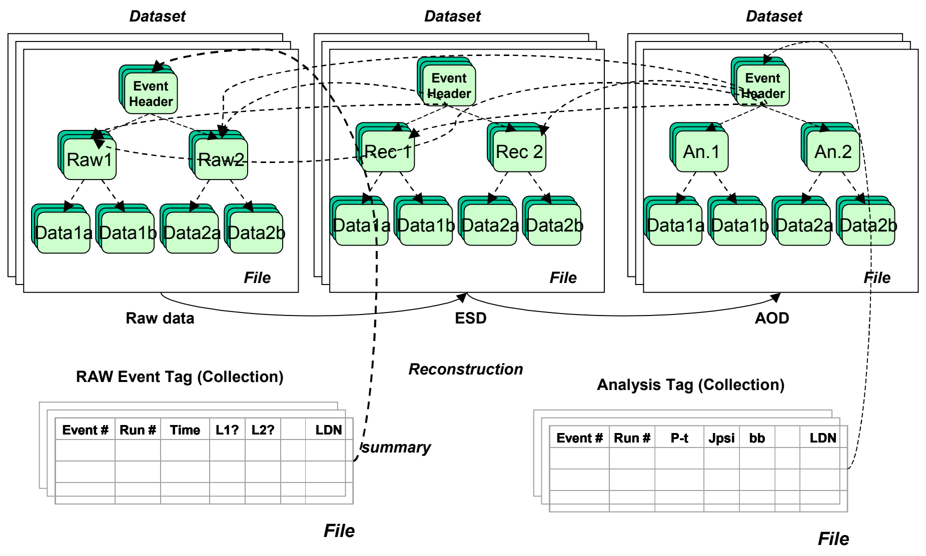


Figure 9: Event data structures and their relationships. At the top, three datasets are shown. Within each group, there are multiple planes (white rectangles), depicting how a dataset can consist of many files. The events components are also represented as stacks, depicting how each dataset may contain many events. The relationship between events and files is not defined; the components of a single event might be distributed over several files. Continuous lines represent processing relations and dashed lines represent references that can be used to navigate from one object to the other.

5 USE CASE DESCRIPTION

In the following section we describe the various use cases. Afterwards the use cases are presented in a more rigorous tabular form; our goal is to provide something concrete enough to be translated unambiguously to a formal language such as UML.

In this document we provide examples of APIs and calling sequences, but these should not be taken as actual suggestions to implementers. However, we want to state clearly our requirement that these use cases should be simple to execute. Especially at the lowest level — meaning those use cases that do not include other use cases — we see the use cases being executable by a single call. The number of calls (visible at the user application level) should in any case *never* exceed the number of steps listed in the "basic flow" section of the use case's tabular representation. We also require that methods should be ultimately provided to execute each use case using all[2] the following interfaces:

1. From the command shell;
2. Via a C++ API;
3. From a Web portal.

5.1 DEVIOUS FLOW

Introducing devious flow is somewhat misleading, because many things can go wrong, and it is virtually impossible to list them all. Most of the things that can go wrong are the same for all the use cases. An extreme example is an electrical power cut. This would make most of the use cases fail severely, and it is clear that we do not want to mention this possibility in each use case. In general it seems justified to consider only those events that can adversely affect the actions and steps specified in the use case flow. Even there, however, it is certainly not enough to simply state "this can go wrong", but, to make this an useful exercise, we have tried to indicate the action that we expect from the system in response to a devious flow.

A generic case of failure for all commands is that the user is not authorised to perform a specific action. Probably some use cases should be developed here, but we did not felt we had enough grasp of the problem to produce concrete use cases. Examples could be

- A user requests to be authorized for a given action;
- A user tries to find out why she is not authorised;
- A user marries, changes her name and suddenly is not authorised anymore;
- A user has more than one certificate and belongs to several VOs.

Another devious flow common to all commands is that of invalid syntax provided by the user. The system response should be as clear and understandable as possible. The kind of error and how to correct it should be clearly indicated. The answer should also point to the available documentation. Unix commands do this in a very terse way with message of the kind "see man <command> for more information". This is generally enough as long as the information referred to exists and is complete.

Another very general devious flow involving the creation of output files is that the user or the VO does not have enough quota on the chosen output location. If the output location is specified explicitly by the user, then a message should be sent back to the user, indicating the reason of the failure and optionally how or to whom ask for more quota. If the location is transparent to the user, as for instance the creation of a DS replica on a SE, then we expect the Grid to take appropriate corrective action to avoid a crash of the user job in most cases. We will omit this devious flow, as it is always the same wherever it applies.

[2] We realize that there may be a few use cases for which not all three interfaces will make sense. A user will probably not want to "obtain Grid authorization" using a C++ API, for example.

A basic requirement is that the system is left in a "clean state" when an operation is aborted. We try to assume very little about implementation issues in this document. However we understand that operations in a distributed environment are far from atomic and synchronous. Nevertheless, whenever a command encounters a failure condition and has to be interrupted, the state of the system has to be "rolled back" to the one before the command was issued without explicit user intervention. Let's take an example. A user wants to upload a new DS to the Grid. It is plausible to assume that a possible implementation would create and fill an entry in the DMC and then upload the physical files to the chosen SE. If the upload fails, then the Grid has to recognise that, and remove the information about this DS in the DMC, not to leave the system in an "inconsistent" state that may cause user jobs to fail. As this requirement is common to all devious flows we will not repeat it any further.

Usually when the system cannot perform an operation, it notifies the user. It is of capital importance that the error messages and the return codes are as precise as possible in identifying the reason for the failure. We require also that all failures be properly logged in some monitoring and bookkeeping system to help MW experts to debug the system. Explicit and informative error messages and proper logging of the error conditions are essential in a distributed asynchronous system, where error situations are usually difficult to reproduce. Every time we mention a possible problem in the system, we imply that a clear and explicit error message is delivered to the user, and that the problem is logged in the system logging and bookkeeping system for further retrieval and analysis.

One aspect we have not treated at all is the devious flow that can result from the concurrent execution of otherwise perfectly functioning use cases. What happen for instance if a user issues a *dsimport* use case and another user issues a *dsdelete* for the same DS at the same time? Given the number of combinations of this sort that can arise, we did not even try to classify them. We expect that the action taken by the system follow the general guidelines described above, trying to be as conservative as possible in preserving user data.

5.2 GENERAL USE CASES

These use cases concern the basic access to the Grid. We have the following cases:

1. Obtain Grid authorisation (*gridauth*);
2. Ask for revocation of Grid authorisation (*gridrevoc*);
3. Grid login (*gridlogin*);
4. Grid logout (gridlogout);
5. Browse Grid resources (*gridbrowse*);

5.3 DATA MANAGEMENT USE CASES

Data replication and management is one of the most important aspects of HEP computing Grids. Consequently there may be a lot going on behind the scenes in our data access use cases. Our primary interest as HEP users is that we get the best possible access. Hence the use cases state that we provide the logical name (LDN) of the dataset we want, and the system returns the information we need to access the data in it. The system accounts for any access-protocol constraints we may have specified.

In choosing the best access to the dataset, we expect the data management system to have considered the following cost options (respecting possible protocol constraint):

1. access (possibly via remote protocol) to each existing physical copy of the DS;
2. making a new replica to an SE and subsequently using that one;
3. making a local copy to temporary storage at the node where the job is running;
4. if a virtual definition of the dataset exists, materializing the DS to either a suitable SE or local temporary storage at the node where the job will run.

Regardless of which method the Grid chooses, the user accesses the DS by providing an LDN and passing the returned file identifier to an open call.

A physical copy of a dataset appears on a SE in four different ways:

1. Uploading it to the Grid for the first time, DS upload;
2. Copying it from another SE (DS replication);
3. Requesting a virtual dataset, which causes it to be produced on demand, using the algorithm known to the DMS and possibly other input datasets (Virtual DS declaration and materialization).
4. Importing directly from local storage (DS import). This can be useful when moving replica via removable devices, but other than that it can be a very error prone operation;

There is in principle a fifth choice, which is to write a dataset directly onto an SE. We consider this a special case of the "DS upload" use case. In this case the Grid provides a dataset staging area where files can be created via standard POSIX calls. This will be either a suitable area on the local machine or on the SE, or even a different area as long as it optimises the subsequent upload of the dataset to the SE.

We expect the data management system to track the access patterns for datasets. The system should take intelligent actions concerning replication (or materialization in the case of virtual datasets) to an advantageous SE when a pattern of frequent access emerges.

A special case is the transfer of a dataset to and from removable media. The export to removable store is a specialization of the copy of a physical instance of a dataset to non-Grid storage. The import from removable store is a different use case, as it implies the upload of a physical copy of an existing dataset to the Grid, and its addition to the dataset metadata catalogue of the DMS. In this case the DMS may simply trust that the user's uploaded replica is identical to all the other replicas, but more likely it will require passing some identity tests (e.g. checksums on the files) before the registration is accepted.

File-based use cases are specialisations of the data set use cases with the data set composed by a single file. The following are the data set management use cases including also the data set metadata management:

- DS metadata update (*dsmdupd*);
- DS metadata access (*dsmdacc*);
- Dataset registration to the Grid (*dsreg*);
- Virtual dataset declaration (*vdsdec*);
- Virtual dataset materialization (*vdsmat*);
- Dataset upload (*dsupload*);
- User-defined catalogue creation (*catcrea*);
- Data set access (*dsaccess*);
- Dataset transfer to non-Grid storage (*dsdownload*);
- Dataset replica upload to the Grid (*dsimport*);
- Data set access cost evaluation (*dsaccesscost*);
- Data set replication (*dsreplica*);
- Physical data set instance deletion (*dsinstdel*);
- Data set deletion (complete) (*dsdelete*);
- User defined catalogue deletion (complete) (*catdelete*);

- Data retrieval from remote Datasets (*remdsacc*);
- Data set verification (*dsverify*);
- Data set browsing (*dsbrowse*);
- Browse condition database (*cdbbrowse*);

5.4 JOB MANAGEMENT USE CASES

Users or Experiment production teams have logged into the Grid and wish to use the available resources. They do so by submitting *jobs* (see section 3.6) to the Grid Workload Management System (WMS). The job submission step involves at least specification of what program will be run, optional specification of the input and output datasets, and further optional specification of environment requirements (operating system, installed software) and expected resource consumption. The WMS is expected to make a number of choices, based on that user input, to optimise the usage of Grid resources so that the job is run in an optimal fashion. The user must, however, be able to override any automatic choice of the WMS.

We assume that each job has a *cost* based on the amount of resource used (CPU, memory, storage, bandwidth) and on the *priority* with which these resources are used. Furthermore we assume that the WMS is able to calculate the job cost on the basis on the information provided by the user in the JDL, and that it can use this information to optimise job execution. If some of the requests cannot be expressed easily in the JDL, the user can provide a *plugin* that returns a cost, and that it is supposed to be called by the WMS to evaluate the global cost of the job. This point should of course be discussed with the middleware developers.

In some use cases we speak of *job priority*. We realise that the level of detail here is not sufficient to guide implementers, however this is one of the many areas where more iterations will be needed to come to an understanding. The basic idea is that a user may want a shorter *time to solution* for a given job than the *default* one. This can translate in a higher placement of the job in the batch queue of the CE, so that it enters execution more rapidly, and a higher execution priority (sort of renice –20) on the WN, so that it gets a larger share of the WN CPU. Such a system will also need associated controls on priorities. For example, the system might deduct a larger amount from a user’s resource usage quota for a high-priority job than it would for the same job submitted with normal priority. Another possibility is to limit high-priority jobs to a special class of users. The system also will need to deal with priorities that differ from site to site.

In jobs dealing with input and output Datasets, their names may be specified to the WMS (to give it optimisation hints). In this case the WMS must pass these LDNs to the user program, to avoid specifying them separately.

Running jobs will create output of various kinds: informational or error messages about their progress (corresponding to *stdout* and *stderr* of Unix jobs), output generated locally on the execution host, written by *e.g.,* POSIX file calls, and output written to remote locations using *e.g.,* rfio, gridFTP, or a Grid file system.

By default, the SE for output datasets will be assigned dynamically. It must be possible to trace, after execution is completed, where a job has written a dataset, and it must be possible at least for the submitting user to access this file.

We assume the job current directory to be assigned dynamically. This local disk space is temporary and it will be reclaimed by the system. The files produced there must be accessible during job execution and they will be kept till the post-processing operations specified by the users are successfully completed, including returning the files produced in the job current directory. We expect the system to take appropriate action if this directory gets full. The amount of local space needed should be one of the parameters that can be specified in the JDL.

We expect the Grid to assign a unique job identifier to each Job (see also section 4.9). We also expect the Grid to store some summary information about each job, which can be retrieved using the job identifier as a key. The amount of information stored, its location and the expiration period should be customisable at the VO level. When a dataset is produced by a job, we expect the job identifier to be registered with the LDN. This would provide a tool to solve the problem of job-output tracing.

Furthermore we assume that the user will be able to store information in the job catalogue in three possible ways:

1. Attributes are passed by the user at submission time and stored by the Grid;
2. Attributes are set by the running job. This procedure is important because it allows tracking of job status via the job catalogue;
3. The user sets or modifies attributes of an existing (executing or completed) job.

Use cases that we have identified are:

1. Job catalogue update (*jobcatupd*);
2. Job catalogue query (*jobcatquery*);
3. Job submission (*jobsubmit*);
4. Job Output Access or Retrieval (*joboutaccess*);
5. Error Recovery for Aborted or Failing Production Jobs (*jobrecov*);
6. Job Control (*jobcont*);
7. Steer job submission (*jobsteer*);
8. Job resource estimation (*jobresest*);
9. Job environment modification (*jobenvspec*);
10. Job splitting (*jobsplit*);
11. Production job (Analysis);
12. Analysis (*analysis*);
13. Data set transformation (*dstran*);
14. Job monitoring (*jobmon*);
15. Simulation Job (*simjob*);
16. Experiment software development for the Grid (*softdevgrid*);

5.5 VO MANAGEMENT USE CASES

It is at the moment not clear what will be the privileges of the person in the experiments responsible for VO management. The possibilities range from a total control, including adding, removing and configuring sites to a very restricted set of rights. In the latter case the VO management will actually be done by a VO-independent entity, sometimes called Grid operation centre. The heart of the matter is the extent of the effects that modifications of the VO configuration have on the components of the Grid outside that VO.

In a global Grid approach, where different VO's share parts of the resources and infrastructure, some higher authority may have to control and coordinate part of the VO configuration to avoid security breaches and other operations that may affect the stability or integrity of the VO's.

In our discussions we have identified the following actions, which may evolve into use cases. The above discussion translates in a very small difference in the flow, as in one case the action is directly performed via Grid VO management tools, while in the other the request is submitted to the appropriate authorities.

1. Configuring the VO:
    a. Configuring the DS metadata catalogue (either initially or reconfiguring);
    b. Configuring the job catalogue (either initially or reconfiguring);
    c. Configuring the user profile (if this is possible at all on a per-VO basis);
    d. Adding or removing VO elements, e.g. computing elements, storage elements, DMS and WMS and the like, if possible on a per-VO basis. If not, this action might still be expressed by having a VO publish a table of "blessed" elements.
    e. Configuring VO elements and attributes, including quotas, privileges etc;
2. Managing the Users:
    a. Add and remove users to/from the VO;
    b. Modify the user information, including privileges, quotas, priorities and authorisations for the VO, either for single users or for subgroups of users within a VO.
3. VO wide resource reservation (*resrev*);
    a. The Grid should provide a tool to estimate the time-to-completion given as input an estimate of the resources needed by the job. This is needed in particular for the instantiation of a virtual Dataset, to estimate the access cost;
    b. There should be use cases for releasing reserved resources, and system manager use cases for what to do in case a user does not submit a job for which resources are reserved;
    c. A further open question is how a user associates reserved resources with a particular job. If a user submits a job that is supposed to use the reserved resources and one that does not, there must be a way to specify this to the Grid.
4. VO wide resource allocation to users (*userresmod*);
5. Condition publishing (*condpubl*);
6. Software publishing (*swpubl*);

6 QUANTIFICATION OF REQUIREMENTS

The quantitative requirements for LHC computing are still somewhat under discussion, even if very serious estimates exist, which are constantly updated. However the four experiments conduct periodic Data Challenges. We have decided to present, as an example of quantification of requirements, an example of a distributed MC production done in early 2003 and a set of tables with the numbers involved in the data challenges that all experiments foresee to hold in 2004.

6.1 EXAMPLE OF A RECENT EXPERIMENT DISTRIBUTED MC PRODUCTION

To give a feeling for target performance requirements we summarise below the recent results obtained by LHCb for distributed MC production with their DIRAC system. These figures are typical of current systems in the LHC experiments.

- 50 days running from 13 Feb 2003;
- 17 centres in production configuration (CERN, UK, FRANCE, Italy, Germany, Holland, Russia, Brazil);
- 36,600 jobs ran
    - 34,000 ran successfully 92% success (2% failed with LHCb software problems, 6% due to data transfer, site and farm problems);
- ~250,000 files where produced, each ~100 MB on average for a total of ~20 TB of data;
- Total CPU used 1.4 M hours 1GHz PC (equivalent to ~1000 CPUs for 50 days);

All the data produced was copied to CERN for storing on CASTOR.

6.2 PLANS FOR 2004 DATA CHALLENGES

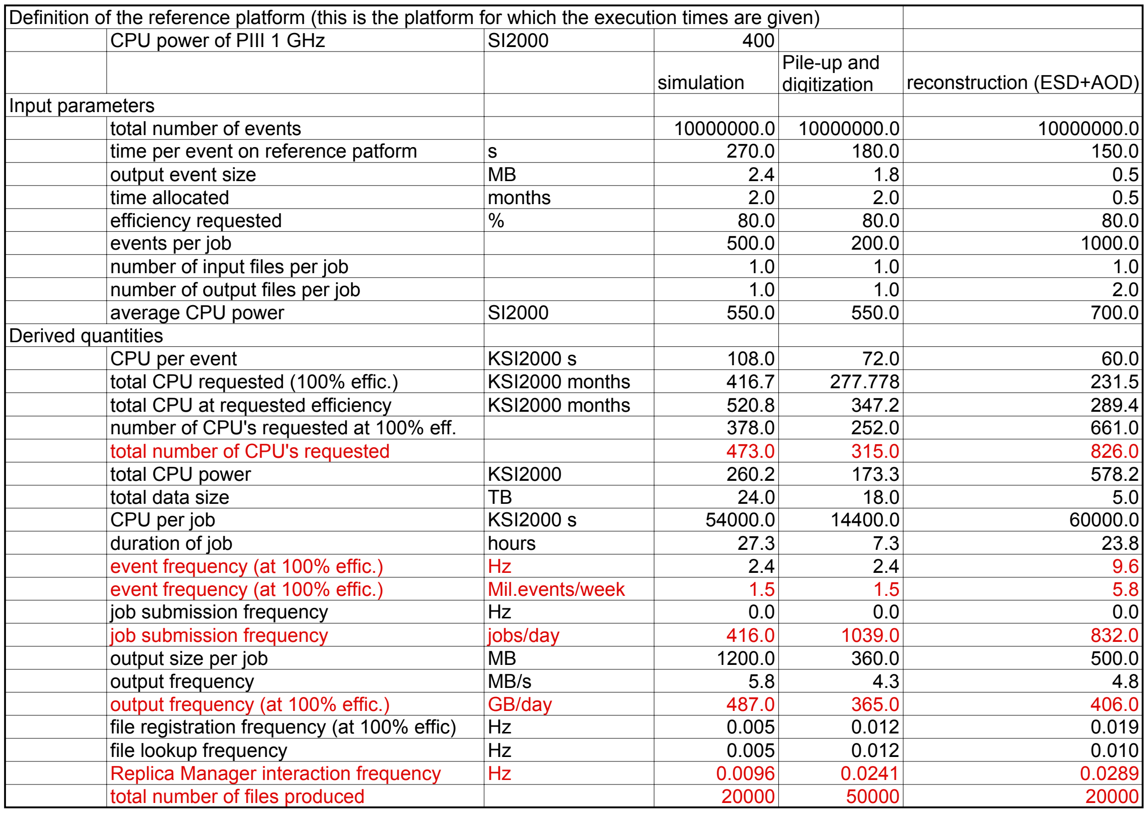
Here we present the plans for the 2004 data challenges for some of the LHC experiments.

Definition of the reference platform (this is the platform for which the execution times are given)					
CPU power of PIII 1 GHz	SI2000	400	400		
		digitization		reconstruction (ESD+AOD)	
Input parameters (they already contain a 20% development factor for cpu & 50% for disk)					
		Pb-Pb	p-p	Pb-Pb	p-p
total number of events		100000.0	10000000.0	5000000.0	10000000.0
time per event on reference platform	s	54000.0	324.0	2700.0	10.8
output event size	MB	900.0	1.2	10.5	0.0
time allocated	months	3.0	3.0	5.0	5.0
efficiency requested	%	95.0	95.0	95.0	95.0
events per job		1.0	1000.0	30.0	1000.0
number of input files per job		1.0	1.0	30.0	30.0
number of output files per job		1.0	1.0	1.0	1.0
average CPU power	SI2000	700.0	700.0	700.0	700.0
Derived quantities					
CPU per event	KSI2000 s	21600.0	129.6	1080.0	4.3
total CPU requested (100% effic.)	KSI2000 months	833.3	500.0	2083.3	16.7
total CPU at requested efficiency	KSI2000 months	877.2	526.3	2193.0	17.5
number of CPU's requested at 100% eff.		396.0	238.0	595.0	4.0
total number of CPU's requested		417.0	250.0	626.0	5.0
total CPU power	KSI2000	291.9	175.0	438.2	3.5
total data size	TB	90.0	12.0	52.5	0.3
CPU per job	KSI2000 s	21600.0	129600.0	32400.0	4320.0
duration of job	hours	8.6	51.4	12.9	1.7
event frequency (at 100% effic.)	Hz	0.014	1.350	0.406	0.810
event frequency (at 100% effic.)	Mil.events/week	0.008	0.817	0.245	0.490
job submission frequency	Hz	0.014	0.001	0.014	0.001
job submission frequency	jobs/day	1167.0	116.0	1168.0	70.0
output size per job	MB	900.0	1200.0	315.0	30.0
output frequency	MB/s	12.2	1.6	4.3	0.0
output frequency (at 100% effic.)	GB/day	1026.0	136.0	359.0	2.0
file registration frequency (at 100% effic)	Hz	0.014	0.001	0.014	0.001
file lookup frequency	Hz	0.014	0.001	0.406	0.024
Replica Manager interaction frequency	Hz	0.027	0.003	0.419	0.025
total number of files produced		100000.0	10000.0	166666.7	10000.0

Figure 10: Numbers for ALICE data challenge in 2004

Definition of the reference platform (this is the platform for which the execution times are given)					
CPU power of PIII 1 GHz	SI2000	400			
Input parameters					
Size of a generated event	MB	0.04			
Size of a simulated event	MB	0.9			
Size of a digitized event	MB	0.7			
Size of an ESD event	MB	0.15			
Size of an AOD event	MB	0			
			simulation	**digitization**	**reconstruction (ESD+AOD)**
total number of events			50000000	50000000	50000000
time per event on reference patform	s		400	60	30
time allocated	months		5	2	1
efficiency requested	%		75	75	75
events per job			250	1000	250
number of input files per job			1	4	3
number of output files per job			1	3	16
average CPU power	SI2000		550	550	700
Number of T1 in DC04					5
Fraction of digi's in each T1 in DC04	%				0
Fraction of ESD in each T1 in DC04	%				100
Fraction of AOD in each T1 in DC04	%				0
Derived quantities					
input event size	MB		0.04	0.90	0.70
output event size	MB		0.90	0.70	0.15
CPU per event	KSI2000 s		160	24	12
total CPU requested (100% efficiency)	KSI2000 months		3086	463	231
total CPU at requested efficiency	KSI2000 months		4115	617	309
number of CPU's requested at 100% efficiency			1122	420	330
total number of CPU's requested			1496	561	440
total CPU power	KSI2000		823	309	308
total data size	TB		45	35	7.5
CPU per job	KSI2000 s		40000	24000	3000
duration of job	hours		20.20	12.12	1.2
event frequency (at 100% efficiency)	Hz		5	13	26
event frequency (at 100% efficiency)	Mil.events/week		3.1	7.8	15.5
job submission frequency	Hz		0.021	0.013	0.103
job submission frequency	jobs/day		1777	1110	8870
output size per job	MB		225	700	38
output frequency	MB/s		5	9	4
output frequency (at 100% effic.)	GB/day		390	759	324
file registration frequency (at 100% efficiency)	Hz		0.021	0.039	1.643
file kookup frequency	Hz		0.021	0.051	0.308
Replica Manager interaction frequency	Hz		0.041	0.090	1.951
total number of files produced			200000	150000	3200000
Disk cache (I/O for 3 jobs), per CPU	MB		705	4800	637
Total disk cache (I/O for 3/CPU)	GB		1029	2629	273
Data transfer rate to each T1 for digi's	MB/s				0.00
Data transfer rate to each T1 for ESD	MB/s				3.85
Data transfer rate to each T1 for AOD	MB/s				0.00
Total data transfer rate for each T1	MB/s				3.9
Total data transfer rate from T0	MB/s				19
Data transfer rate to Castor at CERN	MB/s				22

Figure 11: Numbers for the CMS data challenge in 2004



Definition of the reference platform (this is the platform for which the execution times are given)					
	CPU power of PIII 1 GHz	SI2000	400		
			simulation	Pile-up and digitization	reconstruction (ESD+AOD)
Input parameters					
	total number of events		10000000.0	10000000.0	10000000.0
	time per event on reference patform	s	270.0	180.0	150.0
	output event size	MB	2.4	1.8	0.5
	time allocated	months	2.0	2.0	0.5
	efficiency requested	%	80.0	80.0	80.0
	events per job		500.0	200.0	1000.0
	number of input files per job		1.0	1.0	1.0
	number of output files per job		1.0	1.0	2.0
	average CPU power	SI2000	550.0	550.0	700.0
Derived quantities					
	CPU per event	KSI2000 s	108.0	72.0	60.0
	total CPU requested (100% effic.)	KSI2000 months	416.7	277.778	231.5
	total CPU at requested efficiency	KSI2000 months	520.8	347.2	289.4
	number of CPU's requested at 100% eff.		378.0	252.0	661.0
	total number of CPU's requested		473.0	315.0	826.0
	total CPU power	KSI2000	260.2	173.3	578.2
	total data size	TB	24.0	18.0	5.0
	CPU per job	KSI2000 s	54000.0	14400.0	60000.0
	duration of job	hours	27.3	7.3	23.8
	event frequency (at 100% effic.)	Hz	2.4	2.4	9.6
	event frequency (at 100% effic.)	Mil.events/week	1.5	1.5	5.8
	job submission frequency	Hz	0.0	0.0	0.0
	job submission frequency	jobs/day	416.0	1039.0	832.0
	output size per job	MB	1200.0	360.0	500.0
	output frequency	MB/s	5.8	4.3	4.8
	output frequency (at 100% effic.)	GB/day	487.0	365.0	406.0
	file registration frequency (at 100% effic)	Hz	0.005	0.012	0.019
	file lookup frequency	Hz	0.005	0.012	0.010
	Replica Manager interaction frequency	Hz	0.0096	0.0241	0.0289
	total number of files produced		20000	50000	20000

Figure 12: Numbers for the ATLAS data challenge in 2004

7 PRIORITIES OF THE USE CASES

Most of the information contained in this chapter comes from the analysis of HEPCAL done by the users, and in particular EDG ones. In the original HEPCAL document we decided not to give much attention to the prioritisation of the use cases because HEPCAL has been written as an "atomic" document, representing the *minimum* set of use cases that would allow running HEP job on the Grid. During discussions with the developers however, it has become clear that, even in this case, it would be useful to have a prioritisation, because those who develop the middleware would need to know on which item they should start to work first. We have therefore decided to introduce such a prioritisation, being clear that the HEPCAL still is an *atomic* document and for the purpose of providing a minimal complete Grid system for HEP, no use case is more important than the others.

In the original HEPCAL document there was already an attempt at a prioritisation in the single use cases. In the present version we have reorganised and rationalised the priority description with the following classification from 0 to 3. The meaning of this scale is reported in Table 2.

Priority	0	1	2	3
Meaning	Cannot even start testing the system without	Necessary for initial testing	Necessary for normal usage testing	Necessary for production or *acceptance* testing

Table 2: Priority scale description

For all use case we report the priority according to this scale.

Use Case	Priority	Use Case	Priority	Use Case	Priority
gridauth	0	*gridrevoc*	2	*gridlogin*	0
gridlogout	2	*gridbrowse*	1	*dsmdupd*	1
dsmdacc	0	*dsreg*	0	*vdsdec*	2
vdsmat	2	*dsupload*	0	*catcrea*	2
dsaccess	0	*dsdownload*	1	*dsimport*	2
dsaccesscost	3	*dsreplica*	1	*dsinstdel*	3
dsdelete	1	*catdelete*	3	*remdsacc*	2
dsverify	3	*dsbrowse*	1	*cdbbrowse*	2
jobcatupd	2	*jobcatquery*	1	*jobsubmit*	0
joboutaccess	1	*jobrecov*	2	*jobcont*	1
jobsteer	1	*jobresest*	3	*jobenvspec*	1
jobsplit	3	Analysis	2	*analysis*	3
dstran	1	*jobmon*	1	*condpubl*	2
swpubl	2	*resrev*	3	*userresmod*	3
simjob	1	*softdevgrid*	2		

During interaction with middleware developers, we came across the request to define *requirements* as well as use cases. While this is not the primary goal of HEPCAL, which has decided to concentrate mainly on use cases, LHC experiments have expressed a number of requirements that may be considered as a complement to the HEPCAL use cases. We report in a short summary of the most commonly expressed requirements.

Area	Requirement
Middleware robustness	• Need less than 2% job failures for internal WMS reasons (not including network, quota etc… problems)
Middleware reliability and scalability	• System must handle thousands of concurrent jobs, and be able to cope with job submission rates of one/few secs • Must be able to register millions of files • Must be able to handle multi-GB file transfers with global reliability of ~99%
Security	• Need a comprehensive grid-security implementation (multiple user groups with different levels of privilege) • Security procedures should permit outbound IP connectivity from WNs • Need to control file and metadata access right at the user and VO subgroup level (ACLs)
Accessibility	• Need programmable APIs(C++ and Java) to middleware services • Need 'recent' system support (Redhat 7.3, 8.0 and 9.0)
Data management	• Need grid access to files from a running application (gridopen etc..) • Need support for application metadata (as user defined fields associated to the LDN) • Must have a system for SE, CE, WN space management
Job handling	• Scheduling of jobs should take into account job input/output files location and trigger replication if needed. • WMS handle job sequences (e.g. DAGs)
Error reporting	• Uniform standards for middleware error reporting (making it easier for applications to intercept middleware errors and take the appropriate actions), e.g. reliable return status.
Mass Storage	• Mass Storage should be essentially invisible to Grid users, who should interact with storage only via the DRC.

8 CONCLUSIONS

We set out to identify a set of common use cases for how the four LHC experiments plan to use Grid technology in their computing infrastructure. Grid architectures that can implement such use cases will be useful to all four experiments.

The original designs for the various Grid projects were driven in part by requirements documents from experiments. These documents were constructed without prior experience in Grid computing, since there were no HEP Grid testbeds available at the time. Nearly a year later, all the LHC experiments have experience with the EDG testbed, as well as some national testbeds such as INFN-Grid, and some experiment-developed prototypes such as AliEn or D0-Grid, as well as with the LCG-x testbeds.

We started from a basic description of HEP computing tasks drawn from the complete lifecycle of an experiment. We succeeded in identifying common use cases for nearly all these areas, showing that the use cases were much more "common" across experiments than had been expected. Quite literally, the results exceeded our own expectations, as well as those of almost everyone with whom we discussed the project. This report provides a complete view of the common use cases we identified.

Given the limited duration of this RTAG, several important issues could not be completely covered:

1. The requirements and use cases for updateable, Grid-managed information are incomplete. Possibly catalogues are enough, but in this case our catalogue requirements do not sufficiently address the problems associated with network connectivity (in the case of a centralized catalogue), nor replication/synchronization (in the case of a distributed catalogue). If catalogues are not sufficient, use cases and requirements must be developed for read-write datasets, which we barely touch upon here.
2. The requirements and use cases for Grid-powered interactive work are incomplete. Our discussion did not even go as far as defining what "interactive on the Grid" meant.
3. We sketch some ideas about job splitting, but there is much left to do. This is a particularly important area for HEP computing. Given the inherent data parallelism in our field, a careful analysis of requirements, use cases, and architecture here could make life much easier for experiment framework developers.

These issues should be explored further. We believe the exploration should be an iterative procedure, including in some stages participation by middleware developers.

9 REFERENCES AND GLOSSARY

9.1 APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS

Applicable documents

R1	DataGrid-08-TEN-0201-1-11

9.2 TERMINOLOGY

Glossary

Analysis Object Data (AOD)	Event information containing the kinematics of the interaction reconstructed final state and additional information about the event. The exact content of the AOD is experiment dependent. AOD are the data with which most of the physical analysis is done with and are obtained from ESD.
Computing Element (CE)	Grid element that executes Grid jobs. It publishes information about its status to an information service.
Condition Data	Data that describe the condition of the detector at the moment of the acquisition of the events. These include the status of the detectors and their electronics, the exact position and configuration of the detectors and their calibration, i.e. the relation between their output and the physical quantity they are supposed to measure (position, energy, time etc).
DAG	Directed Acyclic Graph: a set of jobs with acyclic directed dependencies representing dependencies from one to another.
Data Management System (DMS)	Grid component that manages data. In particular the DMS is able to provide access to the data identified by a LDN or LFN (defined later).
DataSet (DS)	Container of information. This concept is extensively described in the text.
DS Metadata catalogue	A catalogue that contains user defined meta-information describing the content of each dataset.
Element	In this document the term element indicates the combination of a Grid service and the hardware on which this service is running on.
Event Summary Data (ESD)	Complete event information after reconstruction. ESDs are produced from RAW data and contain fairly complete information about the event. The exact content of ESD is experiment dependent.
Experiment framework	Software system, usually specific to a given HEP experiment, which provides services such as data access and storage, error handling, event visualisation and so on, to the different software modules used in the experiment.
Grid	Set of distributed heterogeneous resources and software (middleware) that allows transparent access to these resources.
Grid middleware	Grid software that allows to access distributed resources.
Information Service (IS)	Grid service that collects and makes available information about Grid elements and users.
Job	A single invocation of the Grid submission service. The concept is extensively described in 3.6.
Logical Data Set Name (LDN)	Location independent identifier of a data set managed by the Grid. A LDN is unique within a VO on a given Grid.
Logical File Name (LFN)	Location independent identifier of a file managed by the Grid. A LFN is unique within a VO on the Grid.

Metadata	Information about other data in the Grid, for example a list of physical copies of a particular dataset.
MonteCarlo	Simulation algorithm where an appropriate sequence of random numbers allows the prediction of a real process. MonteCarlo programs in HEP are characterised by an input (initial conditions and run parameters) that usually is much smaller than the produced output.
Physical File Name (PFN)	Character string including an access protocol, a machine (host) name, and a physical path, by which a particular file can be accessed.
Production	Data-transformation activity aimed at generating an official set of data. Typically this happens when transformation algorithms and calibrations have sufficiently matured. Productions will likely consist of many jobs. Production jobs can be tracked if they enter a *production identifier* field in the Job Catalogue.
Raw Data (RAW)	Data as produced by the detector data acquisition system and recorded during data taking.
Resource Specification Language (RSL)	Language used to request resources and Grid services to the Workload Management System (defined later).
ROOT	An Object Oriented framework for data analysis widely used in HEP (http://root.cern.ch)
Storage Element (SE)	Grid element that provides data access. It holds physical copies of datasets. It publishes information about its status and the datasets it stores to an information service.
Tag Collection (TAG)	Data associating events with summary information coming from the reconstruction process (typically <1kB) to allow event selection without reading the events themselves. A Tag may associate to each event a pointer with the LDNs where the event is stored and its position in it. Tags can be associated with either raw or reconstructed events.
User Catalogue (UC)	Catalogue containing Grid relevant information about each Grid user, such as privileges, accounting information, authorisation and authentication credentials.
Virtual Organisation (VO)	A particular distributed community of Grid users working for the same employer or project. VO users share common privileges and resources on the Grid.
Workload Management System (WMS)	Grid component responsible for the execution of a computing task on the Grid. WMSs are supposed to choose the resource to be used for the execution of a computing task in a way that optimises the use of the resources.

10 DESCRIPTION OF USE CASES

USE CASE: OBTAIN GRID AUTHORISATION

Identifier	*UC#gridauth*
Goals in Context	*Obtain authorisation to access the Grid*
Actors	*User, VO Access Authority Manager*
Triggers	*Need to access the Grid*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*The user has either a valid account on a computer connected to the Grid, or has access via the Web to a server that can execute Grid commands on her behalf;*
Post-conditions	*User can perform a Grid login as a member of a VO;*
Basic Flow	*1. User submits a request for authorisation to use the Grid (either via a web interface or a command line)* *2. The access authority manager confirms his authorisation as a member of a VO;* *3. User receives the instructions and any necessary physical token;* *4. Following the instructions the user properly configures his personal workspace;*
Devious Flow(s)	*1. N/A* *2. Authorisation is refused. A clear reason must be provided together with the instructions on the conditions to obtain authorisation.* *3. N/A* *4. User fails to configure her personal workspace. Support must be easily available to solve the configuration problems.*
Importance and Frequency	*Done when a Grid user wants to become member of a VO to have access to the Grid resources of that VO. In principle once per user and VO, but very high importance.*
Additional Requirements	
Example	

Here both the User and VO Access Authority Manager perform actions in synchronisation. To be more precise and show the interface between each Actor and the System (Grid) we should split this into:

- Generate Grid Authorisation request (Actor: User)
- Accept/Reject Authorisation request(s) (Actor: VO Access Authority Manager)
- Check Grid Authorisation request(s) (Actor: User)

However, even if we accept the fact that this is a “composite” use case, the user sees it as a single atomic action and any further specification would bring us too close to the implementation issues.

USE CASE: ASK FOR REVOCATION OF GRID AUTHORISATION

Identifier	*UC#gridrevoc*
Goals in Context	*Ask for revocation of the authorisation to access the Grid*
Actors	*Person authorised to ask for revocation for the user in question. This may include the user himself, the VO manager or perhaps other people such as an experiment software coordinator. This depends on the VO policy.*
Triggers	*User no longer needs to access the Grid*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*The user has either a valid account on a computer connected to the Grid, or has access via the Web to a server that can execute Grid commands on her behalf;* *User has authorisation to use the Grid;*
Post-conditions	*User cannot perform a Grid login;*
Basic Flow	*1. A request is submitted to revoke a user authorisation to use the Grid (either via a web interface or a command line);* *2. The access authority manager confirms the revocation;* *3. User possibly receives the instructions on how to complete the revocation procedure;* *4. Following the instructions the user completes the revocation procedure;*
Devious Flow(s)	*1. User fails to submit authorisation. The error message should explain why the operation failed and what to do next.* *2. Revocation is refused. The error message should contain a pointer to existing documentation on the Web on the authorisation procedures.* *3. Instructions are not clear or not available. In this case the user will probably contact the support centre. Instructions should be updated for the benefit of all users.* *4. Revocation procedure cannot be completed. User support should be readily available to solve the problem to avoid security hazards.*
Importance and Frequency	*Done when a Grid user no longer needs access to the Grid resources of a VO. In principle once per user and VO, but very high importance.*
Additional Requirements	
Example	*$ gridrevoc [-u user] [-r role] [-v VO]* *Rationale: A given user is revoked from a VO. The syntax suggests that also the ability of a user to play a given role in a VO can be revoked, which could be considered an extension point.*

USE CASE: GRID LOGIN

Identifier	*UC#gridlogin*
Goals in Context	*Initiate a Grid session*
Actors	*User*
Triggers	*Need to access the Grid*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*User has obtained Grid authorisation;* *The user has either a valid account on a computer connected to the Grid, or has access via the Web to a server that can execute Grid commands on her behalf;*
Post-conditions	*User can access Grid resources and services;*
Basic Flow	*1. User supplies her identification information via a middleware line command or web portal;* *a. **Extension point**: If the same Grid user can have different roles, specify role for the current session;* *b. **Extension point**: If the same Grid user belongs to more than one VO, specify the VO for the current session;* *2. System confirms successful Grid login;*
Devious Flow(s)	*1. Possible problems* *a. Middleware cannot launch the authentication procedure. The error message should point to the relevant information on the web and to the user support service in order to solve the problem.* *b. User is not allowed to assume the specified role. Operation is aborted.* *c. Specified role does not exist. Operation is aborted.* *d. User is not part of the specified VO. Operation is aborted.* *e. Specified VO does not exist. Operation is aborted.* *2. Login is refused. Possible corrective action should be indicated.*
Importance and Frequency	*High frequency and importance.*
Additional Requirements	*1. A job will not fail during execution due to expiration of Grid login;* *2. The Grid login does not expire during the user session in which it was executed (i.e. the user must log out or exit his web browser).*
Example	*$ gridlogin [-u user] [-r role] [-v VO]* *Rationale: The login procedure to the grid should be a simple command that can be executed from the command line. It should be possible to specify*

	different roles. We see at the moment no need for an API.

Note: We did not consider necessary to introduce a *change role* use case, as it could be built from a Grid logout use case followed by a Grid login with a different role.

USE CASE: GRID LOGOUT

Identifier	*UC#gridlogout*
Goals in Context	*Terminate a Grid session*
Actors	*User*
Triggers	*End of the need to access the Grid*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*User has obtained Grid authorisation;* *The user has either a valid account on a computer connected to the Grid, or has access via the Web to a server that can execute Grid commands on her behalf;*
Post-conditions	*User can no longer access Grid resources and services;*
Basic Flow	*1. User indicates that she wants to end the Grid session via a middleware line command or web portal;* *2. System confirms successful Grid logout;*
Devious Flow(s)	*1. Middleware cannot launch the logoff procedure. The error message should point to the relevant information on the web and to the user support service in order to solve the problem.* *2. Logoff is refused. Possible corrective action should be indicated.*
Importance and Frequency	*High frequency and importance.*
Additional Requirements	
Example	*$ gridlogout* *Rationale: The logout procedure to the grid should be a simple command that can be executed from the command line. We see at the moment no need for an API.*

USE CASE: BROWSE GRID RESOURCES

Identifier	*UC#gridbrowse*
Goals in Context	*Obtain list of Grid resources;*
Actors	*User*
Triggers	*Need to obtain list of Grid resources (VO active services, CEs, SEs, etc.)*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	
Post-conditions	*User obtains the requested information;*
Basic Flow	1. *User specifies the resources he wants to be informed about, including their type, their availability and other parameters of the query (e.g. show all the nodes with more than 3TB of scratch space available to my jobs);* 2. *User specifies the kind of report, textual listing and graphical presentation should be provide;* 3. *User submits the query via command, a Web portal or API;* 4. *System returns the result;*
Devious Flow(s)	1. *N/A* 2. *N/A* 3. *Middleware cannot execute the command. Operation is aborted..* 4. *Information is not entirely available to the user. The middleware should clearly explain whether this is because the user has no right to obtain this information or because the information is temporarily not available due to a failure of the information system. In no case non-available information should be "silently" skipped by default, but the command could provide an option to silently skip this information.*
Importance and Frequency	*High frequency and importance.*
Additional Requirements	*The resource availability should reflect the user's VO and privileges.* *The listing should be dynamic and reflect the current state of resource allocation. As an example, if one requests available computing power, the number of CPUs already in use should be accounted for.*
Example	*$ gridlist –r <resource> -u <user>* *$ gridlist –r <resource> -v VO* gridlist(struct *resource, struct *resource-list) *Rationale: Grid resources of a certain class are browsed via a CLI or an API. In an alternative form of this command, the query could be expressed by a SQL select statement. In the CLI version of the command we have given the possibility to specify a given user other than the one issuing the command or a*

	different VO. In the API version this information can be passed in the structure describing the resource.

This use case raises the issue of which user in which role is enabled to retrieve which information. This is an implementation issue, but a very important one for which our use case does not provide any hint. We realise this and we are aware that we need a proposal from middleware to react to, in order to make progress in this direction.

For how long the information must be available. Again it is difficult to answer in the absence of a precise description of the information that can be made available. More iterations are needed here.

USE CASE: DS METADATA UPDATE

Identifier	*UC#dsmdupd*
Goals in Context	*Modify a LDN entry in the DS metadata catalogue*
Actors	*User*
Triggers	*Modification of user-defined metadata of a data set;*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*User has the rights to perform the operation requested*
Post-conditions	*DS metatadata catalogue is updated*
Basic Flow	1. *User specifies* a. *the logical data set name (LDN);* b. *the list of* key=value *pairs that describe the metadata to be added or modified;* c. *Optionally specifies how to connect to the DS metadata catalogue;* 2. *The operation is executed on the DS metadata catalogue;* 3. *System confirms the operation;*
Devious Flow(s)	1. *N/A* 2. *Possible problems* a. *Operation cannot be executed because some of the keys are not supported. The command can fail completely (no key is updated) or partially (the supported keys are updated). The choice between these two failure modes should be either always the same, or else should be reported in the diagnostics produced by the failure. In either case the offending key(s) should be specified.* b. *User has no authorisation to modify keys. If different keys have different "access control lists", then see the above discussion on the handing of the error.* c. *The LDN is not in the catalogue. Operation is aborted.* 3. *The system cannot confirm the operation. In this case the operation should be aborted, rolling back the metadata changes, as the system must be left in a consistent, well-defined state after each operation.*
Importance and Frequency	*Used every time the user portion of the metadata information about a dataset is modified*
Additional Requirements	*At least three permission levels will be needed:* 1. *No permission to update attributes* 2. *Permission to update attributes but not add new ones*

	3. *Permission to add new attributes* *We have not been able to determine whether permission to update attribute values needs to be fine-grained (i.e. user can update some but not others);*

Example	$ dsmdupd –l LDN –k<key>=<value> -k<key>=<value>
	 dsmdup(char *LDN, struct *key);
	 Rationale: We imagine a list of key structures that contain the key name and the key value that are set by the call.

USE CASE: DS METADATA ACCESS

Identifier	*UC#dsmdacc*
Goals in Context	*Read metadata of a (virtual) data set*
Actors	*User; WMS;*
Triggers	*Search for data sets from meta-information (e.g. select all data sets that satisfy a given query);* *Get information about a dataset, including the information to materialize a virtual data set;*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*Actor has read access to the DS metadata catalogue*
Post-conditions	*Result of the user query is returned*
Basic Flow	*1. Actor specifies the query* *2. The result of the query is returned*
Devious Flow(s)	*1. N/A* *2. Possible problems* *a. User has no authorisation to see some of the fields. The fields not shown should be clearly indicated.* *b. The system is not capable of retrieving some of the information. This has to be clearly indicated. In no case non-available information should be silently skipped.*
Importance and Frequency	*High frequency and importance*
Additional Requirements	
Example	*$ dsmdacc LDN key* *dsmdacc(char *LDN, struct *key);* *Rationale: The call has a list of key structures that contain a key field and a value field that is filled by the call.*

USE CASE: DATASET REGISTRATION

Identifier	*UC#dsreg*
Goals in Context	*Register a new dataset to the Grid;*
Actors	*User, but the use case is only included by higher-level DS creation or upload use cases (i.e. a user never directly executes this use case);*
Triggers	*Creation of a new Grid Dataset;*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	
Post-conditions	*LDN of Dataset is registered in Data Management System;*
Basic Flow	1. *Actor specifies:* a. *The logical data set name (LDN);* b. *Optionally a default access protocol by which users will access the files;* c. *Optional metadata;* 2. *The LDN is registered on the Grid* 3. *A new entry (with key the current LDN) is created in the DS metadata catalogue* 4. *any metadata, if specified, is added to this entry in the DS metadata catalogue;* 5. *The system returns confirmation, along with the exact LDN assigned[*].*
Devious Flow(s)	1. *N/A* 2. *Possible problems* a. *LDN is already in use. The response to this case depends partially on the implementation (see discussion on LDN uniqueness). Ideally the user should be informed and the operation should be aborted.* b. *User has no privilege to create LDN. Operation is aborted.* c. *User has no privilege to specify some of the MetaData keys. The offending keys should be indicated in the error message and the operation is aborted.* 3. *The Middleware is not capable to perform the operation. Operation is aborted.* 4. *Same as above.* 5. *The system is unable to return confirmation. In this case the operation*

[*] In case the system alters LDNs to ensure uniqueness, this is a requirement.

	should be "rolled back", as the user must know in which state the system is.
Importance and Frequency	*Used every time a new data set is created*
Additional Requirements	
Example	$ dsreg –l LDN –p protocol –k <key>=<value> -k <key>=<value>
	dsreg(char* LDN, struct *key, protocol);

USE CASE: VIRTUAL DATASET DECLARATION

Identifier	*UC#vdsdec*
Goals in Context	*Creation of new virtual data set*
Actors	*User*
Triggers	*User wishes to define a new data set without immediately producing it*
Included Use Cases	*Registration of a new dataset to the Grid*
Specialised Use Cases	
Pre-conditions	*Necessary software has been registered on the Grid;* *Input LDN is registered;*
Post-conditions	*Virtual DS registered on Grid; corresponding materialization instructions registered in DS metadata catalogue*
Basic Flow	1. *User specifies:* a. *LDN of virtual dataset to be registered;* b. *Materialization instructions metadata, including* i. *Input LDNs (allowed to be virtual)* ii. *References to registered programs;* iii. *Estimated storage required for a material instance of this DS;* iv. *Estimated computing resources needed (e.g. bogomip-hours) to calculate this DS;* 2. *Dataset registration use case is executed;*
Devious Flow(s)	1. *The materialisation information is invalid. The reaction of the middleware depends on the extent on which the middleware is in charge of the materialisation, as opposed to delegation to the user. If the middleware can check at least some of the materialisation information, this should be done and a message returned to the user indicating the offending information.*
Importance and Frequency	*Every time a new virtual dataset is created;*
Additional Requirements	*Materialization information are stored in the DS metadata catalogue as pre-defined* key=value *pairs that are understood by the Grid component responsible for data set materialization.*
Example	*$ vdsreg –l LDN –p protocol –i file.ins –k <key>=<val> -k <key>=<val>* *vdsreg(char *LDN, char *inst_file, struct *key, protocol);* *Rationale: Instructions for generating a virtual dataset are contained in a file. For the API, we could also imagine a variable-size structure that describes the instructions in a set of stanzas.*

USE CASE: VIRTUAL DATASET MATERIALIZATION

Identifier	*UC#vdsmat*
Goals in Context	*Materialization of pre-declared virtual data set*
Actors	*User;*
Triggers	*User wishes to force materialization of a virtual DS, to a specific location;*
Included Use Cases	*Data Transformation job;*
Specialised Use Cases	
Pre-conditions	*Virtual DS has been declared to the Grid;* *User has the proper access rights for this dataset;*
Post-conditions	
Basic Flow	1. *User specifies LDN of virtual DS and (optionally) the location for output DS;* 2. *User optionally specifies whether the materialized product should be registered as a physical instance of this DS (the default depends on the specified output location).* 3. *System executes DS metadata access use case, retrieving materialization instructions;* 4. *System constructs a data transformation job description from the information with* a. *Input DS as specified;* b. *Registered programs;* c. *Output location as specified;* 5. *The system executes the data transformation job use case* a. *If output was specified as a new physical instance of the dataset, the instance is registered with the Grid as a new replica*
Devious Flow(s)	1. *LDN does not exist or output location is invalid, the action is aborted.* 2. *The output is specified as new physical instance but there is no local SE, or the SE indicated by the user cannot be used (insufficient space, unavailable, etc). Either the operation is aborted or the user allows the system to chose a close SE for registration.* 3. *Materialisation instructions no longer valid. Again if the middleware is in charge of at least some part of the materialisation, this information can be checked at this stage, if all goes via delegation to experiment plugins, this may be impossible. In general as much as possible should be checked at the earliest opportunity not to waste resources. Operation is aborted.* 4. *Job JDL should be parsed by middleware to verify existence of input DS and registered programs. If they are missing or unusable, a*

	message should be sent to the user before submitting the job. Similar remark on the output location. The user may not be allowed to write to output location. 5. *The job fails to materialise the DS. The operation is aborted and the system is "unrolled" and left in the same state it was before the job was submitted (apart from job log information of course);*
Importance and Frequency	*Every time materialisation is chosen as access method for a virtual dataset.*
Additional Requirements	*If the output location is a valid SE, the default is to register the material copy of the DS. If the output location is not a valid SE, the default is to not register the material copy. If no location is specified, the default is to create and register a copy on the closest SE. Specifying the hostname of an SE is in general enough, but for non-SE locations the user must specify a complete URI (e.g. gsiftp://hostname/dir/file).*
Example	*$ vdsmat –l LDN[–s (SE | filename)] [=r]* *vdsmat(char *LDN, char *outloc, Boolean register);* *Rationale: It is supposed that the command can check whether the output is a SE or a simple file based on the format of the text string*

USE CASE: DATASET UPLOAD

Identifier	*UC#dsupload*
Goals in Context	*Make a new data set available on the Grid*
Actors	*User*
Triggers	*Decision to have a dataset accessible by Grid services*
Included Use Cases	*Data set registration*
Specialised Use Cases	
Pre-conditions	*All the files of a new dataset are accessible for reading from the machine where this use case is being executed;*
Post-conditions	*Data set is stored on a SE and registered on Grid;*
Basic Flow	1. *User specifies:* a. *The list of files belonging to the data set;* b. *Information to register a dataset;* 2. ***Extension point:** the user specifies an SE, where the physical file should be placed;* 3. *A Physical instance of the data set files is uploaded to an SE;* 4. *Dataset registration use case is executed;* 5. *the physical instance is registered in the Grid as a replica of the target LDN;* 6. *The system confirms success and reports the LDN under which the file(s) are registered;.*
Devious Flow(s)	1. *N/A* 2. *N/A* 3. *Possible problems* a. *Some of the files belonging to the DS are not accessible. An error message is returned specifying the offending files and the operation is aborted;* b. *The user has no authorisation to upload a physical copy to the specified SE or the SE is not available. An option should be available to let the system continuing the operation on another SE or to abort the operation.* c. *The upload of the files fails. Operation is aborted.*
Importance and Frequency	*Used every time a new data set is created*
Additional Requirements	*The LDN should not be visible to the Grid until the physical file upload has successfully completed.* *Compound datasets cannot be uploaded. The component dataset have to be uploaded first and then the compound dataset is registered.*

Example	*$ dsupload* –l LDN –p protocol –k <key>=<value> -k <key>=<value> files dsupload(char* LDN, struct *key, char *[] files, protocol); Rationale: Here there is a problem in how to identify (refer to) the files belonging to the DS. This is mostly application dependent, in the sense that we can imagine a way to retrieve the list of files composing a dataset, but the application has to have a way to determine their meaning and content, and this cannot be provided by the Grid.

USE CASE: USER-DEFINED CATALOGUE CREATION

Identifier	*UC#catcrea*
Goals in Context	*Create a new user-defined ctalogue on the Grid*
Actors	*User*
Triggers	*Decision to have a user-defined catalogue accessible on the Grid;*
Included Use Cases	*Data set registration*
Specialised Use Cases	
Pre-conditions	
Post-conditions	*The catalogue is registered to the Grid and it is accessible on the Grid;*
Basic Flow	1. *User specifies:* a. *The name of the catalogue;* b. *The structure of the catalogue;* 2. ***Extension point:** the user specifies an SE, where the physical instance of the catalogue should be placed;* 3. *The creation of a new catalogue is requested to the Grid;* 4. *the system confirms success and reports the LDN under which the catalogue registered;*
Devious Flow(s)	1. *N/A* 2. *N/A* 3. *Possible problems* a. *The user has no right to create the catalogue. Operation is aborted.* b. *The structure of the catalogue is not supported. Operation is aborted.* c. *The SE is not available either due to a technical failure or because the given VO/user has no quota left, or because has no right to use that SE. The user should have the option to let the system continue the operation on another SE or to abort the operation.* 4. *The system fails to confirm the successful creation, then the operation is aborted.*
Importance and Frequency	*The role of user-defined catalogues in the experiments is not completely clear yet*
Additional Requirements	*"Structure of the catalogue" means the names and type of the keys, i.e. the database schema. There may be other aspects to "structure", we alluded to our uncertainty about user-defined catalogues under "importance and frequency".*
Example	$ catcrea –l CN –s catstruct.dat –s SE

	catcrea(char *CN, char *struc_file, char *SE); Rationale: we imagine that the structure is defined in a file. For the API either the same file is given as input or a structure can be created in memory describing the catalogue.

USE CASE: (VIRTUAL) DATASET ACCESS

Identifier	*UC#dsaccess*
Goals in Context	*Open data set for reading;*
Actors	*User;*
Triggers	*Need to access the data; Note that this use case can be triggered when the user application follows a reference to an object stored in a different Grid data set.*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*The requested dataset has been previously registered on the Grid;* *The user has read access to the data set;*
Post-conditions	*The user application is able to read the dataset either with POSIX reads or using the specified access protocol;*
Basic Flow	*1. User specifies:* *a. the logical data set name (LDN)* *b. optionally the file access protocol* *2. The Grid returns valid name(s) for the file(s) of the selected dataset;* *3. The user opens the files for reading with a POSIX open or using the syntax of the specified access protocol;*
Devious Flow(s)	*1. N/A* *2. Possible problems* *a. The specified protocol is not supported. The user should have the option to specify a fall=back protocol that the system could use.* *b. The LDN cannot be accessed by the user. (authorisation or non-existent LDN). Operation is aborted.* *c. There is a failure accessing the replica, It can be the SE down, a network problem or whatever. This should not be a fatal error, and in the case of multiple replicas of the requested DS, the system should automatically retry with the "next best" copy.* *3. The files are corrupted or unreadable. Barring any user error, the user should be given the possibility to force a "restaging" of the files (where appropriate) to try correcting the problem.*
Importance and Frequency	*Very high importance and frequency*
Additional Requirements	*As already said, accessing a compound dataset means to recursively access all the component datasets.* *When accessing a dataset that is composed of multiple files, the proper file must always be returned.*
Example	<u>General example with a protocol:</u>

	$ dsaccess –l LDN –f name –p protocol
	$ cat_prot name
	dsaccess(char *LDN, char *name, protocol);
	f_prot_open(name,”r”);
	<u>Example with the default POSIX protocol:</u>
	$ dsaccess –l LDN –f name [-p POSIX]
	$ cat name
	dsaccess(char *LDN, char *name, “POSIX”);
	FILE *f=fopen(name,”r”);
	Rationale: In case a LDN is composed by several physical files, <name> is the name of the directory where the files will be found. In case of a LDN composed by a single file, <name> is the name of the file to be opened via the specified protocol. Both the cardinality of the LDN and the names of the files that will appear in the <name> directory can be queried.

USE CASE: DATASET TRANSFER TO NON GRID STORAGE

Identifier	*UC#dsdownload*
Goals in Context	*Copy a data set to non-grid storage;*
Actors	*User;*
Triggers	*Need to have an unregistered copy of a data set*
Included Use Cases	*Data set access*
Specialised Use Cases	
Pre-conditions	*A data set is registered on the Grid;* *The user has read access to the data set;*
Post-conditions	*The data set files are copied to storage specified by the user*
Basic Flow	1. *User specifies:* a. *the logical data set name (LDN);* b. *output location;* 2. *Dataset access use case is executed with specified LDN, opening the DS for reading;* 3. *The data are copied to the specified location;*
Devious Flow(s)	1. *N/A* 2. *Possible problems beyond those of dsaccess* a. *The output location cannot be written. This can be due to the size of the data (with respect to the quota or the device size) or to the permissions. Operation is aborted* 3. *N/A*
Importance and Frequency	*Medium importance and frequency*
Additional Requirements	*For compound datasets, all the component datasets are transferred to non-grid storage.* *Non-grid storage in this context means only that the file being created will not be managed or tracked by the DMS. It may be a local (meaning on the machine on which the use case is executed) file, or it may be a remote file specified by a proper URI.*

Example	$ dsdownload –l LDN –f name
	dsdownload(char *LDN, char *name);
	Rationale: In case a LDN is composed by several physical files, <name> is the name of the directory where the files will be found. In case of a LDN composed by a single file, <name> is the name of the file. Both the cardinality of the LDN and the names of the files that will appear in the <name> directory can be queried.

This use case is very similar to *dsaccess* when the protocol is POSIX. We decided not to join the two cases because we do not want to suggest an implementation for *dsaccess*.

USE CASE: DATASET REPLICA UPLOAD TO THE GRID

Identifier	*UC#dsimport*
Goals in Context	*Upload a physical replica of a dataset;*
Actors	*User;*
Triggers	*User has a local copy of a dataset and wishes to place it on a local SE. Proposed use case is for poorly-connected user who receives e.g. the data on removeable media on a truck.*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*A data set is registered on the Grid;* *The user has the right to perform the replica upload operation;*
Post-conditions	*A new replica appears on a given SE for the specified LDN;*
Basic Flow	*1. User specifies:* *a. the logical data set name (LDN);* *b. the local files that compose the replica to be uploaded* *c. The name of the SE to which the DS should be uploaded;* *2. The system uploads the file(s) to the SE* *3. The system adds the new file(s) as a physical instance of the specified LDN updating the DS catalogue;*
Devious Flow(s)	*1. N/A* *2. Possible problems* *a. The LDN is not accessible, either because it does not exist or because the user has no right to access it. Operation is aborted.* *b. The local file cannot be read. Operation is aborted.* *c. The user has no right to use the given SE. The user should be given the option whether to abort the operation or let the system chose another SE.* *3. User has no right to update the DMC. Operation is aborted.*
Importance and Frequency	*Low importance and frequency*
Additional Requirements	*This use case has been introduced to cover the case of replica transfer on removable media. VOs may place additional requirements on this use case, such as enforcing identity checks (does the new file have the same checksum as an existing registered copy?) or the addition of information in the DS catalogue that this is a "private" replica.* *For compound datasets, this use case is similar to dataset upload: first all the component datasets have to be uploaded to the SE, then the compound is declared to be present on that SE.*

Example	*$ dsimport* –l LDN files –se SE_name dsimport(char* LDN, char *[] files, char* SE_name);

USE CASE: DATASET ACCESS COST EVALUATION

Identifier	*UC#dsaccesscost*
Goals in Context	*Estimation of the cost for data access*
Actors	*User;*
Triggers	*Need to know the Grid evaluation of the cost to access a physical copy of a dataset on a specific SE (including the cost of putting the copy there if there isn't one already);*

Included Use Cases	*DMS query;*
	DS metadata catalogue access;

Specialised Use Cases	
Pre-conditions	*The (virtual) data set is registered on Grid;*
Post-conditions	*The cost of access for a dataset is known;*
Basic Flow	1. *User specifies access parameters:* a. *LDN;* b. *Access protocol;* c. *The data "source": SE hosting a physical copy of the specified LDN; should the data not be there, the Grid will include in the result of the query the minimum cost to put the data there;* d. *The data "sink": CE from which a program will attempt to access the dataset;* e. *In case the data set is virtual, whether the dataset should be copied or regenerated according to the materialisation instructions;* 2. *The total cost and its components (including effects of bandwidth if available) is returned for the specified parameters;*
Devious Flow(s)	1. *Same problems that can arise for dsaccess for LDN and protocol and in addition:* a. *The user has no right to run on the specified SE. Operation is aborted.* 2. *Possible problems* a. *The user has no right to ask for the cost of the specified combination of elements. Operation is aborted.* b. *The analysis performed by the system is impaired by some problem of the Grid. Again it should be clearly reported that the analysis reported is based on sub-optimal information.*
Importance and Frequency	*Important for production planners or Grid debugging, unknown frequency*
Additional	*Grid should be able to provide cost information*

Requirements	*Wildcards or lists may be used for the different elements of the query, in this case the system should return the evaluation for each permutation of the specified parameters;*
Example	*$ dsaccesscost –l LDN –p protocol –i SE –o SE [-c | -r]* *int dsaccesscost(char *LDN, protocol, char *inSE, char *outSE, char *opt);* *Rationale: A normalised access cost is returned. If access cost is needed for each component, either a structure can be returned with more information or this function can be called several times con different options*

Note: Interaction with middleware experts made us aware of the many issues left open in this use cases, and namely

1. How is the "cost" defined? What are its component and units of measure?
2. Does the cost depend on the access pattern? If this the case, how can a user specify an access pattern?

The only remark we can make at this point is that we have the need for such a service, but we do not know how to define it better. We should receive a proposal from MiddleWare developers.

USE CASE: DATA SET REPLICATION

Identifier	*UC#dsreplica*
Goals in Context	*Copy a data set to another Grid location*
Actors	*User, Grid*
Triggers	*User or Grid decision*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*DMS query;* *A data set is registered on the DMS and a physical copy exists on a SE*
Post-conditions	*Data set is physically accessible from the specified SE, and the physical copy is added to the DS catalogue.*
Basic Flow	*1. Actor specifies:* *a. The logical data set name (LDN)* *b. The output SE* *2. The system executes the dataset access use case for the specified LDN* *3. The system copies the data to the output SE* *4. The system updates the DS catalogue with the new physical location*
Devious Flow(s)	*1. N/A* *2. All failures possible for the dsaccess use case* *3. Possible problems* *a. The user does not have the right to materialise the DS on the specified SE. Operation is aborted.* *b. There is not enough space, either physical or quota to materialise the DS on that SE. Operation is aborted.* *4. The user does not have the right to update the DMC. Operation is rolled back.*
Importance and Frequency	*Used every time a new data set is copied*
Additional Requirements	*When replicating a compound dataset to an SE, the DMS must replicate all the component datasets to that SE and registered their presence there. What happens if some of the components are already present on that SE is an implementation issue.*

Example	`$ dsreplica -l LDN -s SE`
	`dsreplica(char *LDN,char *SE);`

USE CASE: PHYSICAL DATA SET INSTANCE DELETION

Identifier	*UC#dsinstdel*
Goals in Context	*Delete a physical instance of a data set from the DMS*
Actors	*User; Local SE manager*
Triggers	*Free space on a SE*
Included Use Cases	*DMS query;*
Specialised Use Cases	
Pre-conditions	*A data set is registered on the DS* *User has deletion rights on the data set*
Post-conditions	*Entry for the specified instance of the dataset is removed from the DS catalogue and the space on the corresponding SE is marked as free*
Basic Flow	*1. User specifies* *a. the logical data set name (LDN)* *b. the SE on which the data set copy is present* *2. The system verifies that the instance can be removed (is it the last physical instance, and no virtual definition is available?) If it cannot be removed, the system returns an error status* *3. The SE entry for the LDN is removed from the DS catalogue* *4. The data set files on the SE are marked deleted*
Devious Flow(s)	*1. N/A* *2. Possible problems* *a. The user has no right to remove the specified copy, either because she has no right on the LDN or because she has no right to remove a copy at the specified SE. Operation is aborted.* *b. The LDN is invalid. Operation is aborted.* *c. The targeted copy is the last one and the DS is not virtual. Operation is aborted.* *d. The target copy is the last one and the DS is virtual. A warning is issued and the user is given the choice whether to go ahead or not.* *3. N/A* *4. N/A*
Importance and Frequency	*Medium importance and frequency*
Additional Requirements	*When deleting a physical instance of a compound dataset from an SE, the component datasets are not deleted from the SE and remain registered to the DMS unless explicitly deleted. Deleting the physical instance of a dataset that is a component of a compound dataset present on the same SE is forbidden. The physical copy of the compound dataset has to be deleted*

	first.
Example	$ dsinstdel –l LDN –s SE dsinstdel(char *LDN, char *SE);

USE CASE: DATA SET DELETION

Identifier	*UC#dsdelete*
Goals in Context	*Delete a data set from the DMS and DS metadata catalogues*
Actors	*User*
Triggers	*A data set is no longer needed on the Grid*

Included Use Cases	*DMS query;*
	Physical data set instance deletion;
	Update of DS metadata catalogue;

Specialised Use Cases	
Pre-conditions	*The data set is registered on the DMS* *User has deletion rights on the data set* *User has the rights to update the DS metadata catalogue*
Post-conditions	*The target dataset is removed from the Grid;*
Basic Flow	*1. User specifies* *a. The logical data set name (LDN)* *b. Whether the delete should be "hard", that is any dependencies should be ignored. The default is a soft delete.* *2. If the delete is not hard, the system checks for dependencies; for example are there any virtual DS defined which require the current DS as input? If there are dependencies, the system returns an error and list of dependencies that would have broken.* *3. A DMS query is executed, retrieving the list of the physical instances of the data set* *4. For each of the instances, the physical dataset instance deletion use case is executed;* *5. The LDN entry in the DS metadata catalogue is removed*
Devious Flow(s)	*1. N/A* *2. Possible problems* *a. The user has no right to perform a hard delete; it is undesirable that any user will be able to do a hard delete on any DS in her VO. Operation is aborted.* *b. The delete is soft and there are dependencies. Operation is aborted.* *3. The DMS query fails due to problems in the DMS Operation is aborted.* *4. The user does not have the right to remove a given PDS on an SE. Operation continues, but the user is warned and the given PDS is marked as an "orphan" for further cleanup.*

	5. N/A
Importance and Frequency	*Medium importance and frequency*
Additional Requirements	*Deleting a logical compound dataset doesn't remove any of the component logical and physical datasets. Deleting a logical dataset that is component of a compound dataset is forbidden.*
Example	$ dsdelete [-h] –l LDN dsdelete(char *LDN, bool hard);

USE CASE: CATALOGUE DELETION

Identifier	*UC#catdelete*
Goals in Context	*Delete a catalogue from the Grid*
Actors	*User*
Triggers	*A catalogue is no longer needed on the Grid*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*The catalogue is registered on the Grid* *User has deletion rights on the catalogue*
Post-conditions	*The target catalogue is removed from the Grid;*
Basic Flow	*1. User specifies* *a. The catalogue name on the Grid;* *2. The system returns a message with the result of the operation;*
Devious Flow(s)	*1. Possible problems* *a. Catalogue does not exist. Operation aborted.* *b. User has no right to delete the specified catalogue. Operation aborted.* *c. If the system can detect whether the catalogue is in use, this would prevent the deletion of the catalogue.* *2. The system cannot confirm the successful completion of the operation. Operation is aborted.*
Importance and Frequency	*Medium importance and frequency*
Additional Requirements	
Example	$ catdelete –c CAT catdelete(char* CAT);

USE CASE: DATA RETRIEVAL FROM REMOTE DATASET

Identifier	*UC#remdsacc*
Goals in Context	*Access remotely portions of a DS*
Actors	*User*
Triggers	*Datasets contains one or more events. Some jobs will read only a few events per file. If the fraction of data is small enough, jobs could execute more quickly if they could access single events rather than accessing entire files, making local replicas. If the middleware can provide this service, it may need a hint to indicate that a given DS may be opened remotely.*
Included Use Cases	*DS access;*
Specialised Use Cases	
Pre-conditions	*Availability of remote access protocol;*
Post-conditions	*Desired portion of the DS read in memory;*
Basic Flow	*1. User specifies a DS to be accessed and* *a. A hint that the DS can be accessed remotely;* *b. The protocol to access this DS remotely;* *2. The system (DMS or WMS) chooses the SE to be accessed remotely. The chosen SE must support the remote access protocol specified.* *3. The User opens the DS according to the specified protocol;*
Devious Flow(s)	*1. Protocol is unknown. The operation is aborted. An option could be offered to default to another remote accessing protocol, if such exists.* *2. No remote SE is available that support the specified protocol. See above point for reaction from the system.* *3. The remote connection fails. The Grid should try to re-establish the connection.*
Importance and Frequency	*Medium importance, low frequency.*
Additional Information	*The trade-off is between complexities of programming (providing single-event access) vs. typical event sizes and typical bandwidth. Given sufficient bandwidth and not-too-large single events, it might not be worth the trouble;* *This use case is no different than DS access. We have decided to keep it to underline the specificity of this kind of access.* *In principle the safest way of remotely accessing a compound dataset is to access it from an SE where it is present (i.e. ALL of it). If a replica of a component dataset is "closer" to the WN than the compound dataset, this replica can be accessed. This is essentially just an implementation issue.*

Example	`$ remdsacc –l LDN –f name –p protocol`
	`$ cat_rem name`

	remdsacc(char *LDN, char *name, protocol);
	f_rprot_open(name,”r”);

USE CASE: DATA SET VERIFICATION

Identifier	*UC#dsverify*
Goals in Context	*Verify that a data set respects the data quality criteria*
Actors	*User* *Production manager*
Triggers	*Need to validate the data;* *Can be started automatically;*
Included Use Cases	*DS transformation job*
Specialised Use Cases	
Pre-conditions	*Availability of validation software;*
Post-conditions	*DS metadata updated with validation result;*
Basic Flow	*1. User specifies validation job information including:* *a. DS to be validated* *b. Validation program* *c. If needed reference DS* *d. Metadata catalogue DSN* *2. User submits validation job executing the dstran use case;* *3. DS to be validated and reference data files are accessed;* *4. Validation program is run;* *5. Metadata catalogue is updated;*
Devious Flow(s)	*1. N/A* *2. Common problem for job submit use case* *3. Same problems as dsaccess use case.* *4. Validation procedure is incorrect. Probably the program just fails. The system design should be such that "failure of the validation program" can be distinguished from "dataset failed the validation procedure".* *5. User has no right to update DMC. Operation is aborted.*
Importance and Frequency	*Can be run at the end of each production job. Can be used by any user;*
Additional Information	*Validation jobs are expected to be associated to data transformation jobs via job dependency (DAG);*
Example	$ dsverify –l LDN –p program [-d DS] [-m DSN] int dsverify (char *LDN, char* program, char* DS, char *DSN);

USE CASE: DATASET BROWSING

Identifier	*UC#dsbrowse*
Goals in Context	*Browse the LDNs*
Actors	*User*
Triggers	*Need to consult the DS list*
Included Use Cases	*Grid login*
Specialised Use Cases	
Pre-conditions	*User has a valid Grid login. A VO DMS is accessible by the user and contains the files to be browsed*
Post-conditions	
Basic Flow	*The user connects to her VO DMS, via Web or command line interface* *The user browses the available DS.*
Devious Flow(s)	1. *User has no right to browse the DMS database. Operation is aborted.* 2. *DMS database is not accessible. Operation is aborted.*
Importance and Frequency	*As important and probably frequently used as the* ls *command.*
Additional Requirements	

Example	$ dsbrowse [parameters to be defined such as date created etc] <SQL query>
	int dsbrowse(char* SQL_query, char* option, char*[] LDNs);
	
	Call returns the number of LDNs that satisfy the search options.

USE CASE: BROWSE EXPERIMENT CATALOGUE

Identifier	*UC#cdbbrowse*
Goals in Context	*Browse the content of an experiment catalogue*
Actors	*User*
Triggers	*Need to know the content of the experiment catalogue*

Included Use Cases	*Grid login;*
	DMS query;

Specialised Use Cases	
Pre-conditions	*A DMS is available;* *A database DS is registered on the DMS;*
Post-conditions	*Content of experiment catalogue is known to the user;*
Basic Flow	*1. User specifies* *a. LN of catalogue to browse* *b. Query to be submitted* *2. Query is submitted to the Grid* *3. Result is returned*
Devious Flow(s)	*1. N/A* *2. Possible problems* *a. DB (catalogue) is not accessible, either because it does not exist or because the user has no righ to access it. Operation is aborted.* *3. Results cannot be returned to the user.*
Importance and Frequency	*Medium importance and frequency*
Additional Requirements	*This is similar to dataset access, replacing the actor "user" by actor "experiment catalogue browsing program". We decided to keep this and other similar ones, to illustrate how we see things working.* *The question about replicating the entire experiment catalogue to a single machine, in the case of a user wanting to submit a job to her laptop and then take it on a plane, is still open as databases are not trivially isomorphic to files. We probably need to introduce the concept of "catalogue snapshot" to local storage. As we have delegated to the GRID the choice of the technology, the snapshot is really an "export" to a given DB, Oracle, postgres, mysql, XML …* *The whole question of experiment catalogues is still very undefined, so we need a trial implementation to make progress here. For this very reason we do not provide an example.*
Example	

USE CASE: JOB CATALOGUE UPDATE

Identifier	*UC#jobcatupd*
Goals in Context	*Modify the user fields of a job entry in the job catalogue;*
Actors	*User or job;*
Triggers	*Modification of user-defined attributes of a job;*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*User has the rights to perform the operation requested*
Post-conditions	*Job catalogue is updated;*
Basic Flow	1. *User specifies:* a. *Job identifier;* b. *the list of* key=value *pairs that describe the attributes to be added or modified;* c. *in case the job specified is composite, optionally indicate to apply recursively the update to all children;* 2. *The operation is executed on the Job catalogue: the given values are assigned to the keys, for all the jobs indicated in the request;* 3. *System confirms the operation;*
Devious Flow(s)	1. *N/A* 2. *Possible problems* a. *Job does not exist. Operation is aborted.* b. *Invalid key(s). Operation can abort completely or only the keys that are correctly specified can be updated.* c. *User has not the right to update the given job catalogue. Operation is aborted.* d. *User should have the possibility to specify that the given job should not be running. In this case if the job is running, operation is aborted.* 3. *System cannot confirm operation. Operation is aborted.*
Importance and Frequency	*Used every time the user portion of the job information is modified;*
Additional Requirements	*An authorised user may want to delete some entries from the job catalogue. How this is done is an implementation issue, but as a result of this action the corresponding jobs should not be returned by subsequent queries. At least three permission levels will be needed:* 1. *No permission to update attributes* 2. *Permission to update attributes but not add new ones* 3. *Permission to add new attributes*

	It is not known whether permission to update attribute values needs to be fine-grained (i.e. user can update some but not others);
Example	$ jobcatupd –j jobid –k<key>=<value> -k<key>=<value> [-r] jobcatupd (char *jobid, struct *key, const char *options); Rationale: We imagine a list of key structures that contain the key name and the key value that are set by the call.

USE CASE: JOB CATALOGUE QUERY

Identifier	*UC#jobcatquery*
Goals in Context	*Query the job catalogue to retrieve job identifiers matching query parameters;*
Actors	*User;*
Triggers	*Query the job catalogue (e.g. select all jobs that satisfy a given query;*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*Actor has read access to the job catalogue*
Post-conditions	*Result of the user query is returned*
Basic Flow	1. *Actor specifies the query* 2. *The result of the query is returned, a list of job identifiers;*
Devious Flow(s)	1. *N/A* 2. *Possible problems* a. *Malformed query. Operation is aborted.* b. *User has no right to perform the query. Operation aborted.* c. *Some of the information is not accessible (for instance for running jobs, or not-yet-run jobs, in case jobid can be pre-allocated). A warning is issued and existing information is returned. Unavailable information should be reported as such, unless one specifies the silent flag*
Importance and Frequency	*High frequency and importance*
Additional Requirements	

Example	$ jobcatquery key=<value> key=<value> ... [-s]
	int jobcatquery(struct *key, char *[] jobids, boolean silent);
	Rationale: The call has a list of key structures that contain a key field and a value field that is filled by the call. A more classic SQL call can be imagined. The integer value returned is the number of jobids matching.

USE CASE: JOB SUBMISSION

Identifier	*UC#jobsubmit*
Goals in Context	*Send Job to Grid Computing Resources*
Actors	*User*
Triggers	*Decision to submit job*
Included Use Cases	*Specify program; Dataset Access;*
Specialised Use Cases	
Pre-conditions	*User or Program logged into Grid;* *Needed Datasets available on network;*
Post-conditions	*Program is run. Any files specified as "valuable output" are available for further use or retrieval;*
Basic Flow	*1. User specifies job information:* *a) Environment needed (hardware and software, can be any);* *b) Any Grid input dataset needed;* *c) Any local input files needed;* *d) The program to be executed;* *e) Any output files which should not be deleted;* *f) Optionally job attributes in the form of* key=value *pairs to be set in the job catalogue;* *2. EXTENSION POINTS: (steer submission), (resource estimation), (environment modification);* *3. User submits description to job submission command;* *4. The job catalogue is updated* *5. Job executes;* *6. Upon completion, system optionally notifies user;*
Devious Flow(s)	*1. N/A* *2. Depending on the number of inputs given, many things can fail, here is a non exhaustive list* *a. Environment specified not available. Either job stays in the queue till it becomes available or it aborts directly. User is notified in all cases* *b. Specified Grid DS not available. Again job can stay in the queue waiting for those to become available or abort. User is notified.* *c. Local input files not available. Operation is aborted.* *d. Program to be executed not found. Operation aborted.* *e. Invalid keys in the job attributes. Those keys that are valid are*

	stored. A warning message is issued for the offending keys. 3. *User has no right to submit job with the specified parameters. In particular user has no right to request the specified resources. Job stays in the queue and user is notified.* 4. *Job catalogue cannot be updated. Operation aborted.* 5. *Job crashes during execution. If the crash is independent from the user program, job is resubmitted. User is notified.* 6. *System cannot notify the user. An entry with this information is entered in the job log.*
Importance and Frequency	*Basic job. High frequency and importance.*
Additional Requirements	*When the job is submitted, there may be a list of LDN's in the JDL. These LDN's will be materialised in PDN's, probably accessible on local storage. We should have a mapping: {LDN}=>{actual name of the local file(s) to be opened} so that a program could open files using a standard naming.* *To optimise resource usage and response time, the RB should use the resource estimate given.*
Example	*There are many examples of batch job submission from leading Resource Management Systems, so it would be useless to add one more. We would like to have also an API to submit a job. An API for job submission and one for DS browsing would allow a user process to drive the job splitting dynamically.*

USE CASE: JOB OUTPUT ACCESS OR RETRIEVAL

Identifier	*UC#joboutaccess*
Goals in Context	*Retrieve output of a running job*
Actors	*User*
Triggers	*Need to monitor job progress*
Included Use Cases	*Job submission; Grid login*
Specialised Use Cases	
Pre-conditions	*User logged into Grid;* *User knows the identifier of a job running;*
Post-conditions	*Information on the files produced in the job space local to the execution site is retrieved. Selected files are retrieved or browsed.*
Basic Flow	1. *User specifies job identifier;* 2. *User submits a query to list the content of the job local storage* 3. *System returns a list of files in the job local storage;* 4. *User submits a query to retrieve one or more of these files;* 5. *System returns the files on local user storage (in this case meaning on the computer from which the user executes this use case);*
Devious Flow(s)	1. *N/A* 2. *Possible problems* a. *Specified job does not exist. Operation aborted.* b. *User has no right to access information about the job. Operation aborted.* 3. *Possible problems* a. *User has no access to some of the files. This has to be communicated to the user.* 4. *User's query specifies one or more files that don't exist in the job space (typo!).* 5. *Local storage insufficient, either physical or quota. Operation is aborted.*
Importance and Frequency	*High frequency and importance.*
Additional Requirements	
Example	$ joboutaccess –j jobid [–l] [–c file] [–p] [-o file] Rationale; different functions can be performed with different options, Examples could be **l** list the files in the sandbox

	c	copy the file to output location
	p	cat the content of the file to standard output or to the location specified by **-o**
	o	output file

USE CASE: ERROR RECOVERY FOR FAILED PRODUCTION JOBS

Identifier	*UC#jobrecov*
Goals in Context	*Stop a production that is known to fail*
Actors	*User, Production manager*
Triggers	*Failure of a job in a production for reasons that will lead the whole production to fail.*
Included Use Cases	*Job submission*
Specialised Use Cases	
Pre-conditions	*A production job has been submitted according to a corresponding Use Case and one of the subjobs fails.*
Post-conditions	
Basic Flow	1. *The middleware sends information about crashed or aborted jobs along with diagnostic information.* 2. *The production manager can take action the following actions:* a. *Delete the entire production job;* b. *Delete the subset of this job that has been submitted to the faulty site* c. *Migrate the subset submitted to the faulty site, resubmitting at a different site* 3. *The system releases the resources freed (if any) by the preceding action;*
Devious Flow(s)	1. *N/A* 2. *Possible problems* a. *Specified jobid is invalid. Operation is aborted.* b. *User has no right to perform the operation. Operation is aborted.* 3. *N/A*
Importance and Frequency	
Additional Requirements	*Points b) and c) require to associate an Id with a list of jobs, e.g. those submitted to one site, to perform global operations. Given this Id, the job management is implemented by other use cases.* *It must be possible to associate an error report with a (high-level) production job, i.e. with the set of subjobs.*
Example	$ jobrecov –i jobid [–a] Rationale: This command graciously stops *jobid*. If the –a option is specified, all the production to which *jobid* belongs is stopped. There is no API for this call.

USE CASE: JOB CONTROL

Identifier	*UC#jobcont*
Goals in Context	*Perform management or control functions on a job*
Actors	*User, production manager;*
Triggers	*Need to change the current status of a job*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*A job has been submitted;* *User has enough privileges to perform specified actions;*
Post-conditions	*Specified action is performed;*
Basic Flow	1. *The user specifies the action and an identifier for a job or a set of jobs;* 2. *The user submits the request to the system;* 3. *The system performs the action;* 4. *The system returns a message with the result of the action and any additional information;*
Devious Flow(s)	1. *N/A* 2. *Possible problems* a. *Invalid Jobid. Operation aborted.* b. *Unsupported action. Operation aborted.* c. *Insufficient rights. Operation aborted.* 3. *The action requested could not be performed due to a temporary problem in the elements involved. The user is notified and she is given the opportunity to schedule the action at the earliest opportunity or to abort it.* 4. *N/A*
Importance and Frequency	*High importance. High frequency, people make lots of mistakes.*
Additional Requirements	*The following actions must be possible for jobs in queue* (i.e. *the job has been submitted to the resource broker but is not yet executing);* 1. *Cancel the job;* 2. *Change the job's priority;* 3. *Reroute the job to a different queue compatible with its parameters;* 4. *Hold a job in the queue;* 5. *Resume a job that has been held;* *The following actions are desirable for jobs in queues:*

	1. Reroute the job to a specific Computing Element; *2. Change (some of) the parameters with which it has been submitted;* *The following actions must be possible for a running job:* *1. Kill the job retrieving the local files;* *2. Kill the job without retrieving the local files;* *3. Resubmit the job (if the job is running this may mean killing the current running instance);* *4. Suspend a job;* *5. Resume a job* *The following actions would be nice to have for a running job;* *1. Checkpoint a job;* *2. Move a job to another location;* *This accepts also composite jobs, and we need a way to refer to individual component jobs. Composite jobs also have unique job ids, and there is a tool to provide info on subjobs.*

Example	$ jobcont –j jobid [parameters]
	int retval = jobcont(struct *param);
	Rationale: we did not want to specify in detail the syntax of the parameters as this is installation independent and may not add much relevant information.

USE CASE: STEER JOB SUBMISSION

Identifier	*UC#jobsteer*
Goals in Context	*Send Job to Constrained Grid Computing Resource*
Actors	*User; Submission Daemon; User Program*
Triggers	*Desire to direct jobs to specific sites*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*User or Program logged into Grid;* *Preconditions for UC#jobsubmit*
Post-conditions	*Behaviour of basic job submission is influenced at extension point (steer submission)*
Basic Flow	*1. User specifies steering criteria. Following are acceptable:* *a. Specific computing element ID* *b. Proximity to specific storage server* *c. Select from available computing elements only those for which user-specified String is found in run-time environment published by Computing server* *d. Alternate resource broker (experiment-provided)* *e. Plugin cost function module for WMS broker (if plugin provided)* *2. System confirms reception of valid criteria*
Devious Flow(s)	*1. Possible problems* *a. The system cannot satisfy some of all the user requests. The user should be able to specify whether each request is a "hint" or an imperative request. In the first case the system will default to another choice, and in the second the operation is aborted. Hints may need to be ranked to resolve conflicts.* *b. The resource requested is temporarily unavailable. The user should be able to specify whether he wants the job to wait for the resource, default to a system choice or abort.* *c. Insufficient rights. The user does not have the right to use the specified resource. See a) above for the system response.* *2. N/A*
Importance and Frequency	*Unknown importance. Expect moderate frequency, users who wish to use "known" computing resource or do not trust resource broker.*
Additional Requirements	
Example	*Again we do not intend to give an example here as there are a lot of examples from Workload Management Systems.*

USE CASE: JOB RESOURCE ESTIMATION

Identifier	*UC#jobresest*
Goals in Context	*Provide estimate of resources needed for job; assist resource broker*
Actors	*User; Submission Daemon; User Program*
Triggers	*Expectation of significant resource consumption*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*User or Program logged into Grid;* *Preconditions for UC#jobsubmit*
Post-conditions	*Resource broker has estimates of various resources needed to execute basic job submission; influences UC#jobsubmit at extension point (resource estimation)*
Basic Flow	*1. User provides resource estimates. Following are acceptable:* *a. Estimated CPU time that will be used by the job* *b. Estimated memory usage* *c. Upper limit on CPU time needed* *d. Local disk space (scratch) needed* *2. System confirms reception of valid resource estimate description.* *3. System uses information to optimise the submission of the job*
Devious Flow(s)	*1. N/A* *2. Possible problems* *a. System is not able to make use of given estimate to optimise job submission. A warning is issued. In no case information provided by the user should be silently ignored.* *b. Resource estimate provided by the user exceed user or VO quota. The user should be notified and the job should be put on hold waiting for more resources to be allocated or to be killed.* *3. Optimisation is not possible or it is sub-optimal. The user should be informed.*
Importance and Frequency	*Moderate importance. Expect moderate frequency, likely used for most production job submissions.*
Additional Requirements	
Example	*Again no example as it is an extension of the job submit use case.*

USE CASE: JOB ENVIRONMENT MODIFICATION

Identifier	*UC#jobenvspec*
Goals in Context	*Modify or add to job execution environment; supply needed environment variables*
Actors	*User; Submission Daemon; User Program*
Triggers	*User program needs specific environment variables*
Included Use Cases	
Specialised Use Cases	
Pre-conditions	*User or Program logged into Grid;* *Preconditions for UC#JobSubmit*
Post-conditions	*Executing program on Grid will have access to shell variables via standard variant of "getenv" system; influences UC#JobSubmit at extension point (environment modification)*
Basic Flow	*1. User provides list of environment variables and their values*
Devious Flow(s)	*Apart from syntax problems, no devious flow expected.*
Importance and Frequency	*High importance. Expect high frequency.*
Additional Requirements	
Example	*Simple extension of job submit*

USE CASE: JOB SPLITTING

Identifier	*UC#jobsplit*
Goals in Context	*Distribute Jobs over multiple CPUs or sites*
Actors	*User; Submission Daemon; User Program*
Triggers	*Program consumes enough resources to make splitting advantageous*
Included Use Cases	*UC#jobsubmit*
Specialised Use Cases	
Pre-conditions	*User or Program logged into Grid;* *Preconditions for UC#jobsubmit*
Post-conditions	*Same as for UC#jobsubmit, except that job will have to run faster than on a single CPU. Output should be identical*[3] *to that produced by running program on single worker node. The job identifier returned should be recognized by other Grid tools as attached to multiple running instances of the program.*
Basic Flow	*To be decided.*
Devious Flow(s)	*Very dependent on the splitting strategy.*
Importance and Frequency	*High importance. Distribution is one of the most important motivations for HEP on the Grid.*
Additional Requirements	***Notes:** Most of what this use case originally said was already discussed under use case UC#**Error! Reference source not found.**, where job input specification via selection criteria was discussed. However in that use case, the question of job partitioning was not discussed. Job partitioning should be a separate use case, since input specification (UC#**Error! Reference source not found.**) is logically different from job partitioning. One real point of contact is that a selection-criterion input probably results in events from many different logical files, which makes partitioning perhaps more attractive from a cost perspective. An alternative is to make it an extension of the basic job submission. Suggest leaving this use case for discussion in round 2.*
Example	*Same remark as for job submit*

[3] Two forms of "identical" exist. Strong: all bytes of output are the same. Weak: output has identical statistical properties (same within error bars). Request more info from experiments on which options are needed and when.

USE CASE: PRODUCTION JOB

Identifier	*UC#jobprod*
Goals in Context	*Produce large quantity of official data product*
Actors	*Production Manager; Submission Daemon;*
Triggers	*Official decision on maturity of analysis program, conditions, calibrations*
Included Use Cases	*UC#dstran*
Specialised Use Cases	
Pre-conditions	*Preconditions for UC#dstran*
Post-conditions	*Output dataset(s) physically located on at least one Grid storage element; Output dataset(s) registered in DS metadata catalogue. Job identifier returned should refer to entire production job (expected that job splitting will occur so there will be multiple instances of the program)*
Basic Flow	*To be decided. Not obvious how production job is different from dstran+ job splitting.*
Devious Flow(s)	*This use case is composite, as it is made up of previously described use cases. Please refer to those for devious flow.*
Importance and Frequency	*High importance. Frequency several times per year per VO.*
Additional Requirements	***Notes:** need to integrate software publishing; UC#jobprod will likely use registered versions of the software, specifying "Logical Program Names" analogous to Logical File Names. The output, if file-based, should be generated such that a) all output file names are unique, b) output names may be constructed according to an experiment-defined recipe, and c) some sort of listing or database containing the complete list of output LDNs must be available.*
Example	***N/A***

USE CASE: ANALYSIS

Identifier	*UC#analysis*
Goals in Context	*User analysis*
Actors	*User*
Triggers	*Analyse data to produce scientific results for publication*

Included Use Cases	*Job submission;*
	DS access;
	DS upload (in case the TAG is registered to the Grid);
	Interactive event display;
	Update of DS metadata catalogue;

Specialised Use Cases	
Pre-conditions	*Availability of input DS on the Grid;* *Availability of production software on the Grid;*
Post-conditions	*TAG DS registered on the Grid (optional);*
Basic Flow	1. *User specifies job information including* a. *Selection criteria;* b. *Metadata DS (input);* c. *Output TAG DS (optional);* d. *Program to be run;* 2. *User submits job;* 3. *Program is run;* 4. *Selection Criteria are used for a query on the Metadata DS;* 5. *Event ID satisfying the selection criteria and LDN of corresponding DSs are retrieved;* 6. *Input DSs are accessed;* 7. *Events are read;* 8. *Algorithm (program) is applied to the events;* 9. *Output DS are uploaded;*
Devious Flow(s)	1. *N/A* 2. *See job submission use case.* 3. *User program crashes. JDL can specify appropriate corrective action.* 4. *User has no right to query the Metadata DS.* 5. *N/A* 6. *Input DS cannot be accessed. User program decides appropriate*

	corrective action. 7. *N/A* 8. *User program fails. WMS takes corrective action possibly specified in the JDL.* 9. *Output DS cannot be uploaded. Possible reasons are lack of privileges to upload DS or temporary failure of the system. In this case output DS's should be saved until appropriate corrective action is performed.*
Importance and Frequency	*Basic analysis job. High frequency and importance.*
Additional Requirements	*This is a special case of dstran, again a candidate for a section on isomorphic use cases.*
Example	*Same remark as for job submit use case*

Note: This use case is expanded in the HEPCAL II document.

USE CASE: DATA TRANSFORMATION

Identifier	*UC#dstran*
Goals in Context	*Creation of new data set starting from input data*
Actors	*User*
Triggers	*Need of output data for further processing*

Included Use Cases	*Job submission;*
	DS access;
	DS upload;
	Update of DS metadata catalogue;

Specialisation Use Cases	*Scheduled production with validation step;* *User specifies input data selection criteria;* *Output DSN is generated by the user application;*
Pre-conditions	*Availability of input DS on the Grid;* *Availability of production software on the Grid;*
Post-conditions	*DS registered on the Grid;* *DS registered in metadata catalogue with corresponding metadata;*
Basic Flow	1. *User submits job (jobsubmit) specifying* a. *Input DS;* b. *Metadata DS;* c. *Output DS;* d. *Program to be run;* 2. *Input DS is accessed (dsaccess);* 3. *Selected program is run;* 4. *Output DS is uploaded (dsupload);* 5. *Output DS optionally validated (dsverify);* 6. *Metadata catalogue is updated (dsmdupd);*
Devious Flow(s)	1. *N/A* 2. *See job submission use case* 3. *See dsaccess use case* 4. *User program cannot be run. The user JDL may specify? appropriate corrective action.* 5. *Output DS cannot be uploaded. This can be due to lack of authorisation to upload DS or temporary system failure. In both cases the output should be temporarily saved till corrective action is taken.* 6. *Validation of DS fails. In this case the user implements the necessary operation, either the DS is deleted, or it is saved and marked as non-*

	validated in the Metadata catalogue 7. *The user cannot update the Metadata catalogue. This is equivalent to non-being able to upload the DS in point 5.*
Importance and Frequency	*Basic data processing job. High frequency and importance.*
Additional Requirements	
Example	*Again this is a job submission use case, the difference is in the JDL, of which we think it is not useful to give an abstract example*

USE CASE: JOB MONITORING

Identifier	*UC#jobmon*
Goals in Context	*Monitor a single running job*
Actors	*User*
Triggers	*Curiosity; need to know the status of a job;*
Included Use Cases	*Grid login;*
Specialised Use Cases	
Pre-conditions	*A job has been submitted;* *User knows the job identifier;*
Post-conditions	
Basic Flow	1. *User submits a query using the job identifier as key and retrieves information about the job.* 2. *The amount and type of information retrieved can be specified via options in the query.*
Devious Flow(s)	1. *N/A* 2. *Possible problems* a. *Invalid jobid. Operation is aborted.* b. *Insufficient rights to access some or all the information for the specified jobid. Only the allowed information is returned, and a warning message specifying which information could not be returned. In no instance missing information should be silently skipped.*
Importance and Frequency	*High frequency and importance.*
Additional information	*Typical information:* 1. *Position in the queue;* 2. *Splitting information (if applicable);* 3. *Estimated time before running;* 4. *Estimated cost (arbitrary units);* 5. *Actual cost (arbitrary units);* 6. *Time and date of submission;* 7. *Time of start of execution;* 8. *Time of completion;* 9. *Priority;* 10. *Completion status;* 11. *CPU time used;*

	12. Real time elapsed; *13. Input I/O (amount and rate);* *14. Output I/O (amount and rate);* *15. CPU time left;* *16. Executable running;* *17. Dataset accessed;* *18. CE, WN and SE used;* *19. Current stout, stderr;* *20. Job status;* *21. Job environment variables;* *22. User who has submitted the job;* *23. User attributes for the job specified in the job catalogue;* *24. Queue used;*
Example	*Again there are abundant examples from WMS around.*

USE CASE: CONDITIONS PUBLISHING

Identifier	*UC#condpubl*
Goals in Context	*Publish (new) version of conditions set, making it available on the Grid*
Actors	*Conditions Database Librarian*
Triggers	*Official decision on appropriate conditions*
Included Use Cases	*Upload DS* *Update Grid-enabled Database*
Specialised Use Cases	
Pre-conditions	*Specification of condition set in appropriate format*
Post-conditions	*Physicists can specify usage of the condition set in the data-selection part of the job submission stanza.*
Basic Flow	*Person responsible:* 1. *Uploads the condition DS;* 2. *Updates dataset Metadata catalogue;*
Devious Flow(s)	*This use case is the composition of previously described use cases. Please refer to them for the devious flow.*
Importance and Frequency	*Basic analysis administration task. Low frequency but high importance.*
Additional Requirements	*Note: This use case is almost identical to the software publishing use case.* *Experiments want the authorization of the conditions database & repository to be Grid based.* *Questions:* 1. *How does one update a Grid database? Uploading a new "file" or using an API to update the records?* 2. *How does one access Grid database; is DB access "special" in that we know we won't want lots of it, or is it just another case of "file access"?* 3. *Do experiments want a standard language for conditions or do they wish to register in their own "proprietary" format?* 4. *Do experiments care about "advanced features" like having one symbolic name mapped to several condition sets, each appropriate for some run period?*
Example	*Too many questions open to provide an example*

USE CASE: SOFTWARE PUBLISHING

Identifier	*UC#swpubl*
Goals in Context	*Publish (new) version of software package, making it available on the Grid*
Actors	*Software Librarian*
Triggers	*Official release of new package or new version of package*

Included Use Cases	*Grid Login;*
	DS Upload;
	Software Catalogue Update

Specialised Use Cases	
Pre-conditions	*Availability of software package (source code and/or binary)*
Post-conditions	*Physicists can specify usage of this package in job submissions.* *Grid information system is updated with local existence of package*
Basic Flow	1. *Librarian prepares package* 2. *Librarian logs onto Grid* 3. *Librarian specifies:* a. *Location of package in appropriate format* b. *Symbolic name of package ("official name")* 4. *Software Catalogue is chosen automatically per VO* 5. *Catalogue is updated and package is transferred (if necessary) to proper location (this step is essentially DS Upload use case).* 6. *System provides confirmation of success/failure, including symbolic name and package location*
Devious Flow(s)	*Please refer also to the included use cases for the devious flow.* 1. *Package preparation fails. User corrective action is applied.* 2. *N/A* 3. *N/A* 4. *N/A* 5. *Catalogue update fails for lack of privilege or temporary system failure. Operation is aborted. See also the catalog update use case* 6. *System cannot confirm successful operation. Operation is aborted.*
Importance and Frequency	*Basic software administration task. Low frequency but high importance.*
Additional Requirements	*VO environment init routines may be impacted;* *Questions:* 1. *Do experiments want the software to be dynamically installed? You*

	may want to have software installed automatically via the same sort of "hints" that the replica manager will use. Explore "software environment manager" with similar feature to replica manager.
Example	*The use case is too general to provide a sample implementation*

USE CASE: VO WIDE RESOURCE RESERVATION

Identifier	*UC #resrev*
Goals in Context	*Block Grid resources for use by specific task during specified period*
Actors	*User, Computing Coordinator, VO User Catalogue Manager (UCM)*
Triggers	*Need of Resource Reservation: examples are scheduled production or conference deadline.*
Included Use Cases	*Grid login*
Specialisation of this Use Cases	*Specify the precise location of the resources to be reserved.*
Pre-conditions	*Actor is authorised to reserve resources.*
Post-conditions	*Resources are reserved for the user or group of users requesting them for a given period of time. This means that the user or group of users is guaranteed to have access to the reserved resources during specified time.* *The allocated resources are published in the Grid IS.*
Basic Flow	*Authorized actor:* 1. *Logs in to the Grid* 2. *Submits a request for resource reservation to the appropriate Grid service specifying:* a. *Type of resource* b. *Amount for each type* c. *Period of allocation*
Devious Flow(s)	1. *N/A* 2. *Possible problems* a. *Actor is not authorised to reserve resources. Operation is aborted.* b. *Request exceeds the allocation. This can happen in several ways, and depends on the "role" of the actor. The production manager can try to reserve resources for an analysis on the muon spectrometer quota on behalf of the muon group. If this quota is exceeded, she may decide to notify the muon group of the failure, or use her authority as production manager and allocate these resources from the general experiment budget.*
Importance and Frequency	*Unknown importance (it depends on the total availability of resources)* *In case this is necessary, it will be done before important conferences and for scheduled production, i.e. several times a year.*
Additional Information	*Resource reservation can be done by single user, privileged users or only by UCM, and can be subject to a quota, according to VO policy.* *Which resources can be reserved depends on the capability of the Grid accounting system. CPU, storage, network QoS and so on are all possible candidates. A VO will probably have a limit on the type and amount of resources that can be allocated.*

	This use case lets a user reserve a specified amount of resources, but experiments also asked for a tool to help make estimates on what is needed. *A weaker version of this use case should be provided by the middleware; otherwise jobs may often fail when the resource broker fails to account for expected storage usage of already-submitted jobs when it makes resource matching for newly submitted jobs.* *We suppose that the resource reservation call would return a sort of a token, which in its simplest form could be an alphanumeric key. Jobs will be able to access to the reserved resources specifying this token. A more sophisticated scheme based on something akin to the authentication/authorisation mechanism would be welcome.* *One must define what happens in case the reserved resources are not being used; sites will be quite unhappy about empty CPUs and jobs waiting in the queue.*
Example	*We do not consider very instructive to have an example here.*

USE CASE: VO WIDE RESOURCE ALLOCATION TO USERS

Identifier	*UC#userresmod*
Goals in Context	*Set/Modify Resource Allocation for groups/users of a VO*
Actors	*Computing Coordinator, Manager of the VO User Catalogue (UCM)*
Triggers	*Management decision on the Resource Allocation; need to change user's existing quotas/priorities*
Included Use Cases	
Specialised Use Cases	*Adding/Removing authorized users and groups to a VO* *Adding/Removing authorized users to a VO group* *Assigning resources to groups of users;*
Pre-conditions	*The UCM knows the list of involved users and the new values for their resource allocation.* *The UCM is recognized by the UC as its manager.*
Post-conditions	*Resource Allocation for (some of) the VO users has changed.*
Basic Flow	*VO User Catalogue Manager:* 1. *Connects to the VO UC (authorisation is part of this step);* 2. *Selects a user* 3. *Changes her/his priorities/allocation parameters* 4. *System displays updated changes* 5. *Possible return to step 3* 6. *Disconnects from the VO UC*
Devious Flow(s)	1. *Connection to the VO UC cannot be established or actor is not authorised to connect. Operation is aborted* 2. *Invalid user. Operation is aborted.* 3. *Actor is not authorised to perform operation. Operation is aborted.* 4. *System is not able to confirm operation. Operation is aborted.* 5. *N/A* 6. *N/A*
Importance and Frequency	*Enables proper resource sharing within a VO. Can be relatively frequent (i.e., once a day). We realise that there is very little detail here to guide implementers. However no further detail is possible without a practical example coming from an existing Grid.*
Additional Requirements	
Example	*See note for previous case.*

USE CASE: SIMULATION JOB

Identifier	*UC#simjob*
Goals in Context	*Creation of new data set using a simulation job*
Actors	*User*
Triggers	*Need of simulated data for further processing*
Included Use Cases	*Data transformation job;*
Specialised Use Cases	*Scheduled production with validation step;* *Output DSN is generated by the user application;*
Pre-conditions	*Availability of production software on the Grid;*
Post-conditions	*DS registered on the Grid;* *DS registered in metadata catalogue with corresponding metadata;*
Basic Flow	*User specifies job information but no input DSN;* *Data transformation use case is executed;*
Devious Flow(s)	*Composite use case. Please refer to the component cases for devious flow.*
Importance and Frequency	*Basic simulation job. High frequency and importance.*
Additional Requirements	
Example	*This is a job submission use case.*

USE CASE: EXPERIMENT SOFTWARE DEVELOPMENT FOR THE GRID

Identifier	*UC#softdevgrid*
Goals in Context	*Ensure that the experiment software is suitable for the Grid environment and performs properly*
Actors	*Software developer.*
Triggers	*Appearance of a new/updated software*
Included Use Cases	*Software publishing on the Grid, basic job submission;*
Specialised Use Cases	
Pre-conditions	*Software is provided by a developer; user is logged in to the Grid*
Post-conditions	*Software is proven to work and is used for Grid applications*
Basic Flow	*1. Software developer contacts users and provides them with instructions on how to access the new software;* *a. Software developer publishes the software, probably registers it as a dataset with a suggestive "development" software DS name.* *2. Software developer collects feed back from users who launched a task using the software in the Grid environment and checks the task status, this is a normal data transformation or analysis job, possibly with a transformation step. Software developer applies previously defined criteria for success of testing.* *3. In case of a success, the software is validated and published on the Grid; in case of failure, an iteration with the software developer is done*
Devious Flow(s)	*1. Developer is not authorised to register the software on the Grid. Operation is aborted.* *2. New software fails validation. User corrective action is taken.* *3. Actor is not authorised to publish the new software. Operation is aborted.*
Importance and Frequency	*A necessary test for any new/updated software meant for the usage in the Grid environment*
Additional Requirements	*Grid middleware and the proper VO environment are installed and deployed.*
Example	*This too is a job submission use case*