# Developing LHCb Grid Software: Experiences and Advances

I. Stokes-Rees, A. Soroko, C. Cioffi, University of Oxford, UK
A. Tsaregorodtsev, V. Garonne, CPPM, France
R. Graciani, University of Barcelona, Spain
M. Sanchez, University of Santiago de Compostela, Spain
M. Frank, J. Closier, CERN  G. Kuznetsov, RAL, UK

## Abstract

*The LHCb grid software has been used for two Physics Data Challenges, the most recent of which will have produced 90 TB of data and required over 1000 processor-years of computing power. This paper discusses the group's experience with developing Grid Services, interfacing to the LCG, running LHCb experiment software on the grid, and the integration of a number of new technologies into the LHCb grid software. Our experience and utilisation of the following core technologies will be discussed: OGSI, XML-RPC, grid services, LCG middle-ware, and instant messaging.*

**Keywords**

grid services, instant messaging, LCG, grid integration, OGSI, XML-RPC, computational grids, fault tolerance, matchmaking, pull scheduling

## 1 Introduction

LHCb is one of four particle physics experiments currently being developed for the Large Hadron Collider (LHC) at CERN, the European Particle Physics Laboratory. Once operational, the LHCb detector will produce data at a rate of 40 Mb/s[21]. This data is then distributed around the world for 500 physicists at 100 sites to be able to carry out analysis. Before this analysis of real physics data can begin simulations are required to verify aspects of the detector design, algorithms, and theory. LHCb has worked closely with the LHC Computing Grid (LCG)[5], which is coordinating the common computing strategy for the four LHC experiments (LHCb, Atlas, Alice, and CMS). The LHCb computing model intends to utilise many aspects of LCG but will have, in addition, experiment specific components.

DIRAC (Distributed Infrastructure with Remote Agent Control) is the resulting LHCb grid computing environment which evolved from largely standalone Python-based agents and services, distributed across various physics institutes, into a grid-aware package which makes use of numerous Grid middle-ware developments. It incorporates LCG computing resources and functionality, while also, critically, allowing the integration of non-LCG resources (see figure 1). This paper recounts our experience of developing DIRAC, integrating it into the LCG grid environment, making use of existing middle-ware services and libraries, and advances from incorporating new technology such as instant messaging into our architecture.

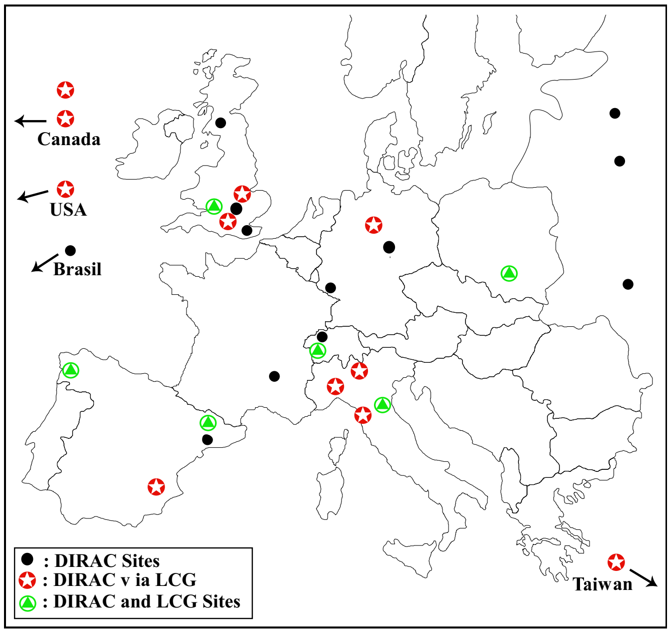

***Figure 1:*** *Sites running DIRAC. This includes a mixture of LCG sites and conventional cluster computing centres.*

We will start with a description of the DIRAC architecture in section 2, outline key features and advances in section 3, describe our experience developing and using grid software in section 4, and conclude in section 5 with perspectives for the future of DIRAC and computational grids.

## 2 Architecture

DIRAC is designed following a lightweight Agent/Service model, which emphasises a *Service Oriented Architecture*, rather than single, monolithic, components. It is meant to provide a scalable high throughput generic grid computing environment for uncoupled or loosely coupled long running computational tasks, possibly requiring significant input data and producing large volumes of output data. The basic design objectives are to support: 100,000 queued jobs; 10,000 running jobs; and 100 sites.

The architecture is divided into four areas: Services, Agents, Resources, and User Interface. The core of the system is a set of independent, stateless, distributed *Services*. The services are meant to be administered centrally and deployed on a set of high availability machines. *Resources* refer to the distributed storage and computing resources available at remote sites, beyond the control of any central administration. Each computing resource is managed autonomously by an *Agent*, which is configured with details of the by the local administrator. The Agent runs on the remote site, and manages the resources, job monitoring, and job submission.

The *User Interface* API allows access to the Services, for control, retrieval, and monitoring of jobs and files, and has been incorporated into command line tools, GUIs, and web sites. A complete GUI interface for managing LHCb jobs has been produced by the Ganga project[2] and this uses the DIRAC Client API to interface to the Services for job submission, monitoring, and retrieval.

The general separation between Services and Agents is that Services are *stateless* and *reactive*, whereas Agents are *stateful* and *proactive*. The Services can be distributed across several machines, or run from a single "server". This allows easy replication for redundancy and load-balancing.

### 2.1 Job Management Services

Jobs are described using the text based ClassAd Job Description Language (JDL) designed by the Condor project for use with the Condor Matchmaking scheduling system[23]. The JDL file is submitted to the Job Receiver Service which registers the job in the Job Database and notifies the Optimiser Service. The Optimiser Service sorts jobs into different job queues and dynamically reprioritises queue ordering. Agents, which run on the distributed computing resources, monitor resource availability. When they detect "free slots", they submit a job request to the Matchmaker Service, which interrogates the various Job Queues and returns a suitable job, based on the resource's profile. These components are illustrated in figure 2.

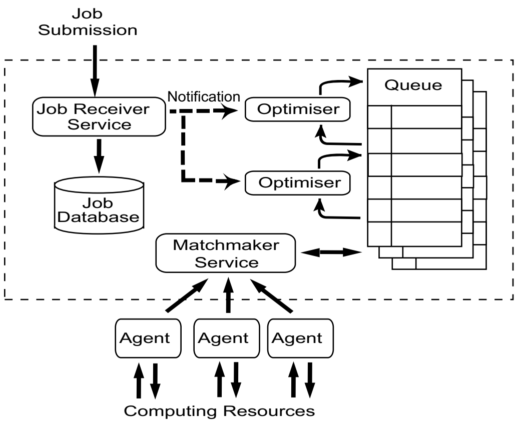

***Figure 2:*** *Core DIRAC Services*

### 2.2 Agent

The Agent manages jobs locally, either executing it directly, or submitting it to the Local Resource Management System (LRMS) (usually a batch system such as PBS or LSF). The current system supports PBS, LSF, BQS, Condor, Globus, EDG/LCG, Fork, and InProcess resource types. The Agent monitors the progress of the job and sends status updates to the Monitoring Service. The Data Management Services can be accessed directly by the job, or information in the JDL can be used by the Agent to handle staging of input and output files.

The Agent is deployed on a computing resource and interacts directly with it. This Agent is under the control of the local site administrators and can be run and configured to operate in a variety of different ways, dependent upon site policy and capabilities. The Agent is easily deployable on a site and only needs outbound Internet connectivity in order to contact the DIRAC Services.

The Agent design consists of a module container and a set of pluggable modules. The modules are executed in sequence. Typically a site runs several agents each having its own set of modules, for example job management modules or data management modules. This feature makes the DIRAC Agent very flexible, since new functionality can be added easily, and sites can choose which modules they wish to have running.

### 2.3 Data Management Services

The DIRAC Data Management Services provide fault tolerant transfers, replication, registration, and meta-data access for files both at DIRAC computing centres and long term mass storage sites.

A *Storage Element* (SE) is an abstracted interface to internet-accessible storage. It is defined entirely by a host, a protocol, and a path. This definition is stored in the Configuration Service (see section 2.4), and can be used by any Agent, Job, Service or User, either for retrieving or uploading files. Protocols currently supported by the SE include: gridftp, bbftp, sftp, ftp, http, rfio or local disk access. The SE access API is similar to the Replica Manager interface of the EDG project.[17]

The *File Catalogue Service* provides a simple interface for locating physical files from aliases and universal file identifiers. This has made it possible to utilise two independent File Catalogues, one from the already existing LHCb Bookkeeping Database, and another using the AliEn File Catalogue from the Alice experiment[3]. In the recent LHCb Data Challenge they were both filled with replica information in order to provide redundancy to this vital component of the data management system, and to allow performance comparisons to be made.

Within a running job, all outgoing data transfers are registered as *Transfer Requests* in a transfer database local to each Agent. The requests contain all the necessary instructions to move a set of files in between the local storage and any of the SEs defined in the DIRAC system. Different replication, retry, and fail-over mechanisms exist to maximise the possibility of successfully transferring the data (see section 3.4). These also include registration of the file in appropriate File Catalogues and entry of appropriate file meta-data.

## 2.4 Configuration Service

It is necessary for Services, Agents, Jobs, and Users to be able to "find" each other, and to learn about the properties of the other components. This is a common issue in all service oriented architectures. Services also need their own configuration mechanism. It was felt that the existing approaches, such as LDAP, UDDI[14], MDS[11], and R-GMA[26, 7], were powerful, yet complex, and required significant infrastructure to utilise.

As such, and in keeping with the principles of simplicity and lightweight implementation, a network-enabled categorised name/value pair system was implemented, which overloads the Python ConfigParser API and the Microsoft Windows INI file format. Figure 3 shows an example of this format. Components which use the Configuration Service do so via a *Local Configuration Service* (LCS). This retrieves information from a local file, from a remote service, or via a combination of the two.

```
[SectionA]
first_option  = some value
second_option = 17

[SectionB]
old_option    = local_config.ini
new_option    = http://example.org/
```

***Figure 3:*** *Example Configuration Service information*

This system proved to be very robust, and the multi-threaded XML-RPC server was able to handle high volumes of concurrent requests, as illustrated in figure 4. The simplicity of the data format and the access API allowed site administrators to easily edit their local configurations, and for multiple Configuration Service implementations to be easily developed.

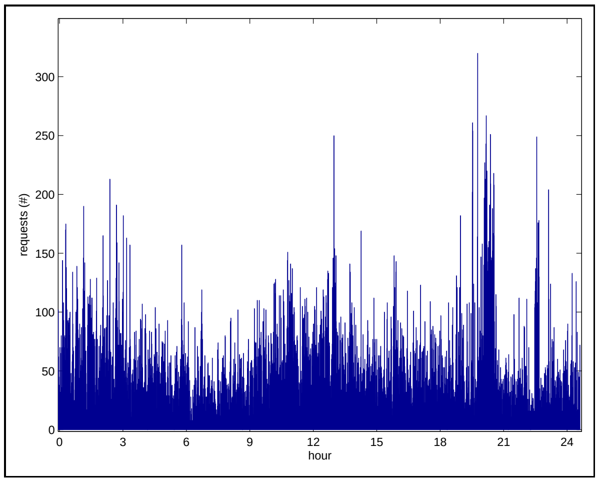

***Figure 4:*** *XML-RPC requests per minute to the Configuration Service over a 24 hour period*

## 2.5 Monitoring and Accounting Services

The *Job Monitoring Service* provides an interface for Agents and Jobs to update job state and for other Services or Users to query job state. This only retains information for jobs which are active in the system. Jobs which have completed or failed are eventually cleared to the *Job Accounting Service*. There are three access modes to the Monitoring Services: an API, a web-interface, and command line tools.

## 3 Key Features and Advances

This section discusses four aspects which have been key to the success of DIRAC: the pull scheduling paradigm, lightweight modular agents, the use of instant messaging, and mechanisms to provide fault tolerance.

## 3.1 Pull Scheduling

DIRAC emphasises *high throughput* rather than *high performance*. This idea was first presented by the Condor project[6], from which DIRAC borrows heavily in terms of philosophy for designing generic distributed computational systems[20]. It advocates immediately using computing resources as they become available, rather than attempting global optimisations of all jobs over all resources. In the Condor approach, which we will call the *pull* paradigm, computing resources request computing tasks by announcing their availability. In contrast a *push* paradigm has a scheduler which monitors the state of all queues and assigns jobs to queues as it wishes.

For *push* scheduling to work, all the information concerning the system needs to be made available at one place and at one time. In a large, federated, grid environment this is often impractical as information may be unavailable, incorrect, or out of date. Even if it is available, job allocation complexity grows quadratically with the number of jobs and resources, where every possible allocation combination must be evaluated to select an optimal schedule. While there are efficient heuristic approaches that in practice approach an optimal solution, such algorithms generally require complete and up to date information regarding system state, and are typically designed to operate on homogeneous computing resources with $10^2 - 10^3$ queued jobs.

As a result of this, *push* scheduling in a grid environment has proven to be problematic. By contrast, the DIRAC Central Services simply maintain queues of prioritised jobs (see section 2.1) and allocate the highest priority job which matches the resource's profile. The Condor Matchmaking libraries facilitates this dynamic definition of resource availability, as opposed to the traditional batch system which contains queues consisting of static characteristics[23].

The previously difficult task of determining where free computing resources exist is now distributed to the local Agents (see section 2.2) which have an up to date view of the local system state. The central Matchmaker Service only compares one-on-one requirements, with a round-robin on each of the job queues until it finds a job which can run on that resource. Since jobs are grouped into queues based on common requirements, the worst case is that each Agent job request will be compared against each queue once, where the number of queues is much less than the total number of queued jobs.

Both long matching time and the risk of job starvation can be avoided through the use of an appropriate Optimiser to move "best fit", "starving", or "high-priority" jobs to the front of the appropriate queue. This frees the match operation from necessarily considering all the jobs within the system. As reported elsewhere[10], this allows a mixture of standard and custom scheduling algorithms.

Figure 5 shows the match times for jobs during LHCb DC04. 97% of the time this operation takes less than one second even with tens of thousands of queued jobs, thousands of running jobs, and dozens of Sites requesting jobs concurrently. In comparison, the LCG Resource Broker (RB) requires several seconds to several minutes to schedule jobs, with only a single user accessing the RB, and on the order of $10^3 - 10^4$ queued and running jobs.

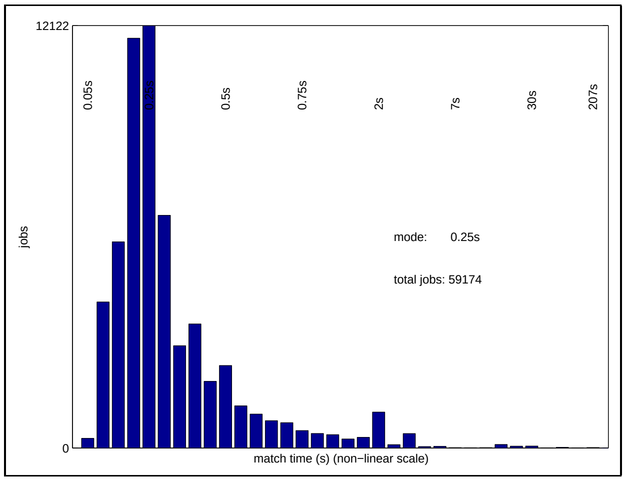

***Figure 5:*** *Match time distribution for 60,000 jobs during DC04*

## 3.2 Lightweight Modular Agents

By providing simple abstractions of *Computing Elements* (CE) and *Storage Elements* (SE), and exposing simple APIs to the Core Services, it was possible to implement lightweight Agents (see section 2.2) which can be installed and run entirely in user-space on any Computing Resource. This allows the rapid utilisation of heterogeneous systems in a federated manner — the most general objective of computational grids. Local site administrators simply install a one megabyte self-contained package with all the necessary software for the DIRAC Agent.

The configuration allows local policies on queue usage to be applied, and selection of specific Agent modules to run. This modularity gives administrators great flexibility and control, and makes it easy to write custom modules.

The only pre-requisite is a recent version of the Python interpreter and outbound internet connectivity, in order to contact the DIRAC Services. This allows the agent to run under virtually any computing and network environment, including behind firewalls and private networks utilising Network Address Translation (NAT) to reach the Internet. Installation entirely in regular user-space mitigates the security risks present in software which requires "root" access and system wide installation.

Agents can operate in a *cycle-scavenging* mode at the cluster level, where they only request and execute jobs when the local resources are underutilised. This idea comes from global computing models, such as SETI@Home, BOINC, and distributed.net [25, 4, 8], which perform cycle-scavenging on home PCs.

## 3.3 Instant Messaging for Grid Services

DIRAC has incorporated an instant messaging framework into all the components: Services, Agents, Jobs, and User Interface. This provides reliable, asynchronous, lightweight, and high speed messaging between components. Public demand for instant messaging has led to highly optimised packages which utilise well defined standards, and are proven to support thousands to tens-of-thousands of simultaneous users. While these have primarily been for person-to-person communication, it is clear that machine-to-machine and person-to-machine applications are possible, and it is in these areas DIRAC has demonstrated a novel application of the technology.

While the DIRAC Services expose their APIs via XML-RPC, due to the simplicity, maturity, and robustness of this protocol, the need to expose a monitoring and control channel to the transient Agents and Jobs led to the use of instant messaging. No *a priori* information is available about where or when an Agent or Job will run, and local networks often will not allow Agents or Jobs to start an XML-RPC server that is externally accessible. This suggests a client-initiated dynamic and asynchronous communications framework is required.

Extensible Messaging and Presence Protocol (XMPP), now an IETF Internet Draft[15], is currently used in DIRAC. This has grown out of the open-source, non-proprietary, XML base Jabber instant messaging standard. XMPP provides standard instant messaging functionality, such as one-to-one messaging, group messaging ("chat"), and broadcast message. An RPC-like mechanism exists called Information/Query, (IQ) which can be used to expose an API of any XMPP entity. The *roster* mechanism facilitates automatic, real-time monitoring of XMPP entities via their *presence*.

The DIRAC Services use XMPP in places where fault tolerant, asynchronous messaging is important. For example, the Job Receiver Service uses XMPP to notify the Optimiser Service when it receives a new job. When the Optimiser gets this message, it will then sort the new job into the appropriate queues. The IQ functionality has the potential to allow users to retrieve live information about running jobs, something which is critical for interactive tasks, or for job steering. It also greatly facilitates debugging and possible recovery of stuck jobs.

MPP is specifically designed to have extrememly lightweight clients, and gracefully handles dynamic availability of entities, buffering all messages until an entity is available to retrieve them. By matching the XMPP IQ functionality to standard XMPP messages, it is possible for users with a standard XMPP client to locate and communicate with Agents, Jobs and Services from anywhere. This has already been put to good use in DC04 for controlling and monitoring the state of Agents.

The two main outstanding issues for the use of Instant Messaging are the security and authentication implications of a "tunnelled" control chanel into remote computing sites, and the scalability to tens of thousands of XMPP entities communicating across the same instant messaging network. For the first, a group at Lawrence Berkley Laboratory (USA), have developed an XMPP server which accepts Globus x509 GSI certificates for authentication. This is a positive step and we will be working with them to investigate how this can be used for end-to-end security and authentication. For the second, many high performance commercial XMPP servers are available, however the freely available open source servers still demonstrate some robustness and scalability issues which have created problems when XMPP is used heavily, for example by thousands of jobs broadcasting status updates (see section 4.4).

## 3.4 Fault Tolerance

In a distributed computing environment it is impossible to assume that the network, remote storage, and remote services will constantly be available. The result is that any remote operation may:

- fail to connect to the remote resource
- stall
- fail to complete properly

These failures often are not permanent, so a retry at a later time or to an alternate equivalent resource may be successful and allow the parent operation to complete, albeit with a delay incurred due to the retry. In order to cope with these failure modes the following mechanisms were used:

**Retry** Many commands retry with a time delay in order to overcome any network outages, service request saturation, or service failure and restart.

**Duplication** Numerous services have a duplicate backup service available at all times.

**Fail-Over** When contacting critical services, after the retry limit is reached, a request to an alternate service is attempted.

**Caching** In the Local Configuration Service, the remotely fetched data can be cached locally for future retrieval.

**Watchdog** Monitors components to ensure continuous availability and restart on failures.

All Services and Agents are run under the *runit* watchdog[22]. This provides numerous advantages over cron jobs or sysv style init scripts. It ensures that the component will be restarted if it fails, or if the machine reboots. It also has advanced process management features which limit memory consumption and file handles, so one service cannot incapacitate an entire system. Automatic time-stamping and rotation of log files facilitates debugging, and components can be paused, restarted, or temporarily disabled. Furthermore, none of this requires root access.

## 4 Implementation and Operational Experience

DIRAC has been developed over the past two years by a core team of two to four developers, with extensive input, contributions, and testing, and deployment feedback from the LHCb Data Management Group, and computing centre administrators. It has aimed to bridge the computing requirements of LHCb with the capabilities available at the collaborating computing centers, and to provide a basis for evaluating grid computing approaches, particularly the functionality offered by the LCG environment.

### 4.1 Historical Background

The initial system developed in 2002-2003 was specialised exclusively for performing LHCb physics simulation jobs which Agents pulled from a pool of outstanding simulation jobs[27]. For 2003-2004 the emphasis shifted to providing a generic computational grid system which could incorporate new developments at that time around the *Open Grid Services Architecture* (OGSA)[12], and the Globus Toolkit 3 (GT3) implementation of the *Open Grid Services Infrastructure* (OGSI)[28].

At the same time the EDG project[1] was in the process of delivering the software from its three year development phase to the LCG project[5] which was meant to deploy and stabilise the EDG software. Due to performance shortcomings of the EDG software, a refactoring of the EDG architecture was proposed under the auspices of the ARDA-RTAG (Architecture Roadmap for Distributed Analysis — Requirement Technical Assessment Group)[19]. This refactoring was to take the form of a service decomposition with clearly specified interfaces. This was done to improve the conceptual organisation of the architecture, decrease dependencies between components, facilitate the incorporation of new services, and make possible the substitution of customised or alternative services.

The early version of DIRAC had already followed a service oriented architecture, so it was hoped this could then be refactored into a Python based set of OGSI Grid Services, implementing the ARDA-defined interfaces. The ARDA refactoring proposal was then handed over to the EGEE (Enabling Grids for E-science in Europe) project[18], which is the successor to the EDG project.

### 4.2 OGSA and OGSI

The DIRAC team strongly supports service oriented architectures, therefore the framework proposed by OGSA and specified in detail by OGSI was seen as a very positive step towards increased interoperability between grid software components. Several months of intensive work was invested in developing DIRAC Services as Java GT3 components, however the DIRAC team abandoned this work shortly before Globus and IBM jointly announced their intent to discontinue OGSI and in its place proposed WSRF (Web Services Resource Framework)[13].

In principal, the DIRAC team supports the idea of dynamic, stateful, transient Grid Services, as compared to Web Services which are static and stateless. The mechanisms for security, lifetime, service data, and publish/subscribe event notifications are all reasonable, however in the end we found OGSI was unworkable for the following reasons:

**heavyweight and complex** impossible to develop lightweight clients, difficult to run as a regular user, significant infrastructure required for deploying Grid Service container

**not standards compatible** unable to leverage existing Web Services tools

**poor documentation** for installation, maintenance, debugging, development

**poor implementation** many bugs in GT3 and constant exceptions being thrown

Together these made it difficult to develop, debug, deploy, maintain OGSI Grid Services. Similar experiences were recorded by others[24]. We also investigated using a pure Python implementation of OGSI, pyGridWare, prepared by Lawrence Berkley Laboratory (USA), but this was not sufficiently complete to be useable. While it is understood that the more recent versions of GT3 have corrected many of the early technical problems, the combined facts that OGSI no longer had a future, and the complexity of development under GT3 forced development of DIRAC to return to Python and XML-RPC.

### 4.3 Development and Deployment Environment

The DIRAC team has utilised CVS and Savannah for software management. Both have proved to be invaluable. Savannah, which is a branch from the popular SourceForge project management environment, is available at CERN and integrated with the CERN CVS repository, and CERN AFS file system. Both users and developers have made extensive use of Savannah for bug reporting, task prioritisation, support requests, software documentation, and software releases. Use of Savannah also allows easy migration of bugs from DIRAC to other software groups, such as LCG or physics software teams.

The Core Services for the initial test deployment were installed on two servers at CPPM Marseille. When the production deployment was instaled on high availability servers at CERN, the test deployment became an emergency fail-over system and also moved to CERN. At times during development the services were spread across servers in Oxford, Marseille, and at CERN, demonstrating the effective distribution of a single DIRAC "installation" with different services installed at different sites. Batch system integration was developed using PBS on the Oxford Physics cluster, Condor and Globus using the Oxford e-Science Centre NGS clusters, BQS at the IN2P3 Computing Centre in Lyon, and LSF at CERN.

## 4.4 Data Challenge 2004

The DIRAC system has been used for the LHCb Data Challenge 2004 (DC04), held from May to July 2004. DC04 had three goals: to validate the LHCb distributed computing model based on the combined use of LCG and conventional computing centres; to verify LHCb physics software; and, to generate simulation data for analysis. Hundreds of thousands of jobs were run, consuming over 400 processor-years of CPU power, and producing 90 terabytes of data. This data was redistributed across the centres for both organised (i.e. planned and predictable) and chaotic analysis of the results.

The system operated smoothly with a sustained level of over 2000 running jobs, and 600 gigabytes of data generated and replicated daily. Figure 1 shows the participating sites, and figure 6 shows a snapshot of the running job distribution. Once installed, the DIRAC Agents ran autonomously and restarted after failures or reboots. A central team watched the monitoring system and alerted site administrators when problems were detected.

The twenty participating sites varied enormously in size, from 20 CPU clusters shared heavily with other users to large 500+ dedicated CPU clusters. The involvement of the site administrators also varied signficantly. A mailing list and weekly phone conferences allowed the DIRAC software developers, site administrators, and data challenge managers to discuss progress and solve problems.

Another 20 sites were accessed via LCG, and are discussed in section 4.5. In total, these 40 sites provided more than 3000 worker nodes. At the typical level of 5000 globally queued jobs, the Matchmaker Service responded to Agent job requests in an average of 0.25 seconds (see figure 5). More than 40,000 jobs were completed in the month of May with an average duration of 23 hours, running on average at 93% load, the remaining 7% being data transfer. Each job produced 400 megabytes, which was replicated to several sites for redundancy and to facilitate later data analysis. To date the system has produced, stored and transfered 20 terabytes of data.

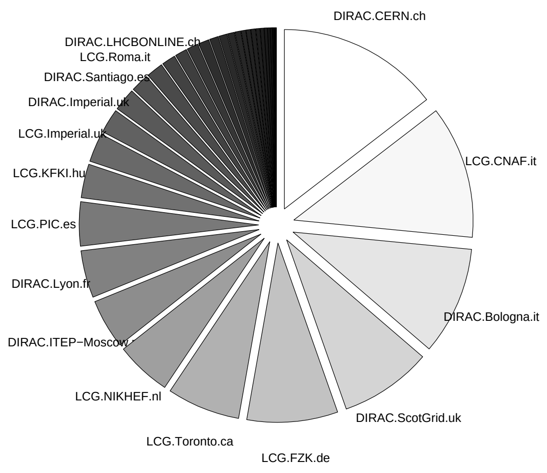

***Figure 6:*** *Representative snapshot of running jobs per site during DC04. Notice a mixture of "standard" sites (prefixed DIRAC), and sites accessed via LCG (prefixed LCG).*

At the time of writing, there have been four major outages in the DIRAC core services availability which resulted in jobs failing, jobs stalling, or sites failing to get new jobs:

**Inappropriate Intervention** One of the high availability core servers, which is monitored 24 hours a day by CERN IT staff, reached 90% of full on the local hard drive. This was due to a large and very actively used database on the server. The DIRAC team were notified in the morning, and were discussing a strategy for gracefully stopping the services, moving the database, and restarting the services when we received a second notification, approximately 6 hours later, that a large, and growing, one gigabyte file had been deleted from the server. Due to open file handles holding the file, the MySQL process was killed. Fortunately the damage was limited to the loss of all queued jobs in the system. CERN IT were asked not to unilaterally stop the database process and delete large database files in the future.

**Security Scan** CERN hosted servers are subjected to regular security scans to detect vulnerabilities. These scans take several hours or more to complete and flood a server on all ports with large pseudo-random operations. The experience during DC04 is that this largely disables the server from responding to legitimate requests and amounts to a Denial of Service (DoS) attack. Negotiations are underway to try to prevent the desireable security scan having the undesireable effect of disabling the server in question.

**Distributed Denial of Service** Early efforts to incorporate instant messaging into all aspects of DIRAC resulted in very effective distributed denial of service attack on the server hosting DIRAC and the instant messaging hub. Thousands of jobs were simultaneously sending status information, and in many cases were (unnecessarily) sharing this information with each other, resulting in an extrememly high, and unmanageable, message volume which compromised the performance of other services running on the same server.

**Network Failure** CERN experienced a site wide network failure for approximately one day due to efforts required to isolate an internally compromised machine. All services were unavailable during this time, and it was proposed that a fail-over system be prepared at an external site. This was not completed due to the infrequency of extended total network failure at CERN and the effort required to configure and manage a second DIRAC system.

Only a few significant bugs were identified in the DIRAC software. These appeared early in the data challenge and were quickly resolved. They generally centered around service scalability and availability, requiring the implemention of operation buffering, timeouts, and fail-over mechanisms on both the client and server sides (see section 3.4). Once these early bugs were resolved, "standard" computing sites observed stable performance. The small size of DIRAC, buffering of transfer requests, use of a local job database, and independence from the local batch system, all meant that it was possible to stop the Agent, even while jobs are still running on the site, perform a software update, and recommence the Agent loosing existing jobs or transfers.

Data Management presented the greatest overall challenge. A number of sites experienced significant data transfer delays or failures, resulting in transfer backlogs. Large sites would quickly fill their queues with hundreds of jobs, producing 40 Gb of data and all finishing at approximately the same time, therefore saturating one or both of the site's outbound bandwidth or the target server's inbound bandwidth. Although DIRAC supports a wide range of transfer protocols (see section 2.3), difficulties in using every one of these were encountered. In particular we note the lack of a simple user-level installation of a grid-ftp client as a major stumbling block in its acceptance. Through a combination of features discussed in section 3.4 it was possible to buffer and retry these transfers, in most cases eventually successfully replicating the data to a remote source.

The Transfer Request mechanism, which disconnects the data transfer from the job execution in a manner similar to that done by Condor Stork[16], goes a long way to providing reliable transfers, however from a global view the system shows shortcomings in identifiying fatally failed transfers (i.e. those that will not be retried) and transfers which are outstanding but queued.

## 4.5 Integration with LCG

LCG is required to make possible the storage and processing of the vast quantities of data produced by the LHC experiments. It will bring together hundreds of computing centres around the world and provide an aggregated computing power equivalent to over 70,000 of today's fastest processors. One of the broad objectives for DIRAC is to provide a smooth transition from cluster based to grid based computing for the LHCb experiment and to integrate LHCb computing with the LCG resources.

DIRAC is able to make use of LCG through an implementation of the abstract CE interface. This demonstrates the generality of the DIRAC CE, and ability of DIRAC to bridge both cluster and grid environments. Due to the relatively recent availability of LCG (January 2004 for LCG-2), there are still many operational issues to be worked through. The LCG project assigned two of their team, Roberto Santinelli and Flavia Donno, as LCG Liasons. Their assistance was invaluable in understanding how to work with LCG and tracking down problems.

The task flow is similar to the one described in section 2.1, with the key difference that the job submitted to LCG is a generic DIRAC Agent installation script, rather than a specific DIRAC job. When the LCG job starts, the DIRAC Agent performs an auto-install and configure, then operates in a *run-once* mode where it fetches and executes a single job. This is very similar to the Condor Glide-In concept.

Given the small size of the DIRAC Agent the overhead to do this is minimal, and it provides the advantage that any failure of the LCG job before the DIRAC job is fetched will have no consequence on the DIRAC job pool. The disadvantage is that it adds another layer to the processing chain, and prevents targeted submission of DIRAC jobs to LCG sites. While other approaches are still under investigation and development, this approach has been the most successful and allowed substantial use of the LCG resources with a low rate of failed jobs.

overloaded nodes throws off benchmarks and queue times.

## 5 DIRAC Future Developments

The service oriented architecture of DIRAC proved that the flexibility offered by this approach allows faster development of an integrated distributed system. The *pull* paradigm Agent/Service model has scaled well with a large and varying set of computing resources, therefore we see the future evolution of DIRAC along the lines of the services based architecture proposed by the ARDA working group at CERN[19] and broadly followed by the EGEE middle-ware development group[18]. This should allow DIRAC to be integrated seamlessly into the ARDA compliant third party services, possibly filling functionality gaps, or providing alternative service implementations. The use of two different File Catalogues in the DIRAC system is a good example of leveraging the developments of other projects, and being able to "swap" services, provided they implement a standard interface.

DIRAC currently operates in a trusted environment, and therefore has had only a minimal emphasis on security issues. A more comprehensive strategy is required for managing authentication and authorisation of Agents, Users, Jobs, and Services. It is hoped that a TLS based mechanism can be put in place with encrypted and authenticated XML-RPC calls using some combination of the GridSite project[9], and the Clarens Grid Enabled Web Services Framework, from the CERN CMS project.

While the *pull* model works well for parameter sweep tasks, such as the physics simulations conducted during DC04, it remains to be seen if individual analysis tasks, which are more chaotic by nature, and require good response time guarantees, will operate effectively. A new class of Optimiser is planned which will allocate time-critical jobs to high priority global queues in order that they be run in a timely fashion.

Expanded use of the XMPP instant messaging framework should allow both Jobs and Agents to expose a Service interface, via the XMPP IQ mechanisms. This has great promise for user interactivity, and real-time monitoring and control of Agents and Jobs.

Furthermore, with this Service interface to Agents, a peer-to-peer network of directly interacting Agents is envisioned. This would reduce, and possibly even eventually eliminate, the reliance on the Central Services, as Agents could dynamically load-balance by taking extra jobs from overloaded sites.

## Acknowledgements

We gratefully acknowledge the involvement of the LHCb Collaboration Data Management Group, and the managers of the LHCb production sites. Help from the LCG Experiment Support Group, particularly Roberto Santinelli and Flavia Donno, was invaluable for integrating DIRAC with LCG.

## References

[1] The EU DataGrid Project. `http://www.eu-datagrid.org`.

[2] D. Adams, P. Charpentier, U. Egede, K. Harrison, R. Jones, J. Martyniak, P. Mato, J. Moscicki, A. Soroko, and C. Tan. The ganga user interface for physics analysis on distributed resources. In *Computing in High Energy Physics (CHEP 04)*, September 2004.

[3] AliEN. `http://www.alien.cern.ch`.

[4] BOINC. `http://boinc.berkeley.edu`.

[5] CERN. The LHC Computing Grid Project. `http://lcg.web.cern.ch/LCG/`.

[6] Condor. `http://www.cs.wisc.edu/condor/`.

[7] A. W. Cooke et al. Relational Grid Monitoring Architecture (R-GMA). 2003.

[8] distributed.net. `http://www.distributed.net`.

[9] A. T. Doyle, S. L. Lloyd, and A. McNab. Gridsite, gacl and slashgrid: Giving grid security to web and file applications. In *Proceedings of UK e-Science All Hands Conference 2002*, Sept 2002.

[10] D. G. Feitelson and A. M. Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *12th International Parallel Processing Symposium*, pages 542–546, 1998.

[11] S. Fitzgerald. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, page 181. IEEE Computer Society, 2001.

[12] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG*. Global Grid Forum, 22 June 2002.

[13] Globus and IBM. `http://www.globus.org/wsrf/#announcement`, January 2004.

[14] W. Hoschek. The Web Service Discovery Architecture. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–15. IEEE Computer Society Press, 2002.

[15] IETF. Extensible Messaging and Presence Protocol. `http://www.ietf.org/html.charters/xmpp-charter.html/`.

[16] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *24th IEEE International Conference on Distributed Computing Systems (ICDCS2004), Tokyo, Japan*, March 2004.

[17] P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger. Advanced Replica Management with Reptor. 5th International Conference on Parallel Processing and Applied Mathemetics, Sept 2003.

[18] E. Laure. EGEE Middleware Architecture. In *EDMS 476451*. CERN, June 2004.

[19] LHC. Architectural Roadmap Towards Distributed Analysis - Final Report. Technical report, CERN, November 2003.

[20] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for High Throughput Computing, 1997.

[21] N. Neufeld. The LHCb Online System. *Nuclear Physics Proceedings Supplement*, 120:105–108, 2003.

[22] G. Pape. runit Service Supervision Toolkit. `http://smarden.org/runit/`.

[23] R. Raman, M. Livny, and M. H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, pages 140–, 1998.

[24] G. Rixon. `http://wiki.astrogrid.org/bin/view/Astrogrid/GlobusToolkit3Problems`, December 2003.

[25] SETI@Home. `http://setiathome.ssl.berkeley.edu/`.

[26] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolsky, and M. Swany. A Grid Monitoring Architecture, Jan 2002.

[27] A. Tsaregorodsev et al. DIRAC - Distributed Implementation with Remote Agent Control. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP)*, April 2003.

[28] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, and P. Vanderbilt. Open grid services infrastructure. In *Open Grid Service Infrastructure WG*. Global Grid Forum, July 2003.