DIRAC: A Scalable Lightweight Architecture for High Throughput Computing

V. Garonne, A. Tsaregorodtsev
Centre de Physique des Particules de Marseille
CNRS-IN2P3
163, avenue de Luminy Case 902
13288 Marseille cedex 09, France
garonne@cppm.in2p3.fr, atsareg@in2p3.fr

I. Stokes-Rees
Department of Particle Physics
University of Oxford
Oxford, OX1 3RH, UK
i.stokes-rees@physics.ox.ac.uk

Abstract

DIRAC (Distributed Infrastructure with Remote Agent Control) has been developed by the CERN LHCb physics experiment to facilitate large scale simulation and user analysis tasks spread across both grid and non-grid computing resources. It consists of a small set of distributed stateless Core Services, which are centrally managed, and Agents which are managed by each computing site. DIRAC utilizes concepts from existing distributed computing models to provide a lightweight, robust, and flexible system. This paper will discuss the architecture, performance, and implementation of the DIRAC system which has recently been used for an intensive physics simulation involving more than forty sites, 90 TB of data, and in excess of one thousand 1 GHz processor-years.

Keywords

grid computing, cluster computing, global computing, instant messaging, meta-scheduler, grid scheduling, batch system, job pull

1 Introduction

The LHCb experiment is one of four particle physics experiments currently in development at CERN, the European Particle Physics Laboratory. Once operational, the LHCb detector will produce data at a rate of 4 Gb/s[24], representing observations of collisions of sub-atomic particles. This massive quantity of data then needs to be distributed around the world for the 500 physicists at 100 sites to be able to carry out analysis. Even before this analysis of real physics data can begin a large number of *parameter sweep*[5] simulations are required to verify aspects of the detector design, algorithms, and theory.

While the four Large Hadron Collider (LHC) experiments have coordinated with CERN to produce the LHC Computing Grid (LCG) [6], there is still a requirement within LHCb to manage and track computing tasks, provide a set of common services specific to LHCb, and to be able to make use of hardware and software resources not incorporated into LCG.

This system needed to be quickly and easily deployed across Fourth or more sites, with little effort from local site administrators, either for installation or maintenance. Figure 1 illustrates the computing sites currently contributing to LHCb. Similarly, users needed to be able to access the system from anywhere, with a minimal set of tools. The need to support intense computational load also required the system to be responsive and scalable, supporting over 10,000 simultaneous executing jobs, and 100,000 queued jobs.

This paper discusses the DIRAC architecture (Distributed Infrastructure with Remote Agent Control) which has been developed to meet these requirements and provide a generic, robust grid computing environment. DIRAC incorporates aspects of grid, global, and cluster computing paradigms. It is organized into a *Service Oriented Architecture* (SOA), with several lightweight independent services, following the decomposition found in the CERN ARDA project[22]. As a meta-cluster management system, DIRAC abstracts interfaces to a wide range of heterogeneous computing and storage resources and provides simple APIs and tools for users and developers. For scalability, simplicity, and efficiency a *pull* scheduling model has been implemented which favors *high throughput*, versus *high performance*, job handling[23].

The paper is organized as follows: section 2 presents the background which led to the development of the latest version of DIRAC, while section 3 discusses the DIRAC architecture and main components in detail. Section 4 presents the implementation choices and highlights features important for robustness including the use of instant messaging
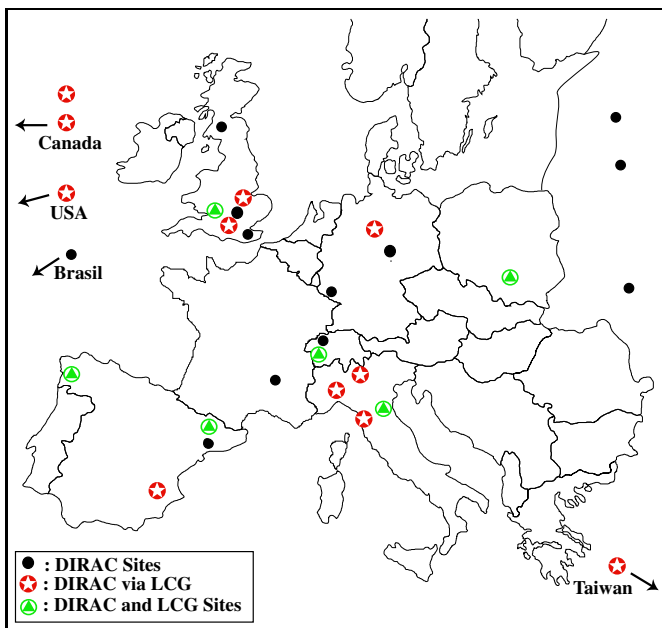
Figure 1. Sites running DIRAC

technology. Section 5 discusses working experience. Section 6 outlines plans for future development, and section 7 finishes with conclusions.

2 Background

Many modern cluster configurations, and all global computing models, focus on *high throughput*, which attempts to maximize the number of jobs completed, on a daily, or longer, basis. This is typical of situations where the supply of computational jobs greatly exceeds the available computing resources, and the jobs are generally not time critical. This strongly favors a *pull* model, where computing resources request jobs from a large job pool. In contrast, a *push* model attempts to centrally optimize the allocation of jobs to resources, and can be overwhelmed by the scale of this problem. A *pull* model only needs to find one job to match one resource, and only when a resource makes a job request.

In 2002 the first version of DIRAC [29] was developed to enable distributed physics simulation, using such a *high throughput pull* based Agent/Service model which could handle the extremely large number of jobs which would be generated. The success of this system validated the broad design principles of *active*, *lightweight* Agents which pulled jobs from *stateless* Services, however the system was only useful for a very limited class of uniform, centrally generated, simulations. The latest version of DIRAC sought to expand on these principles and incorporate the best features of several different computational paradigms. We catego-

rize large scale computational systems into four groups:

Super Computer Specialist machines primarily designed for massive parallel processing with low latency, high bandwidth connections between a large number of co-located processors. Typically used for long runs of a single algorithm by a single user. Examples include the Earth Simulator (NEC), ASCI Q (HP), ASCI White (IBM), and Cray X1.

Cluster Centrally administered high performance commodity hardware, software, and networking designed to provide the most economic computing power for a large number of users at a single site. Resource allocation to users and computation tasks handled through batch queuing software such as PBS[16], Condor[7] or LSF[17].

Grid Federated computing resources which use common interfaces typically to link together computing clusters. The aim is to allow Virtual Organizations (VOs)[14] of users who span institutional boundaries to share computing resources. Examples include the Globus Toolkit [15] used in NorduGrid [11], LCG[6], and EDG[4].

Global Computing *Ad hoc* networks of individual computers, typically desktop machines, which act as slaves for a central job server which usually supports a single parameter sweep application. These operate in a cycle-scavenging mode, using idle CPU power, and pull jobs from the server. Some examples of this include SETI@Home[26], BOINC[3], and distributed.net[9].

Of these, the *Super Computer* category is not applicable to LHCb. In a similar manner to grid computing, DIRAC aims to join disparate computing clusters, however without the overhead of significant grid infrastructure being required at each site. By utilizing aspects used in global computing systems, and designing Agents to run at the user-level, it was possible to streamline the deployment process. DIRAC aims to achieve the same objectives as existing grid and cluster computational systems, which is to present users and developers with a simple, uniform, interface to distributed, heterogeneous computing resources.

3 Architecture

DIRAC can be decomposed into four sections: Services, Agents, Resources, and User Interface, as illustrated in figure 2. 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 central administration. Access to these resources is abstracted via a common interface. Each computing resource is managed autonomously by an Agent, which is configured with details of the site and site usage policy by a local administrator. The Agent runs on the remote site, and manages the resources there, job monitoring, and job submission. The User Interface allows access to the Central Services, for control, retrieval, and monitoring of jobs and files.
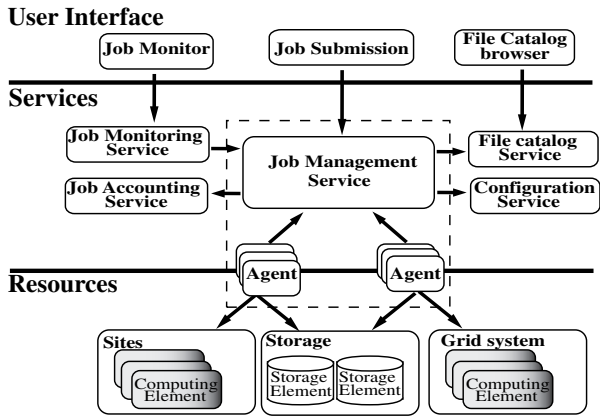

Figure 2. Architecture overview

Jobs are created by users who interact with the system via the Client components. All jobs are specified using the ClassAd language, as are resource descriptions. These are passed from the Client to the Job Management Services (JMS), allocated to the Agents on-demand, submitted to Computing Elements (CE), executed on a Worker Node (WN), returned back to the JMS and finally retrieved by the user.

An idea borrowed from global computing systems is the cycle-stealing paradigm, where jobs are only run when resources are not in use by the local users. This is similar to common batch system backfill algorithms[27], except that it operates only when there are completely free slots, rather than fitting in short jobs ahead of future job reservations.

DIRAC has started to explore potentials for distributed computing from instant messaging systems. High public demand for such systems has led to highly optimized packages which utilize well defined standards, and are proven to support thousands to tens-of-thousands of simultaneous users. While these have primarily been utilized for person-to-person communication, it is clear that machine-to-machine and person-to-machine applications are possible.

The following sub-sections discuss in greater detail the key aspects of DIRAC.

3.1   Job Management Services

The JMS consists of the following services, as illustrated in figure 3:

Job Receiver Accepts job submissions from clients, registers them to the Job Database and notifies the Optimizer Service

Job Database Contains all the information about the job parameters and dynamic job state

Optimizers Sorts jobs into global job queues and continuously reshuffles them depending on queue states, job load, and resources availability

Matchmaker Allocates jobs to resources by selecting from the global job queues and using Condor Matchmaking
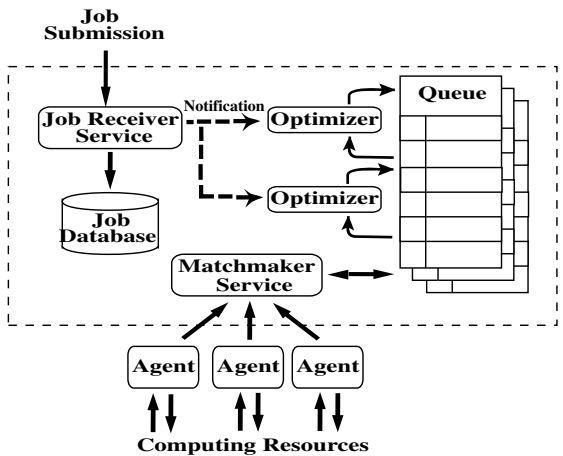

Figure 3. Job Management Services

Most of the work in the JMS is performed by the Optimizers. They prioritize jobs in queues using a range of techniques, and utilizing information from job parameters, resource status, file locations, and system state. As a result of this, jobs can be assigned to a particular computing resource which meets the job requirements, such as replicas of input data files.

Optimizers are designed to be customizable, and simply need to implement a standard interface for interacting with the queues they manage. Multiple Optimizers can exist in the system at the same time, and can be dynamically inserted, removed, started, and stopped at run-time. This allows new algorithms or heuristics for job prioritization to be rapidly inserted into the system.

There are three phases in a typical *push* grid scheduling system:

1. Scheduler collects resource status for entire grid

2. Scheduler selects job allocation to resources

3. Scheduler submits jobs to resources

For phase one, all the information concerning the system needs to be made available at one place at one time. In a large, federated grid environment, this is often impractical, and information will often be unavailable, incorrect, or out of date. In the second phase, the choice of the best pairs of jobs and resources is an NP-complete problem and the size of this problem increases with the number of jobs and resources. This approach is often centralized, as in EDG[4], and does not scale well.

In contrast, the DIRAC *pull* strategy has the following phases:

1. Agent detects free computing resources

2. Agent requests job from Matchmaker

3. Matchmaker checks queues for appropriate match

4. Matchmaker returns best matching job to Agent

The previously difficult task of determining where free computing resources exist is now distributed to the local Agents (see section 3.2) which have an up to date view of the local system state. In phase 3, Condor Matchmaking is used. [23] The Matchmaker 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. This is an *O(n)* operation, with the worst case being all *n* jobs queued in the system are checked once against the resource characteristics defined in the job request.

Typically it is found that job requirements do not vary significantly within a system, therefore it is likely that a match will be made early on (that is, in less than *n* comparisons), even if jobs are randomly distributed among the queues. Both long matching time and the risk of job starvation can be avoided through the use of an appropriate Optimizer 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[12], this allows a mixture of standard and custom scheduling algorithms.

3.2 Agent

The Agent is deployed on a computing resource and interacts directly with it. This Agent is completely 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 includes 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.

The most important of these Agent modules is the Job Request module, which monitors the state of the local computing resource and fetches jobs from the Matchmaker Service when it detects free slots. Upon job submission to the local batch system, it stores job parameters in a local database. This allows it to verify the status of the jobs and spot job failures. This information can also been checked by a lightweight tunneling service interface in the agent, provides by instant messaging technology (see section 4.4). This interface also allows users to interact directly with the agent.

3.3 Computing Element

The Computing Element is the abstracted view of a computing resource, providing a standard API for job execution and monitoring. Using this, an Agent can easily deal with heterogeneous computing resources. A Computing Element is modeled as a Head Node which manages a cluster of Worker Nodes. Such a system is assumed to have its own local scheduler and local queues.

At present, DIRAC provides CE interfaces to LSF, PBS, NQS, BQS, Sun Grid Engine, Condor, Globus, LCG, and stand-alone systems. Each implementation deals with translating the DIRAC job requirements to locally understood settings.

3.4 Data Management System

There is a great deal of complexity in the DIRAC Data Management System which allows fault tolerant transfers, replication, registration, and meta-data access of data between computing sites and long term storage sites. The description here provides a brief outline of the three main components of this system.

Storage Element (SE) This is defined entirely by a host, a protocol, and a path. This definition is stored in the Configuration Service (see section 3.5), and can be used by any Agent or Job, either for retrieving files or uploading generated files/results. 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.[20]

File Catalogs DIRAC defines a simple interface for locating physical files from aliases and universal file identifiers. This has made it possible to utilize two independent File Catalogs, one from the already existing LHCb File Database, and another using the AliEn File Catalog from the Alice project[1]. Catalogs can be used interchangeably. 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.

Reliable Data Transfer Service 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 maximize the possibility of successfully transferring the data.

3.5 Configuration Service

It is a common challenge of distributed systems and Service Oriented Architectures to share information across the system. There is a network of Services, each of which need to configure themselves, and find out configuration information about the other Services. Users then need to know how to access those Services and work with them. When considering a Configuration and Information Service for DIRAC, it was felt that the existing mechanisms, such as UDDI[18], MDS[13], and R-GMA[28, 8], were powerful, yet complex, and required significant infrastructure to utilize.

In keeping with the principles of simplicity and lightweight implementation, a network-enabled categorized name/value pair system was implemented. Components which use the Configuration Service do so via a *Local Configuration Service* (LCS). This can get all information from a local file, from a remote service, or via a combination of the two.

It is possible to cache remote information and have alternate remote services, in the event one Configuration Service is not available. The semantics dictate that local values are always taken in preference to remote values. If a value is not found locally, the LCS will round-robin through the alternate Configuration Service sources.

4 Implementation Details

The current implementation has been written largely in Python, due to the rich set of library modules available, its object oriented nature, and the ability rapidly prototype design ideas. All Client/Service and Agent/Service communication is done via XML-RPC calls, which are lightweight and fast. Furthermore, instantiating and exposing the API of a Service as a multi-threaded XML-RPC server in Python is extremely straight forward and robust. For all Service and Job state persistence, a MySQL database is used.

By keeping the implementation in Python, Clients and Agents only require a Python interpreter for installation. These components which are distributed to the users and computing centers are also very small — less than one megabyte compressed — which further facilitates a rapid installation or update of the software. Software required for jobs is installed in a *paratrooper* approach, which is to say that each job installs all software it requires, if it is not already available. This software is cached and made available for future jobs run by the same Agent.

The following expands on key implementation decisions which have contributed to the successful operation of DIRAC, such as failure tolerance, a robust and simple configuration service, and the use of instant messaging.

4.1 Configuration Service Redundancy

The Configuration Service is the backbone for coordinated access to information regarding the various DIRAC Services. Every component within DIRAC utilizes a Local Configuration Service. The duality of the file based information and the XML-RPC remote API allows an LCS to transparently use one or the other to acquire necessary information. Three strategies have been implemented to make this system robust:

Duplication The central Configuration Service duplicates its information to a secondary server which hosts a backup service.

Fail-Over The Local Configuration Service will fail-over once to the backup service if it fails to contact the primary service, and fail-over a second time to a file containing a saved (but possibly out of date) copy of the Configuration Service data. This second fail-over is essential in the case of network outages so the job may, for a time, proceed without contacting any remote services.

Caching Value pairs and sections are cached locally on the first request, speeding up subsequent operations, and reducing the load on the Configuration Service.

While both the caching and file-based fail-over have the risk of utilizing incorrect, out of date information, this was considered preferable to outright job failure due to inability to access the service.

4.2 Service and Agent Watchdogs

All Services and Agents are run under the *runit* daemon watchdog[25]. 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.

4.3   Job Watchdog and Wrapper

For each job, a *wrapper* script prepares the execution environment, downloads the necessary data and reports to the Job Monitoring Service the Worker Node parameters. It then spawns a watchdog process. This process periodically checks the state of the job and sends a *heart beat* signal to the Monitoring Service. It can also provide a control channel to the job via an instant messaging interface (see section 4.4). At the end of the job, the watchdog process reports the job execution information, for example CPU time and memory consumed, to the Monitoring Service. Finally, it catches failed jobs and reports them appropriately.

4.4   Instant Messaging in Grid Computing

In order to provide asynchronous reliable messaging, DIRAC has incorporated an instant messaging protocol into all components of the system. While the DIRAC Services expose their APIs via XML-RPC, due to the simplicity, maturity, and robustness of this protocol, there is a need to expose a monitoring and control channel to the transient Agents and Jobs. 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 should be used.

The Extensible Messaging and Presence Protocol (XMPP), now an IETF Internet Draft[19], is used in all four areas of DIRAC: User Interface, Jobs, Agents, and Services. XMPP provides standard instant messaging functionality, such as chat, group chat, broadcast message, and one-to-one messaging. Furthermore, an RPC-like mechanism exists called Information/Query, (IQ) which can be used to expose an API of any XMPP entity. Finally, the *roster* mechanism facilitates automatic, real-time monitoring of XMPP entities via their *presence*.

The Services use XMPP in certain places where fault tolerant, asynchronous messaging is important. For example, the Job Receiver Service uses XMPP to notify the Optimizer Service when it receives a new job. When the Optimizer gets this message, it will then sort the new job into the appropriate queues, The IQ functionality has the poten-tial 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.

5   Working Experience

The DIRAC system is being used for the LHCb Data Challenge 2004 (DC04), held from May to July 2004. The goal of DC04 is to validate the LHCb distributed computing model based on the combined use of the LCG and conventional computing centers. A large number of simulation jobs will be run, producing terabytes of data which will need to be redistributed to a network of computing centers for both organized (i.e. planned and predictable) and chaotic analysis of the results.

The system has operated smoothly with a sustained level of 2000 running jobs, and 600 gigabytes of data generated and replicated daily. Figure 1 shows the participating sites, and figure 4 shows a snapshot of the running job distribution. Using the *runsv* daemon control tools, discussed in section 4.2, once a site has installed DIRAC the Agents run autonomously, and restart after failures or reboots. The monitoring system allows the performance of various sites and the behavior of Agents and Jobs to be monitored by anyone, which in practice falls to a central team who can alert site administrators if problems are detected.
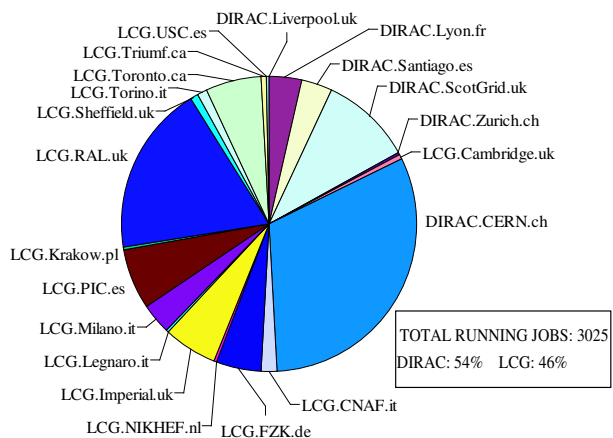

Figure 4. Snapshot of site distribution for running jobs during DC04

At the time of writing the DIRAC system is managing tasks running directly at 20 computing centers, and at another 20 sites via the LCG network. These 40 sites provide a total of more than 3000 worker nodes. At this time, with 5000 jobs in the global queues, the Matchmaker responds to Agent job requests in 0.4 seconds, on average. More than

40,000 jobs have been completed in the month of May with an average duration of 23 hours. In terms of storage capacity, during DC04 the system has produced, stored and transfered 20 terabytes of data. Each job produces about 400 megabytes, which is immediately replicated to several sites for redundancy and to facilitate later data analysis.

5.1 Interoperability with LCG

DIRAC is able to make use of LCG by wrapping it in the standard DIRAC batch system interface. The task flow is similar to the one described in section 3, 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.

5.2 Challenges

The two greatest challenges with DIRAC have been monitoring job status for a large number of jobs, and managing data transfers. With tens of thousands of active jobs, spread across 40 or more sites, there are inevitable problems with network failures, power outages, incorrect configuration, and software crashes. In the early release of DIRAC this led to a build up of stalled jobs which claimed to be running but never completed. Tracking these jobs down and resubmitting them, as well as managing failed jobs and confirming that "successful" jobs really had completed all their steps, required the implementation of cross-checks and additional job monitoring, such as the *heart beat* mechanism, mentioned above.

Managing data transfers for such large quantities of data also proved challenging. In many cases network connections became backlogged, hung, or aborted transfers part way through. These problems occurred on both the client (sending) and server (receiving) side. Initially the bbftp protocol appeared to be the most reliable, but this shifted in favor of gridftp[2], although gridftp was much more difficult to install and use. The queued Transfer Request system with independent DIRAC Agents dedicated just to manag-

ing transfers (via the TransferAgent module) proved invaluable in providing reliable data transfers, possibly time delayed from the end of the job completion by several days.

6 Future Plans

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[22] and broadly followed by the EGEE middle-ware development group[21]. 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 Catalogs 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 authorization 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[10], 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 Optimizer 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.

7 Conclusions

The latest version of DIRAC has proven to be robust, and easy to use and deploy. The *pull* paradigm has meant large job queues and large numbers of running jobs do not degrade system performance, and job allocation to resources takes under a second per job. The service oriented architecture and Agent/Service model has allowed flexible inclusion of new modules and rapid development of the entire DIRAC framework.

As the "go-live" date for the LHC approaches, greater integration with the LCG and ARDA projects is planned. It is expected that DIRAC will form the basis of the distributed computing infrastructure for the LHCb experiment, and be able to utilize Services developed by ARDA and the underlying LCG network.

Acknowledgments

References

[1] AliEN. http://www.alien.cern.ch.

[2] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke. Data Management and Transfer in High Performance Computational Grid Environments. *Parallel Computing Journal*, 28 (5):749–771, May 2002.

[3] BOINC. http://boinc.berkeley.edu.

[4] G. Cancio, S. M. Fisher, T. Folkes, F. Giacomini, W. Hoschek, D. Kelsey, and B. L. Tierney. The DataGrid Architecture Version 2. In *EDMS 439938*. CERN, Feb 2004.

[5] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop*, pages 349–363, 2000.

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

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

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

[9] distributed.net. http://www.distributed.net.

[10] 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.

[11] P. Eerola et al. The NorduGrid Architecture and Tools. In *Proceedings of Computing for High Energy Physics 2003*, 2003.

[12] 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.

[13] 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.

[14] I. Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[15] Globus. http://www.globus.org.

[16] Henderson and H. Tweten. Portable Batch System : External reference specification. Technical report, NASA Ames Research Center, December 1996.

[17] R. Henderson. LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems. In *Proceedings of the Workshop on Cluster Computing*, December 1992.

[18] 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.

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

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

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

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

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

[24] N. Neufeld. The lhcb online system. *Nuclear Physics Proceedings Supplement*, 120:105–108, 2003.

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

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

[27] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective Reservation Strategies for Backfill Job Scheduling. In *Proceedings of 8th Workshop on Job Scheduling Strategies for Parallel Processing*, July 2002.

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

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