DIRAC: Workload Management System

V. Garonne, A. Tsaregorodtsev
Centre de Physique des Particules de Marseille, Marseille, France
I. Stokes-Rees [*],
University of Oxford, Oxford, UK

Abstract

The Workload Management System is the core component of the DIRAC distributed MC production and analysis grid environment of the CERN LHCb experiment. This paper discusses the architecture, implementation and performance of this system. The WMS is a community scheduler, realizing a *pull* paradigm, particulary for the high troughput computing context. It has recently been used for an intensive physics simulation production involving more than 60 sites, 65 TB of data, and over 1000-GHz processor-years.

INTRODUCTION

LHCb is one of the 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[8]. This data will then be distributed around the world more than 100 physicists at 40 sites for analysis. Before this analysis of real physics data can begin simulations are required to verify all aspects of the detector design, algorithms, and theory.

The LHCb Data Challenge (DC04) held from May-September 2004 had three goals: to confirm the LHCb computing model; to verify LHCb physics software; and to generate simulation data for analysis. The ultimate goal of this exercice was to produce the equivalent of 10% of the real data accumulated in one year, which represents 60 TB. To achieve this goal, LHCb had to make efficient use of all available computing resources, which are a combination of LCG(LHC Computing Grid Project) and conventional computing centres. In total, this represented more than 10.000 CPUs, sometimes shared with other organizations, which had to execute more than 250K jobs. Allocation of this number of jobs to the resources needed an efficient worlkload management system (WMS).

This paper will first describe the background of the DIRAC WMS developement and its architecture. The following section presents the scheduling algorithm and the match-making procedure. We will also discuss the recent results obtained during DC04.

BACKGROUND

The DIRAC project was initially started as a system for distributed MC production system for the LHCb experiment. The WMS for a production must focus on *high throughput*, which implies maximization of 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.

An efficient WMS should saturate all the resources available to LHCb independently of their nature. These resources are often shared with other communities. Therefore, a community scheduler approach dedicated to one virtual organization is the most suitable in this setting.

The latest version of DIRAC [11] is organized into a *Service Oriented Architecture* (SOA), with a number independent services, following the decomposition found in the CERN/ARDA blueprint document [6]. An important requirement for the WMS was its design and deployment as a set of independent services, fitting well the DIRAC infrastructure. This implies service implementations which are lightweight, scalable and flexible, matching the requirements and design of the overall DIRAC system.

WMS ARCHITECTURE

The WMS is divided in two distinct parts. First, the Job Managenent Service (JMS) which is itself composed of a set of central services and, second, a number of agents which are deployed on the remote computing resources.

Job Management Service

The Workload Management Service consists of the following components, as illustrated in Figure 1:

Job Receiver accepts job submissions from clients, registers them to the Job Database and notifies one of the Optimizers.

Job Database contains all the information about the job parameters and the dynamic job state.

Optimizers sort jobs into queues and continuously reshuffle them depending on queue states, job load, and resources availability.

[*] also Marie Curie Fellow at CPPM, Marseille, France

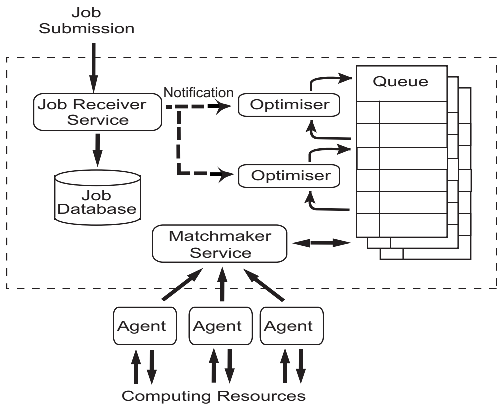

Figure 1: *Job Management Services*

Matchmaker allocates jobs to resources by selecting from the global job queues. Uses Condor Matchmaking to find an appropriate job to a particular agent request.

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 added to the system.

Agent

Agents are light daemons running close to a computing resource and forming the distributed part of the WMS. The design consists of a set of pluggable Agent modules. The modules are executed in sequence in a continuous loop and can be easily added (even dynamicaly) enhancing the agent functionality.

The most important of these Agent modules is the Job Request module. It monitors the state of the local computing resource and fetches jobs from the Matchmaker Service when it detects that the resource is ready to take the workload. Thus, it operates in a *cycle-scavenging* mode at the cluster level, where it only requests and executes jobs when the local resources are under-utilised. This idea comes from global computing models, such as SETI@Home, BOINC, and distributed.net [9, 1, 4], which perform cycle-scavenging on home PCs.

After the reception of a job from the Matchmaker, the Agent checks that the necessary software environment is available on the site and installs any missing software packages if necessary.

Upon job submission to the local batch system, it stores job parameters in a local database. This allows it to compare the status of the jobs as reported by the WMS and by the local batch system. In many cases this check helps spotting job failures which failed to communicate their status to the Job Monitoring Service.

One of the Agent modules is acting as an Instant Messaging Client. This allows users to communicate with the Agent to get its status information as well as the status of the local jobs [10].

Agent deployment

The Agent is easily deployable on a site requiring only the Python interpreter and outbound Internet connection in order to contact the DIRAC Services.

Two distinct ways of the Agent deployment were used. In a static approach, the Agent is installed close to a computing resource and interacts directly with it. Usually, it is deployed on a site gateway and is completely under the control of the local site administrators.

Another, dynamic, approach, is illustrated in the Figure 2. This way of the Agent deployment was used to integrate the DIRAC WMS with the LCG provided resources.

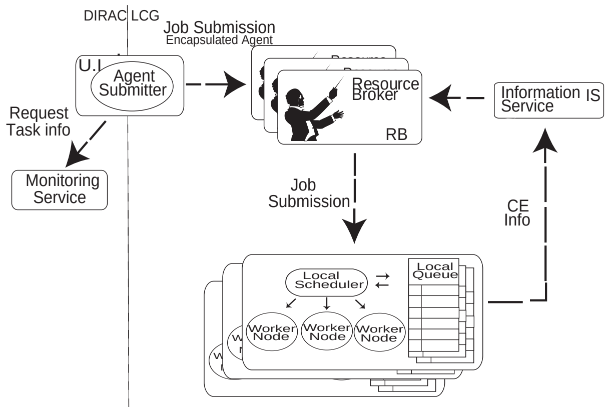

Figure 2: *WMS integration with LCG*

A special process called Submitter is running on an LCG User Interface node. It contacts the DIRAC Job Monitoring service to find out if there are jobs in a waiting state. If there are, it then submits the appropriate number of jobs which encapsulate agents to the LCG Resource Broker. In this way the scheduling system of LCG is just used for deployment of DIRAC Agents on the Worker Nodes.

Once the job arrives to the Worker Node, it launches the DIRAC Agent, as illustrated in Figure 3. It checks the Worker Node capacity and environment and creates the JDL resource description accordingly. In particular, the max CPU time available on the WN is estimated with a simple benchmark test in order to estimate the maximum size LHCb job thaht can be computed given the queue time limit. After that the Agent queries a job from the MatchMaker and the rest of the job processing is performed as on any other DIRAC site.

In fact, this approach is a kind of resource reservation which allows DIRAC jobs to be run on any kind of computing grid.

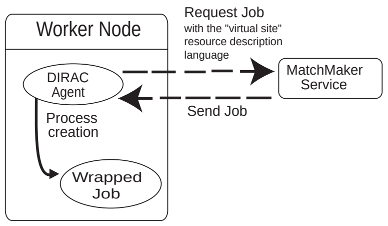

Figure 3: *Agent on a Worker Node*

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(via fork). Each implementation deals with translating the DIRAC job requirements to locally understood job submission command.

Criterion of availibility

To know if a resource is available we use a criterion of availability, which depends on the nature of the resource itself. For example, the following criterion of availability can be defined for a batch system by queue:

$$\frac{Total\ Queueing\ Jobs}{TotalCPUs} < \varepsilon \tag{1}$$

For example we set $\varepsilon = 0.3$ by default. This criterion implies that jobs in the waiting state scheduled on a computing resource should not exceed 30% of the total number of CPUs. This simple criterion allows balancing the number of running and queued jobs on a site over a wide range of capacities.

Job Watchdog and Wrapper

For each job, a Wrapper script prepares the execution environment, as illustrated in Figure 4, 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 ([12]) . In addition, it can provide a connection for interaction with the owner of the job by means of the Instant Messaging protocol [10]. 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.

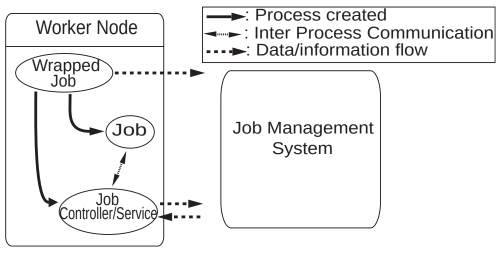

Figure 4: *Job Wrapper*

PULL PARADIGM VS *PUSH* PARADIGM

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[2], 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 which have an up to date view of the local system state. In phase 3, Condor Matchmaking is used. [7] The Matchmaker only compares one-on-one requirements, using 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, so rather than compare a request against all *n* jobs, the jobs are sorted into queues based on common requirements. The matching time is significantly reduced. 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 [5], this allows a mixture of standard and custom scheduling algorithms.

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.

PERFORMANCE

At the time of writing the DIRAC([3]) system is managing tasks running directly at 20 computing centers, and at another 40 sites via the LCG network. These 60 sites provide a total of more than 5000 worker nodes.

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.

More than 200,000 jobs have been completed in four months with an average duration of 21 hours. In terms of data, during DC04 the system has produced, stored and transfered 60 terabytes of data. Each job produces 400 megabytes, which is immediately replicated to several sites for redundancy and to facilitate later data analysis.

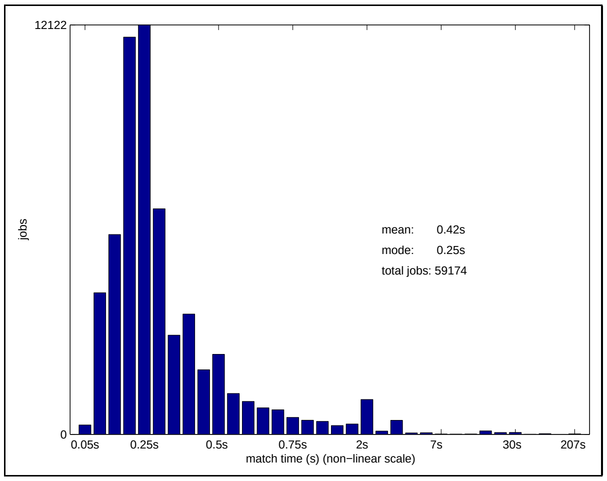

Figure 5: *Match time distribution for 59,174 jobs during DC04*

chapter

FUTURE DEVELOPMENT

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.

CONCLUSION

The worlkload management system of DIRAC has proven to be robust, scallable and easy to 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.

ACKNOWLEDGEMENTS

We gratefully acknowledge the involvement of the LHCb Collaboration Data Management Group, and the managers of the LHCb production sites.

REFERENCES

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

[2] 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.

[3] J. Closier et al. Results of the LHCb experiment Data Challenge 2004. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP)*, November 2004.

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

[5] 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.

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

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

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

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

[10] Stokes-Rees, I. and Tsaregorodsev, A. and Garonne, V. Grid Information and Monitoring System using XML-RPC and Instant Messaging for DIRAC . In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP)*, September 2004.

[11] A. Tsaregorodsev et al. DIRAC - The Distributed MC Production and Analysis for LHCb. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP)*, November 2004.

[12] V. Garonne and R.Graciani-Diaz and J.Saborid and M.Sanchez and R.Vizcaya-Carrillo. A Lightweight Monitoring System and Accounting System for LHCb DC04 Production. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP)*, November 2004.