Performance Testing of DIRAC v2.1 (December 2003)

Ian Stokes-Rees

January 13, 2004

1 Summary

The DIRAC v2.1 system, based on XML-RPC, bbftp, and written in Python, performs well even under reasonable loads. No significant functionality problems were discovered, although that was not the objective of these tests. Some sample results include 4000 job status queries executed in 33 seconds, and 100 job submissions including a 500k file upload completed in 2 minutes 16 seconds. Performance seems to be maintained even with 8000+ jobs stored in the Job DB. The only problem appeared to occur when large numbers (¿1500) of job submissions were initiated simultaneously – we believe the delay here is due to Jabber messaging queues being filled.

2 Introduction

This report summarises tests of the DIRAC physics production and analysis system. The DIRAC system consists of the following key components:

Client – where jobs originally come from and where results are returned to

Job Receiver – accepts job submissions from the Client and registers them to the Job DB

Job DB – stores information pertaining to each job

Matchmaker – decides what jobs to give to a particular computing resource

Agent – which monitors some computing resource (cluster or single processor), and fetches jobs from the Matchmaker

Monitoring DB – contains monitoring information pertaining to individual jobs and to computing resources

The set of tests performed were focusing on performance evaluation as opposed to functionality testing. No significant problems were found with the functionality, however. The performance tests aimed to evaluate the behaviour of Agents and Clients accessing the DIRAC central services.

2.1 Types of Tests

Each set of tests aimed to measure: single invocation response time; "average" loading response time and performance; and "saturation" loading response time. The "average" loading consisted of between 40-100 invocations, and the "saturation" loading between 500-4000 invocations. The repeated execution was handled in two ways: serially, within the code; and, in parallel, either using "fork" or multiple threads. The multi-threaded version was required because with hundreds or thousands of parallel operations using "fork" the memory resources were exhausted. The particular technique is noted in the test description.

The tests can be broken down into a few general categories, based on the common operations for the DIRAC system: job submission, job fetching, monitoring updates, and monitoring queries. Job execution was not tested, as the performance of individual compute nodes cannot be controlled by the DIRAC system.

2.2 Note on Testing Technique

Due to the limited number of suitably configured systems, it was not possible to have multiple Clients or Agents on independent systems accessing the DIRAC central services. We hope to perform tests of this sort in the future, however we belive the testing approach described here provides results which can be generalised to the expected performance of the final system. We were using at most three systems: one hosting the DIRAC central services, one acting as a job submission (or retrieval) Client, and one acting as an Agent and fetching/returning jobs. Simultaneous parallel invocation of Clients or Agents were used to simulate the effect of multipe systems concurrently accessing the central services.

Note also that while timing results lists clock time, user time, and system time, in most cases only the "wall clock" time is relevant. The user and system time only refers to the client side operations, and not the server side. Distinguishing between network and file system latencies, server side load, and client side load would have required significantly more effort to instrument the various services. As a result, the "wall clock" time captures all of these together. The user and system times have been included, but these only indicate user and system time for the client or agent.

3 Testing Results

3.1 Job Submission

The following details results for three different types of job submission: one which simply executes an "echo", one which includes a small (1k) file, and the third which includes a 500k jpeg image file (therefore not very compressible). The two types which include input sandboxes use bbftp to transfer the file to the DIRAC Storage Element.

Description	Count	Clock	User	System	Method
echo	1	1.2s	0.074s	0.043s	fork
echo	100	4.0s	0.68s	0.21s	fork
echo	1500	44s	9.7s	2.9s	fork
echo	4000	7m30s	25s	7s	fork
1k file	1	6.6s	0.20s	0.06s	fork
1k file	100	2m21s	13s	4.9s	fork
1k file	100	1m29s	2.4s	3.4s	thread
1k file	500	7m32s	11.9s	35.1s	thread
500k file	1	7.5s	0.23s	0.1s	thread
500k file	100	2m16s	2.5s	3.8s	thread

Table 1: Job submission results

When executed 4000 times, the "echo" job submission caused the server to essentially lock up after approximately 1800 jobs had been received. These first 1800 jobs were submitted in under a minuet. It appeared that the problem due to the Jabber messaging queues being backlogged. After a significant delay (30s-1m), the job submissions resumed, and the *client* finished after 7m30s, however the *server* continued to produce Jabber messages for a further 7 or 8 minutes, hence the conclusion that it was a Jabber messaging queue backlog. The total time to clear the 4000 messages through Jabber was approximately 15 minutes. As can be noted, with only 1500 jobs submitted the client completed very quickly and the server was able to keep up without a Jabber backlog.

When transfering 500 1k files approximately 50 bbftp transfers timed out around the middle of the transfer band. Given the small size of the file, we assume that this was due to the bbftp server saturating, although this was not verified. At the time, the server machine was reported a load average of 40 and was responding very slowly. It appears that the client side transfer mechanism successfully re-attempted the transfers such that they eventually succeeded. Notice that the 500 file transfer time was approximately 5 times the length of the 100 file transfer time.

Many simultaneous "large" files also transfer quickly. While 500k is, in fact, relatively small, it does establish a test where 100 active file transfers are taking place concurrently, and that they can all complete successfully and in good time.

3.2 Job Fetching

The environment used for testing job execution simply fetches and discards all jobs it gets in order to simulate numerous agents simultaneously accessing the job server. The objective of the job fetch testing is not to see how long it takes for a compute node to execute a job, but how quickly the job matcher can server jobs to nodes which are requesting them.

It is important that the server be able to deal out a large number of jobs rapidly. This will occur whenever the server is first brought on-line or restarted for some reason. It is possible that all active agents will query the server for jobs, so

Description	Count	Clock	User	System	Method
getJob	1	1.1s	0.15s	0.04s	thread
getJob	100	15.0s	0.59s	0.36s	thread
getJob	200	17.0s	1.6s	1.4s	thread
getJob	300	27.1s	2.0s	1.9s	thread
getJob	400	1m50s	2.3s	2.1s	thread
getJob	500(*)	3m30s	2.8s	3.1s	thread

Table 2: Job fetching results (* 34 threads timed out, between threads 344 and 475)

possibly thousands of jobs may need to be dispatched very rapidly.

It appears that timeout issues can occur if more than 10 job fetches per second are requested. In a test to repeatedly execute 200 job fetches, the delays were: 13s, 24s, 52s, 3m32, however the last execution had 37 threads timeout, and the last successful fetch occurred after 2m50s. It is not clear what is causing the increase in time for subsequent fetches. It may be that the DB has background tasks to perform, or that the DIRAC service is responding to re-ordering the queue lists and changes in the DB. It appears that all Jabber communication has been completed after each run, so we do not expect Jabber message backlogs to be the source of the delays. After waiting a few minutes the job match rate returns to approximately 10/second.

A final test was done sequentially fetching 100 jobs at a time. The completion rate for this was as follows: 8s, 23s, 46s, 8s, 37s, 1m52s, 7s. The most interesting point is that for the 1m52s run there were only a few jobs left in the DB to match. At least 90 threads got the response "No work available". The following 7s run also had no jobs available. I cannot provide an explanation for this behaviour.

3.3 Monitoring Updates

Description	Count	Clock	User	System	Method
setJobStatus	1	1.35s	0.06s	0.01s	thread
setJobStatus	100	4.0s	0.28s	0.18s	thread
setJobStatus	500	12.2s	1.5s	0.8s	thread
setJobStatus	1000	14s	3.1s	1.54s	thread

Table 3: Monitoring updates

More than 1000 concurrent setJobStatus operations causes faults which I believe are due to Python threading issues (more than 1000 threads), rather than due to DIRAC limitations.

3.4 Monitoring Queries

Monitoring queries showd very high sustained throughput, in excess of 100 queries per second.

Description	Count	Clock	User	System	Method
getJobStatus	1	0.8s	0.064s	0.03s	fork
getJobStatus	100	1.6s	0.23s	0.12s	fork
getJobStatus	4000	33s	7.0s	2.73s	fork

Table 4: Monitoring queries

4 Further Testing

Further testing should be done to make use of multiple active clients and agents on independent systems simulatneously accessing and loading the DIRAC central services. Also, more testing should be done with large file transfers, and, to date, no testing has been done here on job fetching. These tests are required to evaluate the performance of handling input sandboxes.

5 Testing Setup

This section is only for historic purposes. It documents how the testing system was setup and how the tests were run, in order that they may be reproduced in the future.

1. Currently the definitive CVS repository for DIRACv2 is `:ext:ijstokes@isscvs.cern.ch:/local/reps/di` (replace `ijstokes` with your AFS username, and set `CVS_RSH=ssh`). Set `CVSROOT` to this value and then checkout the `WMSpy` module. If you want the version used for these tests, add the option `-D 2004-01-01`.

```
declare -x CVSROOT=:ext:ijstokes@isscvs.cern.ch:/local/reps/dirac
declare -x CVS_RSH=ssh
cvs co -D 2004-01-01 WMSpy
```

2. Read the file WMSpy/README. This details how to setup the server, client, and agent, although it is very brief. Full install documentation is not yet available.

3. Setup DIRAC on two systems. One will act as the server, and one will act as both the Client and Agent. It is necessary to set the appropriate PYTHONPATH on both, and may be necessary to set BBFTP_EXEC (location of bbftp) and LHCBBBFTPRC (location of file containing bbftp username/password pair).

4. Start the server(s) running by executing WMSpy/scripts/SystemControler.

5. Relevant client, agent, and monitoring scripts can be found at:
   - Client/scripts/wms_jobs_submit JDL-file
   - Agent/scripts/ForkAgent CE-JDL-file
   - JMS/scripts/wms_job_status JobID

- JMS/scripts/wms_job_setstatus RangeToSet

Variations which are suffixed "X" or "Y" generally repeat the operation a specified number of times, using either "fork" or threads.