Journal of Physics: Conference Series **513** (2014) 042048 doi:10.1088/1742-6596/513/4/042048

DDM Workload Emulation

R Vigne[1,2], E Schikuta[1], V Garonne[2], G Stewart[2], M Barisits[2], T Beermann[2], M Lassnig[2], C Serfon[2], L Goossens[2] and A Nairz[2] on behalf of the ATLAS Collaboration

[1] University of Vienna, Austria
[2] CERN, Geneva, Switzerland

E-mail: ralph.vigne@cern.ch

Abstract. Rucio is the successor of the current Don Quijote 2 (DQ2) system for the distributed data management (DDM) system of the ATLAS experiment. The reasons for replacing DQ2 are manifold, but besides high maintenance costs and architectural limitations, scalability concerns are on top of the list. Current expectations are that the amount of data will be three to four times as it is today by the end of 2014. Further is the availability of more powerful computing resources pushing additional pressure on the DDM system as it increases the demands on data provisioning. Although DQ2 is capable of handling the current workload, it is already at its limits. To ensure that Rucio will be up to the expected workload, a way to emulate it is needed. To do so, first the current workload, observed in DQ2, must be understood in order to scale it up to future expectations. The paper discusses how selected core concepts are applied to the workload of the experiment and how knowledge about the current workload is derived from various sources (e.g. analysing the central file catalogue logs). Finally a description of the implemented emulation framework, used for stress-testing Rucio, is given.

1. Introduction

Data management and provisioning in the ATLAS experiment [1] has been done by Don Quijote 2 (DQ2) [2] since 2006. Since that time not only the amount of data has increased but also the performance of applications depending on distributed data provision (e.g. the ATLAS Production and Distributed Analysis System (PanDA) [3]). DQ2 currently manages about 150 petabytes of physics data which are spread over more than 120 sites all around the globe and provides access to this data for around 800 active users. Although DQ2 is able to manage today's workload, it is almost at its limits. Various adaptions and changed requirements over the last years did compromise its basic design, resulting in reduced scalability like no 'native' file-level transfers, no meta-data data discovery, ...

As today's predictions expect the workload of DDM to be three to four times of today's by the end of 2014, DDM becomes a pressing issue to address within the experiment. To avoid DDM becoming the bottleneck for future applications, Rucio [4] was implemented. Its design respects the experience gained over the last 7 years with DQ2 (e.g. user behaviour, application requirements, ...) with a strong focus on scalability. To verify its scalability, a workload emulator was developed to validate its performance at multiples of today's load.

In Section 2 we discuss how knowledge about today's workload in DQ2 was derived and how it was utilised to model the emulated workload. Section 3 illustrates how the used emulation

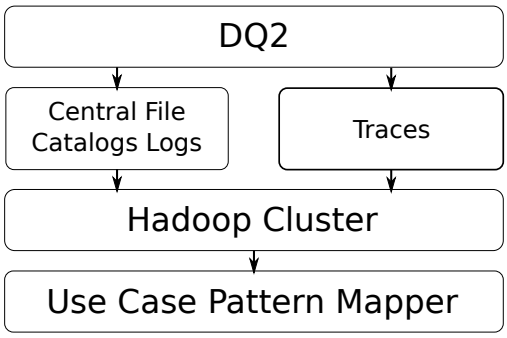

Figure 1. Analysing DQ2 workloads

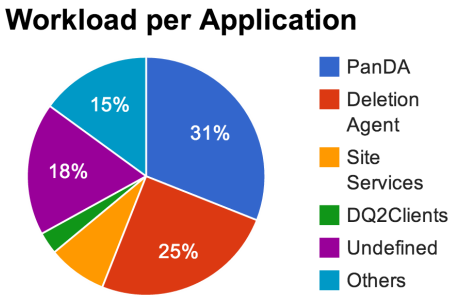

Figure 2. Workload per application

framework was implemented. Section 4 concludes the results gained so far and provides a quick overview about future work.

2. Profiling DQ2 Workload and Requirements

In order to gain reliable data about Rucio's scalability, we first needed to understand today's workload. We therefore analysed DQ2 log data since the beginning of 2013.

DQ2 produces around 100GB[1] of ASCII log data per day, distributed over multiple log files. Due to this huge amount of data, we decided to use our Hadoop cluster to analyse the log data. The cluster consists of 10 nodes with 24GB of RAM and 3TB HDD and 4 nodes with 48GB of RAM and 5TB HDD. We aggregated the log data into chunks of one hour and grouped it by `account`, `application identifier`, and `method` (representing one API call), calculating how often a method was called and what its mean response time was for each resulting group. Figure 2 illustrates the derived workload distribution over the different applications.

Next we assigned a pattern of API calls, each representing distinct use cases, that are executed by the various applications and users. The *Use Case Pattern Mapper* applies these use case pattern definitions to the data derived from the analysis, resulting in information about how frequently a use case is executed. The separate steps of the workload analysis are outlined in Figure 1.

This way we have been able to match 61% of DQ2 workload to known use cases executed by the provided applications. Table 1 summarises the results of the use case pattern matching and further indicates that besides 32% of the workload we were either unable to identify or considered irrelevant, we ended up having 7% as remaining 'noise' (i.e. unassigned API calls of included accounts). We assume that this is related to (a) retries of failed calls and (b) spontaneous or random access by users.

Last we deduced additional target numbers for each use case to validate the generated workload in a semantic way. To do so, we identified various performance metrics to verify against the production system. As this is very specific for each separate use case, and space in this paper is scarce, we only provide one example.

We use PanDA [3] as an example use case, as it causes one third of the overall workload and is one of the main applications utilising the DDM infrastructure. It is a pilot based workflow manager and is in charge of distributed data analysis of ATLAS data on behalf of the users.

For this use case we identified the number of created elements (e.g. datasets, replicas) as useful indicators. They (a) allow to check for semantic correctness and (b) are easy comparable with the production workload. Due to the timestamp applied to data in both systems (DQ2 and Rucio), these indicators further allow to check if the workload is properly distributed over time.

[1] about 25GB from *Central File Catalog Logs* representing API calls, and about 75GB from *Traces* representing data transfers between storage sites.

Table 1. Workload distribution over different applications

Application ID	Workload Share	API Calls	Mapped to Use Cases
pandasrv	31 %	440 K	80 %
deletion-agent	25 %	360 K	100 %
site-services	8 %	110 K	100 %
dq2clients	3 %	36 K	90 %
tzero	0.07 %	1 K	100 %
Identified API Calls	**68 %**	**947 K**	**61 %**
Undefined	18 %	250 K	-
Others	15 %	200 K	-

Table 2. Nominal target numbers for the PanDA use case per hour

Element Type	User Analysis	Group Analysis	Production T1	Production T2
Dispatched Tasks	800	82	12	12
Computed Jobs	17K	3.6K	10K	10K
Containers	400	82	32	32
Output Datasets	800	90	64	64
File Replicas	34K	7.2K	24K	16K
DIS Datasets	-	-	-	2.6K
DIS Files	-	-	-	12.6K
SUB Datasets	-	-	-	1.2K
SUB Files	-	-	-	16K

Table 2 provides a detailed overview of what and how many elements are created by PanDA per hour, based on the information provided by DQ2.

For an example how the data provided in Table 2 is represented in the emulation framework see the plot `Overall Created Elements` in Figure 4.

3. Emulation Framework

In this section we introduce the emulation framework developed for the purpose of scalability tests for Rucio. It is intended to be running continuously and provides valuable information about the systems performance during the daily development process of Rucio. An architectural overview of the emulator and its related infrastructure is illustrated in Figure 3.

The Emulator dispatches use cases in real-time to a distributed job queue (*Gearman Server* [5]). In order to explore the limits of Rucio it allows to adapt the scaling factor of the workload at runtime. For example, Figure 4 shows the emulation framework running twice the nominal load. As we wanted to apply new use cases in an easy way, it is enabled to invoke generic *Modules* on demand. Inside each module, multiple (correlated) *Use Cases* can be implemented. All use cases inside the same module share the same *Context* object, allowing them to persist/exchange information i.e. to be correlated. Further each use case can define an *input* method, providing access to the context object. This method is intended to acquire relevant data from the context object to properly execute the use case. The use case method itself will be executed by any

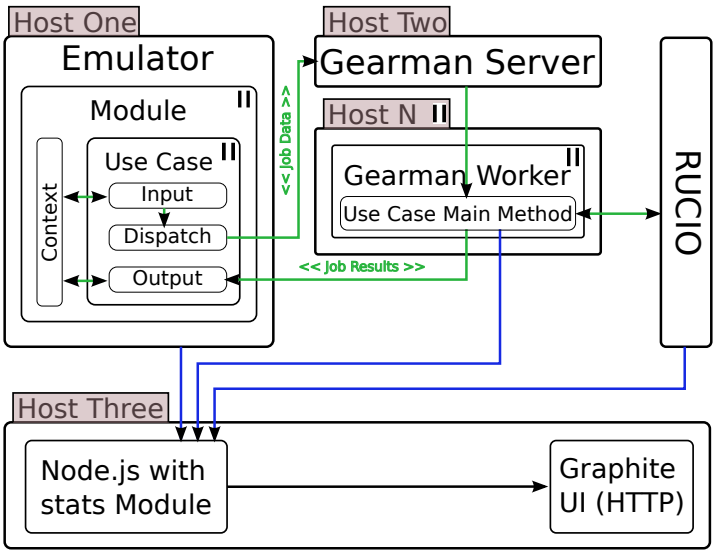

Figure 3. Overview of the emulation framework architecture

available (*Gearman*) worker. Additionally an *output* method is supported to store resulting data from a use case in the context object.

Each module may further define one *setup* and *shutdown* method. They can be used to prepare/post-process the context of the module. For example, loading and persisting the context object helps to immediately pick up the emulation where it was stopped before. In case of use cases having a long ramp-up time like PanDA, with a ramp-up time of approximately 20 hours, it is therefore possible to stop and restart without a significant loss of time. Releasing a new patch set or performing some maintenance work will not cause former progress to be lost and can therefore be done more frequently, complimenting agile development processes.

As we wanted to schedule the workload up to very high frequencies (multiple kHz), the emulator was designed to scale easily. During startup, the aggregated frequencies of use cases defined in the same module are used to distribute the modules over the available processes (i.e. CPU cores). Each process then triggers the execution of a use cases 'input - dispatch - output' sequence in the right frequency.

Our tests, executed on a 2.3GHz quad-core CPU, showed that this multiprocess/multi-threaded architecture allows for around 1.2 kHz per process/core resulting in 4.8 kHz over-all frequency for simple use-cases with asynchronous workers.

Gearman Server and Gearman Worker [5] are part of a framework to distribute jobs over several worker nodes either in a synchronous or asynchronous way. The *Gearman Server* implements a First-In-First-Out queue where clients dispatch their job descriptions to. As soon as a *Gearman Worker* becomes available, the Gearman server assigns the next pending job to it and waits for the worker to report the outcomes of the job execution. If the results indicate a successful execution, the job description is removed from the queue, otherwise it is reassigned to a different worker for a retry.

This framework is very suitable for the emulator as it scales easily. If the queue size becomes too long, one can simply start new workers on any available host to provide additional computing power. Utilising this framework allows the emulator to use all available resources of the host system for scheduling instead of blocking them to wait for e.g. long running database operations to be finished.

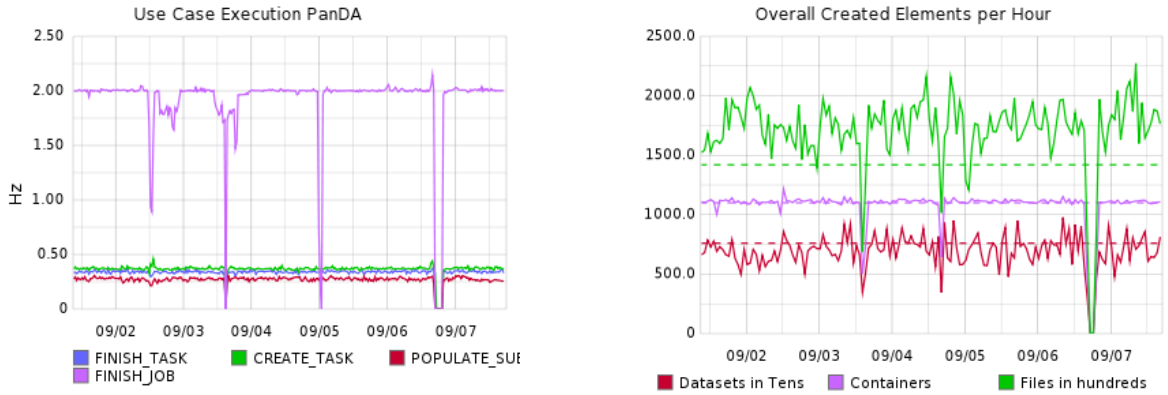

Figure 4. Example plots from Graphite when emulating two times the nominal Load

The Monitoring Infrastructure is the last part of our emulation framework. We decided to use *Graphite* [6] for this purpose. Graphite is designed to store several data points over time and provides a web-based user interface. Furthermore it supports the composition of plots with a set of time series which are used to illustrate the systems performance metrics in real-time. In Figure 4 two example plots are provided.

To minimize IO demands and increase performance, Graphite implements a fixed-size round-robin-database with the granularity of at least one second (1 Hz). Due to this 1 Hz granularity we decided to use Node.js [7] as an aggregator in front of it. Incoming data is now aggregated in memory and regularly flushed to Graphite, e.g. 1 Hz where data gets persisted on disk. For this application, saving only aggregated data is fine, as analysis is mostly done with the granularity of one hour, and it additionally helps reducing computation time and save disk space.

4. Conclusion and Future Work

These three components (Emulator, Gearman framework, and Graphite+Node.js) together provide a highly scalable and powerful framework to keep a constant, and realistic workload on Rucio. The detailed information about the run-time behaviour of the system is of great value while optimising Rucio for production.

In the next step we will use the emulation framework to elaborate different strategies for applications like PanDA to utilise the provided functionality by DDM like requesting a file transfer. Doing so allows not only optimisations for Rucio but also for other applications depending on data provisioning in the ATLAS experiment.

5. References

[1] CERN 2002 The ATLAS Experiment http://atlas.ch/ URL `http://atlas.ch/`
[2] Branco M, Zaluska E, de Roure D, Lassnig M and Garonne V 2010 *Concurrency and Computation: Practice and Experience* **22** 1338–1364
[3] Maeno T, De K, Wenaus T, Nilsson P, Stewart G a, Walker R, Stradling a, Caballero J, Potekhin M and Smith D 2011 *Journal of Physics: Conference Series* **331** 072024 ISSN 1742-6596
[4] Garonne V, Stewart G a, Lassnig M, Molfetas A, Barisits M, Beermann T, Nairz A, Goossens L, Barreiro Megino F, Serfon C, Oleynik D and Petrosyan A 2012 *Journal of Physics: Conference Series* **396** 032045 ISSN 1742-6588
[5] Ewart J 2013 *Instant Parallel Processing with Gearman* (Packt Publishing Ltd) ISBN 978-1783284078
[6] Graphite - Scalable Realtime Graphing URL `http://graphite.wikidot.com/`
[7] Teixeira P 2012 *Professional Node. js: Building Javascript Based Scalable Software* (Indianapolis, Indiana, USA: John Wiley & Sons, Inc.) ISBN 978-1118185469