Crunching Real Data on the Grid

Practice and Experience with the European DataGrid

D. Groep (davidg@nikhef.nl) and J. Templon (templon@nikhef.nl)
NIKHEF

C. Loomis (charles.loomis@cern.ch)
Laboratoire de l'Accélérator Linéaire (LAL) Orsay, France

Abstract. The D0 experiment has used the European DataGrid (EDG) testbed to reprocess real data obtained from the Tevatron collider at the Fermi National Accelerator Laboratory. Pushing the use of the EDG software beyond feasibility studies has produced a set of recommendations for authors of experiment-level software, for producers of middleware, and for designers of grid systems. This paper describes the D0 experience with the EDG software and the resulting recommendations.

Abbreviations:
ALICE – High energy physics experiment at LHC
CERN – European Center of Particle Physics
D0 – High energy physics experiment at D0 region of FNAL
DMS – Data Management System
EDG – European DataGrid
ETT – Estimated Traversal Time
FNAL – Fermi National Accelerator Laboratory
HEP – High-Energy Physics
LAL – Laboratoire de l'Accelerator Lineare
LCG – LHC Computing Grid
LHC – Large Hadron Collider at CERN
MDS – Metadata and Discovery Service
PBS – Portable Batch System
R-GMA – Relational Grid Monitoring Architecture
SAM – Sequential data Access via Metadata
VDT – Virtual Data Toolkit

This article is dedicated to the Integration Team Mailing List*, a forum of hard-working DataGrid project participants where most of these problems were identified, discussed, and solved.*

> Measurement is a crucial component of performance improvement since reasoning and intuition are fallible guides ...
>
> Brian Kernighan and Rob Pike, In *The Practice of Programming* (1999)

1. Introduction

In the third quarter of 2003, the D0 experiment collaboration[1] reprocessed its experimental data from the most recent running periods. D0 is a high-energy physics (HEP) experiment located at the Tevatron collider at the Fermi National Accelerator Laboratory (FNAL). Physicists used the results of this reprocessing to prepare the physics results of the D0 experiment reported at international conferences. This article is based on an effort within the European DataGrid (EDG) project to perform part of the D0 reprocessing on the EDG testbed. The effort's use of "real" data distinguishes it from most other HEP grid studies to date, which have used grids only to generate simulated data.

The focus on "real work" brought with it such concerns as:

Practicality: How much added value does the Grid provide? How much extra work does it cost?

Bookkeeping: Were all data correctly processed? Where and why did jobs fail?

Deadlines: Will the results be ready six weeks before the winter conferences?

Pushing the grid beyond feasibility studies and exposing a grid toolkit to real-life working conditions indicate the state of the art in grid computing and suggest directions for future improvement.

2. The D0 Reprocessing Task

The work on which the current article is based dealt with the reprocessing of collision data from the D0 experiment. Processing and the subsequent reprocessing data refers to the analysis and reanalysis of the data collected from the thousands of particle detectors deployed as part of the experiment.

2.1. Processing and reprocessing

The data being analyzed are typically recorded whenever the front-end computers, scanning the detector outputs, identify a particularly interesting combination of signals. The data recorded after such a "trigger" comprise an *event*. Such an event is physically recorded as a large set (approximately 1 MB) of tuples of numbers. Each second, the experiment records around 100 of these events.

Physicists design algorithms which combine the event information with a geometric model of the detectors and calibrations of those detectors to infer physical properties of the initial particle collision. Physicists process the data using these algorithms to perform their analyses.

Reprocessing refers to the situation where this process has already been performed, but is being performed again. This typically happens when the research team has refined the algorithms, the geometric model, or calibrations. In the current case, the main improvement for D0 was in the subset of algorithms used for particle trajectory reconstruction.

2.2. The reprocessing activity

Reprocessing was performed at several sites or "clusters". The reprocessing activity at each of these clusters proceeds along the following general lines:

1. The reprocessing team at the cluster receives an assignment from the production manager. The assignment comes as a list of projects, each project containing on the order of 70 data files that require processing. Each data file contains 700 MB of information on average. Processing a file takes about 12 hours on a 1 GHz Pentium machine.

2. The cluster team contacts the D0 Data Management System (called SAM, for Sequential data Access via Metadata) and retrieves a project, meaning all the necessary files are transferred to the cluster site.

3. Jobs are submitted to the cluster's computational facility to process the files.

4. The cluster team uses SAM to store the files produced by the reconstruction program.

2.3. Reprocessing implementation on the EDG testbed

In the EDG effort, the entire EDG testbed functioned as a D0 cluster. The "cluster team" was located at NIKHEF. To perform the reprocessing on EDG resources, machinery had to be devised for the following processes:

1. making the input data files available on the EDG Data Management System (DMS);

2. adapting the D0 reconstruction software and environment to work on EDG resources;
3. distributing this adapted software to nodes running the reconstruction jobs;
4. monitoring progress of the jobs and bookkeeping of the results;
5. transferring the reprocessed results back into the SAM system.

Jobs were submitted to the EDG "cluster" using the standard EDG job-submission tools.

For interested readers, the final analysis of the EDG software by end-users[2] contains a full description of how the reprocessing was implemented.

3. Structure of This Document

In the following sections, we describe the issues encountered in designing, implementing, and using the machinery described in the list above. We use this information to draw conclusions about how work should, and in some cases should not, be carried out on grid facilities, and to make suggestions for future developments by designers and implementers of both grid middleware application frameworks.

4. Getting Data Onto Grid

4.1. Data Services need thin client APIs

Programs running on grid computing resources often need access to various kinds of data. These data are very frequently not locally accessible on the node which runs the program. The grid model is that the data are "in the grid", meaning that they can be accessed relatively transparently using grid services.

Current trends indicate that multiple "grids" will exist, and D0 is already confronted with this situation. As explained in Section 2, the input data were originally hosted at FNAL. These data can be easily accessed via SAMGrid tools. The natural approach was to install SAMGrid data-access client tools along with the D0 software, fetching the data directly from the hosting site and avoiding manual cross-registration and transfer into the EDG facility.

Unfortunately, the SAM system has been designed and deployed with a different strategy in mind. Each site participating in SAMGrid

must have a "SAM Station" installed. Client connections on a site talk to the local SAM Station, which then arranges for the transfer to and from other Stations. If the current work were to have used SAM data management, each participating site would have to install a SAM station.

The EDG project fortunately took the approach of having services accessed via lightweight client tools and installing those tools on worker nodes. These client tools can make use of the EDG data-management services possible even in the absence of any local EDG daemons. (One exception to this is R-GMA, the EDG information system, which requires access to a site server to query the information system.)

The two approaches can be perhaps more effectively contrasted by making an analogy with browsing the web. Anyone can browse the web, as long as they have a *web client tool* like Mozilla or Internet Explorer installed. This is like the EDG approach, and the one we feel to be most useful to experiments. The SAMGrid approach is analogous to allowing web browsing only for sites at which an http server is present.

4.2. Data Distribution is Important

A typical high-energy physicist first writes analysis software and tests this software locally on a small number of files. Once this software functions properly, the physicist runs the complete analysis in parallel by submitting hundreds or thousands of jobs each accessing different data files. The large simultaneous submission of jobs usually implies that a large number of these jobs will also start and run simultaneously. If the data is concentrated on a single storage service, then this places a large burden on that service and the network to that service.

Stressing a single service like this increases job failure rates, wastes resources, and increases bookkeeping costs. One way to avoid this problem on the grid is to ensure that the experimental data is distributed widely. This is both easy and completely under the control of the experiment itself. Alternatively, one could provide reservation services in the middleware. Doing so would require that the jobs specify exactly what data is needed, obtain a reservation for the transfer from the storage service, and communicate this reservation to the computing resource running the job. With the current state of the grid software, data distribution is the only viable near-term solution.

Centralization of data applies equally well to databases. During the latest evaluation of the EDG software, other experiments have experienced problems with job failures when accessing a central database[2]. Similarly, the Resource Broker accesses the state of the grid resources through a centralized database. In earlier releases the Metadata and

Discovery Service (MDS) from the Globus Toolkit(TM)[3] was used for this database. Poor query performance of the implementation caused large numbers of failed jobs[4]. Recent EDG releases use an OpenLDAP implementation which performs better. However, the centralized database problem still exists and has simply been pushed to a larger scale.

To maximize the performance of the grid, both distributed data and distributed databases are required.

5. Getting Software onto the Grid

5.1. A Grid is not a dedicated cluster

HEP users are accostumed to having full control over their clusters. Because of the direct control over resources that experiments have historically enjoyed, the software suites of experiments tend toward expedient rather than portable solutions. Examples include:

- Installing unusual third-party software,
- Using experiment- or laboratory-specific installation or setup tools,
- Programming to specific versions of third-party software

All reduce the portability of the code and make operation in a grid environment more difficult. The following subsections give several concrete examples where expedient solutions cause problems in the grid context.

5.1.1. *Third-party software packages*

The benefits of using third-party packages must be weighed against the additional package management costs incurred when those packages are not pre-installed on the computing resources. Many third-party packages are already installed because they add functionality that is missing in the standard Unix(TM) operating-system toolbox. MySQL[5] is one of these; relational databases are used for many purposes by both the grid middleware and the experimental software. As there is no "standard" relational database tool distributed with the Unix operating system. This is a clear case where the benefits of third-party software outweigh the management costs.

Software authors however sometimes use "special" third-party packages to make certain operations more convenient. An example is use of the "Z Shell" command shell by one experiment's software framework. The motivation for use of the Z Shell was its ability to do arithmetic

commands embedded in the shell. This is a weak motivation to require a special shell, as the same effect can be accomplished using the standard Borne shell (`/bin/sh`) and the POSIX-standard `bc` command.

The convenience brought by a slight simplification in arithmetic evaluations is heavily offset by dealing with package management for this special shell. The experiment software installation system will have to make test whether the shell has already been installed, and if not must install it as part of the application suite. In the latter case the entire suite will have to be compatible with the shell located in an arbitrary location instead of the usual `/bin/zsh` location. Further problems can arise since parts of the login sequence at the remote site (necessary to configure the account for access to grid services) sometimes fail for shells other than the standard `/bin/sh`.

5.1.2. *"Setup" packages*

Many HEP laboratories or institutes support multiple experiments and have developed flexible tools to configure the user's software environment. The FNAL package manager UPS/UPD provides a good example. With this tool, one can configure a user environment to use, for example, the ROOT data-analysis package[6] by issuing the command `setup root`. This modifies the user's command search path to include the ROOT executable directory and defines some other environment variables needed by ROOT.

Since experiments are used to complete control over computing resources, often such commands are deeply embedded within the experiment's computing framework. Consequently, the experimental software fails when the setup tool is not installed, even if the necessary adjustments to the environment have already been made by other means.

5.1.3. *Specific package versions*

In a grid context—a distributed computing environment operated by many different institutions with different aims—the versions of installed software will vary from site to site.

Application software which requires a specific version of a package effectively reduces the number of resources available to that application. To maximize the available resources, the application should only use mainstream features available in all of the deployed versions of a package. Moreover, the application should work around bugs in those versions. Writing code which is insensitive to the software version requires significant extra work from the author. There is a trade-off between the work involved to increase the portability and the ben-

efits from enlarged base of computing resources. Working in a grid environment pushes the balance-point towards increased portability.

For grid middleware, native language APIs may cause conflicts with application software. For example, C++ APIs are often dependent on the exact compiler version and vendor used to produce the API libraries. This choice may conflict with that used by the application itself or with that used by other middleware services. To maximize the utility of the middleware services, the creator of a service should provide a thin, language-neutral client interface which can be easily rebuilt by applications as needed. The recent move to web-service specifications and bindings is a positive step in this direction.

Overall, application software authors should program to minimize explicit version dependencies in external software, and middleware authors should work to make their services universally accessible.

5.2. Installation does not scale

Application software differs from standard system software in several important aspects:

- an experiment's full set of application software can be large, sometimes exceeding the size of the operating system
- versions of the application software change rapidly
- several versions of the application software are in simultaneous use on the same machine by different users
- users often require "slightly modified" versions of the application software for their own use
- the experiment supplies special versions of standard system packages either because they contain needed bug fixes or because the application software has been certified against the special version

Because of these differences standard software installation and management techniques to not work well for application software.

The application software for D0 was 0.6 GB for the EDG release and has grown to 2.3 GB in the LCG-2 release; similarly, the ALICE (a HEP experiment at the LHC) software has grown from 2.3 to 4.0 GB. These are single versions of the application software. Projecting to a grid containing thousands of nodes, covering hundreds of sites, and supporting dozens of different experiments hints at the needed level of scalability for application software installation.

EDG used two techniques: packaged applications distributed as part of the release and software installed dynamically by and for a single job.

Distributing the software as part of the release ensured a wide deployment of the application software with minimal effort on the part of the site administrator. However, this limited the virtual organization to a single version and updates were only applied with new EDG releases. In contrast, having a job install its own software was extremely flexible, allowing the job to tailor its software environment to its precise needs. The price of the increased flexibility was the larger use of disk resources and a further delay in the start of a job as the software was downloaded. The increased costs are more pronounced for shorter jobs.

The LHC Computing Grid (LCG) has adopted an intermediate strategy in which dedicated areas in the file system are reserved for each site's supported experiments. A special user then is given write-access to that area and is responsible for the installation of the experiment's software. This shifts the burden of installing and upgrading the software to the experiment. Having the software installed on each site removes the installation costs encountered with having a job install the software itself.

However when extrapolating to hundreds of sites, even the last solution will become unmanagable. The only long-term solution is a dedicated application software installation system which caches needed software and which makes visible to a particular job only those packages which are required for that job. The cache needs to be able to automatically download packages needed by a submitted job, lock those packages (and dependent packages) while in use, and remove packages which are no longer required. Presenting the large number of packages needed by an application in a usable manner while maintaining the ability to effectively manage the cache can only be achieved by using a virtual file system for the application software. Such a system would allow the dynamic loading of application software, be capable of caching to minimize redundant downloads of software, and minimize wasted resources from installed but unused software.

6. Monitoring

6.1. Distributed work needs distributed bookkeeping

The goal of HEP reprocessing is to produce a set of probabilities for certain types of nuclear reactions. HEP theories are then used to compute, according to the theory, what these probabilities are "supposed" to be. In this way one can judge how accurately the theories reflect physical reality.

Probabilities are calculated by ratios, dividing a number of trials into the number of times the trial resulted in the phenomenon of interest

being observed. In HEP terms, the number of successes is related to the number of times the reaction of interested was found in the processed data. Formulated in this way, it becomes obvious that one must ensure that all data acquired are correctly processed and accounted for — otherwise the probabilities will be underestimated. In a distributed computing environment, new mechanisms are needed in order to keep track of the reprocessing. The scientist can no longer inspect the file system of her local machine and verify that each data file has been processed. The scientist needs at least two pieces of information:

1. an indication, for each data file being processed, whether the processing was successful, and

2. in cases of a failure to process a file, a clear indication of which file failed, so that the reprocessing task can be resubmitted for this file.

The EDG toolkit was rather quickly found to be lacking for these purposes. There was one "grid job" per input data file; each of these jobs had a unique identifier (the "jobid") assigned by the grid scheduler. The scientist could query the scheduler and ask for the status of each of the tasks, based on this jobid. However in cases of failure, the jobid carried no information about which data file was being processed by this job. Furthermore, the failure status was relatively coarse; we found several cases of the reprocessing program having failed, but the scheduler reported success. From the scheduler's point of view, the job was correctly delivered to a worker node, and the output was correctly retrieved, so there was no failure.

The problem was solved by using the R-GMA system to provide bookkeeping. This is described in the next section.

6.2. Success of R-GMA model

R-GMA provides a facility that, from the user's perspective, is a grid-wide distributed database. The user (and her programs or agents) interact with the R-GMA system via SQL commands. Both information queries, as well as information publishing (*i.e.,* putting information into the database) can be performed from any location — as long as the client tool has been informed of an entry point (hostname of a server machine) into the system.

Information is published into the system via a *producer* process, and information is extracted (queries are performed) via *consumer* processes. The R-GMA system provides a mechanism to connect the producers and consumers, across site boundaries, thereby creating the impression of a distributed database. The D0 bookkeeping system was

implemented via an *archiver* process. Such a process *consumes* information published by the running jobs, transferring them to a back-end relational database to make the information persistent; the process also *produces* (publishes into the R-GMA network) data contained in the database whenever requested.

The most useful bookkeeping model was based around four events:

job submission: this records the jobid, submitting user, machine from which the job was submitted, for each job.

job instantiation: this publication allows users to verify that the task has actually reached a worker node. The name of the site (worker node location) is recorded, along with the actual reprocessing command line and the jobid. This table thus provides the link between jobid and data file.

reprocessing instantiation: this information is published immediately prior to starting the actual reprocessing. Jobs that survive until this stage have successfully installed all the software as well as the data file to be processed. This table records the start time, worker node ID, input data file name, and some hardware information about the worker node such as physical memory and clock speed.

job termination: information is published for this event immediately before terminating the job. The amount of CPU time and wall-clock elapsed time are recorded, along with reprocessing statistics and a reprocessing "exit status".

An SQL table was defined within R-GMA to hold data for each of these events. A fifth table held event records for failed commands, from the command wrapper described in Section 7.2.

7. Testing and Diagnostics

7.1. Whole grid testing needs real applications

An official release of the EDG software suite typically went through many stages of testing before it was deployed on the main infrastructure. The first round of testing was done by the developer(s) of each component. The amount of testing each component received at this stage varied widely according to the styles of individual developers. The next round of tests came during component integration, during which the various components were co-installed onto full systems and

tested for interoperability. When the integration team was satisfied, the full candidate release was deployed on the "development" testbed, a miniature copy of the main infrastructure, distributed over three sites. An official test suite was run and tests had to be passed before giving the green light for an official release.

One might expect that this would be sufficient to shake out all but the most devious bugs. Unfortunately, the D0 effort was quite effective at "producing" bugs in the software. This could be traced to two effects:

1. There was enough D0 work to fill the entire infrastructure, exposing configuration bugs at some sites not yet heavily used;
2. The usage patterns in the test suite missed some important features of the real usage, hence missing a few crucial bugs.

We provide below examples of both classes.

7.1.1. *Example configuration problem*
Many of the EDG client tools must be configured with entry points into the system. An example is the R-GMA (Relational Grid Monitoring Architecture) facility, for which one must specify a local server machine and a grid-wide "registry" machine. These entry points are typically configured at boot time by the automatic install system.

The entry points should also be upgraded whenever the tool itself (R-GMA in this instance) is upgraded, but with the EDG automatic install system there were a few loopholes; it was possible in some cases to upgrade a facility such as R-GMA without triggering the configuration scripts. In such cases, the entry points sometimes had incorrect values. Typically the default entry points pointed at a developer's development workstation (for obvious reasons).

During the early part of the D0 work, about 10% of the worker nodes were found to have this sort of dangling-developer configuration error. They were not found earlier since it is quite difficult to artificially fill the entire infrastructure with test jobs, and even then the test jobs tended to be an insufficient reflection of the real usage.

7.1.2. *Example of insufficient test coverage*
When the full-production tests started, about 20% of the jobs were failing, reporting that they could not register their output file to the Storage Element machine at NIKHEF. This was traced to the the `edg-rm` command—the main client tool for EDG data management—invoking certain FTP transactions in *active mode*. This mode requires both inbound and outbound network connectivity from the machine on which the command is invoked.

The command had been extensively tested on the development testbed and this error was not discovered. The reason is that all machines on the development testbed have both types of network connectivity. However the larger production clusters typically only have outbound connectivity for worker nodes, and the command failed at all sites for which inbound connectivity was blocked. (Except NIKHEF where the execution nodes and storage element are within the same firewall.)

7.2. Run crucial commands in wrappers

When a grid job runs on a remote site, debugging is difficult for at least two reasons.

- Debugging is intrinsically hard because one cannot use typical debugging tools. These tools are almost all designed to be executed from the same machine on which the command being debugged is executing.
- Debugging is further complicated because the EDG workload management system splits the standard output and standard error streams, returning them to the user as separate files. It is practically impossible to re-merge the streams in such a way that it is obvious which command caused an error.

This problem was solved by wrapping all but the most trivial commands. The wrapper executes the command in a subshell and captures the output (both streams) in a temporary data buffer. The command's exit status is checked; if the command succeeded, its output was passed to the job's standard output stream and execution was returned to the main shell. When an error status was observed, the last 256 characters of the buffered output was published, along with the entire command string, the name of the node running the job, a string identifying the job, date and time, and the exit status. These were published into the monitoring framework described in Section 6.2. This allowed an unambiguous determination of

- the job to which the failing command belonged;
- the exact command (including all arguments) that failed;
- the exact machine on which the command failed;
- the precise time at which it failed.

In practice, the last 256 characters of the output were sufficient to also determine the cause of the failure.

8. Production Work

In the course of using the EDG software in a production environment several design limitations–both globally and for individual services–became apparent. The following subsections describe some of these limitations.

8.1. Workload Management Requires Useful Resource Information

All users desire to maximize the utility of the grid for running their analyses; the grid scheduler (the Resource Broker in the case of EDG) attempts to match job requirements to available resources and to schedule jobs on the most desirable, viable resources. To compare the relative desirability of resources, the grid scheduler must use common, absolutely-normalized measures. Two typical measures are response time given in wall-clock time and cost given in real currency. A policy function which converts an arbitrary job description into a response time or cost is also required.

Using job cost as a basis for scheduling requires a "banking" system to maintain user balances and affect payments for services. Such an infrastructure was not deployed on the EDG testbed; consequently, the default scheduling parameter was the response time measured in wall-clock time. Specifically, the Estimated Traversal Time (ETT) is the time the next submitted job is expected to wait in the queue before it *starts* to execute. The policy function to calculate the ETT for a site assumes that the site implements a strict FIFO scheduling and that there is a single access point to the site's computing resources. The ETT is then an analytical calculation based on the number of jobs in the queue and the wall-clock time limit on the queue. The site publishes the ETT into the information system for use by the Resource Broker to schedule jobs.

The assumptions of the ETT algorithm (and the information schema used to publish it) do not model well the scheduling policies implemented on the various sites. Specifically, sites typically

- implement different priorities for different experiments or for individual users
- create different queues accessing the same resources
- impose limits on the number of jobs a single user can run simultaneously

Each of these causes the Resource Broker to distribute the jobs poorly.

We provide two concrete examples encountered within the EDG project. In the early stages of the project, the ETT datum was based mostly on the number of free CPUs at a site; if there were free CPUs, the estimated traversal time was zero, since in a strict FIFO system a new job would immediately be started on one of these free CPUs. Problems arose at sites implementing process caps for the supported experiments. Consider the situation where a site with fifty free CPUs places a limit of ten concurrent running jobs by members of the ATLAS experiment. After ten ATLAS jobs have been submitted to the site, an eleventh or further job will wait in the queue until one of the first ten has terminated, freeing a slot. The ETT will remain at zero due to the remaining free CPUs, encouraging the scheduler to continue to submit jobs to the site.

More current implementations of the ETT avoided this problem by basing the computation on the number of queued jobs. This has the opposite effect, however; continuing on the example above, the queued ATLAS jobs drive the ETT to a nonzero value. This value is correct for ATLAS, but not for other experiments for whom free CPUs are still available. Hence jobs which could run immediately on the site are directed elsewhere, and free CPUs go unused.

These sub-optimal distributions are not the fault of the Resource Broker, which is acting correctly on the information it had. Instead it is the fault of the mismatch between the actual site policy and that assumed by the EDG software (ETT algorithm and information schema).

There are three possible solutions to this problem:

1. enforce strictly the chosen scheduling algorithm on all sites

2. publish the scheduling policy and all relevant information into the information system

3. provide a service at each site which returns an estimate based on the provided job description

The first solution is the easiest to implement but intrusive; intrusive because it does not allow pre-existing resources to easily join the grid and goes against the grid philosophy of leaving policy issues to the site. The second solution is feasible if the number of different policies is small and the information to implement those policies readily available. This solution places the burden of the calculation on the Resource Broker but allows a scheduling decision to be made based solely on published information. The third solution is most useful if there are a large number of different policies and publishing the algorithms or

information would be difficult. This solution places the burden of the calculation on the resource itself but means that the Resource Broker must query directly candidate resources for the required information.

A prototype service [7] has been built and tested which is a hybrid between solutions 2 and 3 above. This service, for each site where it's run, provided an ETT value for each supported experiment. The system basis its estimates on the current and historical states of the computing resources and job population. The prototype was tested on the EDG testbed and shown to improve the accuracy of the traversal-time estimates by (on average) a factor of 20.

The important point of this experience is that a workload management system can make reasonable choices only if either

- the grid scheduler is fully aware of site policies, and has enough information about the workload at each site to evaluate the outcome of submission of new jobs, or

- the sites publish, or provide upon request, estimates carrying enough information to allow the scheduler to perform a simple unweighted ranking on these data.

8.2. The information system is the grid

The information system provides two critical services to the grid: a service index for locating various grid services and a monitoring service providing state information of those resources. Users consult the service index to find appropriate services; services themselves use the index to find other services. The Resource Broker uses the state information to optimize the scheduling of tasks.

These services are the core of the grid and without them the grid simply does not function. Consequently there is a higher level of reliability and robustness required of the information system. With R-GMA there was a long period during which various failure modes were discovered and fixed. Ultimately, the system did achieve a good reliability which allowed the grid as a whole to provide a stable computing platform for users[8]. There was a similar burn-in period for MDS from the Globus Toolkit which was used in earlier EDG releases[4].

Even though a reasonable reliability was achieved, work must be done to further improve the robustness of the system. R-GMA is less centralized than MDS, but still contains a centralized registry. This represents a single point-of-failure for the grid as a whole and must be eliminated. If a truly global grid is desired, then the information system must be inclusive of many different versions of services running

concurrently on the grid. Moreover, it must demonstrate that it can scale up from the current tens of sites to the thousands of sites in a world-wide grid.

8.3. Scalable service implementations

Scalabililty in real-world systems does not come only from scalable architectures, but depends as much on a scalable implementation of that architecture. The approach promoted by many of the typical object-oriented design patterns like "factories" and the modelling of the grid as a service-based system has lead to architectures that involve a large number of interacting entities. At the architectural level such an approach helps in defining a clean separation of concerns and therefore cleaner interfaces and a more maintainable design.

On the other hand, it is generally not a good idea to have the system design be a one-to-one mapping of these architectural concepts. In particular, not every service should give rise to a new process running on a server. Although such an implementation is very tempting in the prototyping phase (as no design work is needed to translate the architecture into implementation) it leads to immense problems in even a limited-scale deployments such as the EDG testbeds.

Examples of such one-to-one translations are abundant: the job manager as shipped with the Globus Toolkit version 2 provides a "guardian angel" for every job that is sent to a computing element. It allows the user to interact with the job once submitted, for example, to kill the job request or to obtain output and error streams. Architecturally, this is a per-job service and should be addressable as such. In practice, this was implemented as a stand-alone process, forked at job submission time, to keep the state of the submitted job. The process remained active and consumed resources on the computing element for the *entire* lifetime of the job—be the job queued, running, suspended, etc. Submitting any sizeable amount of jobs thus resulted in as many processes on the computing element head node as there were jobs running *and queued* on the CE, effectively bringing the entire system down.

Holding important state in processes has another severe disadvantage: if the process is killed, or if the machine crashes, all state is lost. The "guardian angel" example above applies here as well. When a computing element crashes, jobs running on the worker nodes controlled by the CE usually do not crash. However the job states (according to the workload management system) are lost during the crash, which resulted in these running jobs being declared as "lost" to the workload management system.

Many of these issues have been identified and solved during the lifetime of EDG. The problem of the job manager has been addressed by the gridmanager developed by VDT[9] and LCG[10] that uses a single process to manage multiple jobs on the CE and keeps the per-job state externally.

In other cases, advances in the operating system or the hosting environment may temporarily alleviate the problems: handling of processes with many threads improved significantly in recent (2.6) releases of the Linux kernel and thus, together with improvements in the Java Virtual Machine, implementations that use threads to keep state can defer facing these limitations. However, it seems a a good practice not to rely too heavily on such "external" improvements.

It is encouraging to see that the web services architecture take stateless services as a basis. Of course, web services have to date mostly been used in a transaction-oriented environment, where the web service basically provides interaction with a back-end database system.

Recent developments in the modelling of resources in the web services framework[11] are retaining the idea of stateless services that represent the underlying resource. This leads more naturally to an implementations that respect the limitations imposed by typical hosting environments and operating systems. In general, allocation of unmanaged memory structures is far cheaper and more scalable that the management of threads or processes. A modelling of resources through a stateless interface favours such scalable implementations.

8.4. Jobs need large temporary directories

The EDG toolkit uses the GASS mechanism of the Globus Toolkit to handle working directories for jobs. Because there may be hundreds of jobs running under the same virtual user account on a single site, the naming convention for the working directory contains a long string of random characters. This assures a very small chance of file-namespace collision between concurrent jobs. Unfortunately it generates a long absolute path. Many HEP programs also have a long directory path structure (relative to the home directory), and the combination of GASS working directory path plus software relative path exceeds hard-coded directory path length limits in legacy (FORTRAN) programs (e.g., 120 characters).

For this reason we chose to use the temporary directory (TMPDIR) feature as described in the POSIX standard[12] and provided by the Portable Batch System (PBS)[13], coupled with a much shorter randomization scheme based on the process ID of the running program. Sites either used PBSPro, used OpenPBS patched to implement a

transient TMPDIR per job, or pointed TMPDIR to some suitably large amount of free space. We quickly found that this temporary area needed to have roughly 5 GB of space per CPU for each worker node.

The advantage of the TMPDIR scheme is that some batch systems will completely remove the directory immediately prior to terminating the job. On those sites not supporting this feature, jobs began to fail after several days. This was traced to the occasional abnormal termination of a job which bypassed the normal self-cleanup step and left a few gigabytes of data in the temporary space. We learned to program a catch-all "cleanup" exception into our job script; this only fails if the node itself crashes.

9. Conclusions

The D0 data reprocessing software has been ported to the European DataGrid testbed. The resources available there have been successfully used to reprocess D0 data. The exercise has exposed problems with the application software, grid middleware, and architectural design which can be used to improve future grid programs.

The grid environment places a premium on portable programming. To ensure the greatest available capacity, the application software must use to the largest extent possible widely available software and mainstream features of that software. The code must be written to be insensitive to different versions of external software—both in terms of features and of existing bugs. In a distributed environment, the number of possible failures is much larger than for a small, dedicated cluster; the software must be written to tolerate and recover from failures.

Even for highly portable applications, specific non-standard external packages will be needed. To accomodate this, the grid should provide a service which allows the user to customize dynamically the software environment of the job. There must also be a method to discover standard features of the local computing environment, such as the location and size of a scratch directory.

EDG has largely made the middleware services available from thin clients. This is a useful feature which makes the services generally useful both inside and outside of a grid environment. This should be carried forward to future projects and extended by making the services' bindings language-neutral. The trend toward web-services is a good move in this direction.

To achieve near-optimal use of resources on the grid, the individual site policies must be accurately transmitted to the grid scheduler. Either the arbitrary site policies and necessary data must be transmitted

to the scheduler or the scheduler must consult sites to determine the cost or response time for a particular job.

The grid environment requires distributed data as well as distributed services for optimal operation. Distributing the data avoids stressing any particular data server or the network connecting that server. Distributing services which act as databases (such as the information system) has similar benefits.

Distributing the services improves the upward scalability of the grid and allows a large number of resources to be made available via the grid. However scalability in the other direction is also important. Grid software which scales down to the workstation would permit users to develop and test applications in the grid context and reduce porting costs now associated with using the grid.

The collaboration between the middleware developers and the end-users has been positive; both will hopefully use the experience to improve both the applications and the middleware for future grids.

References

1. For more information about the D0 experiment see http://www-d0.fnal.gov/.
2. European DataGrid (WP8), "Report on the Results of Run #2 Final Application Report: Assessment of the Testbed by HEP Application During Year 3", https://edms.cern.ch/file/428171/3/DataGrid-08-D8.4-0127-3-0-161203.pdf
3. The Globus Toolkit(TM), see http://www.globus.org/.
4. European DataGrid (WP6), "Evaluation of Testbed Operation: Overview of Testbed2 Deployment and Features", https://edms.cern.ch/file/375744/2/D6.6-1.1.pdf.
5. An open-source SQL database, see http://www.mysql.com/.
6. An data-analysis package commonly used in high-energy physics laboratories, see http://root.cern.ch/.
7. Hui Li, David Groep, Jeff Templon and Lex Wolters, "Predicting Job Start Times on Clusters," To appear in 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID2004), Chicago, U.S.A., Apr 19-22, 2004.
8. European DataGrid (WP6), "Final Evaluation of Testbed Operations", https://edms.cern.ch/file/414712/3.0/D6.8-3.0.pdf.
9. Virtual Data Toolkit, see http://www.lsc-group.phys.uwm.edu/vdt/.
10. LHC Computing Grid, see http://lcg.web.cern.ch/LCG/.
11. Web-service resource framework, see http://www-106.ibm.com/developerworks/library/ws-resource/.
12. POSIX standard (IEEE Std 1003.1-2001).
13. Portable Batch System (PBS), see http://www.openpbs.org/.