ATLAS Replica Management in Rucio: Replication Rules and Subscriptions

M Barisits[1,2,3], C Serfon[3], V Garonne[3], M Lassnig[3], G Stewart[3], T Beermann[3], R Vigne[3], L Goossens[3], A Nairz[3] and A Molfetas[4] on behalf of the ATLAS Collaboration

[1] University of Innsbruck, Innsbruck, Austria
[2] Vienna University of Technology, Vienna, Austria
[3] CERN, Geneva, Switzerland
[4] University of Melbourne, Melbourne, Australia

E-mail: `martin.barisits@cern.ch`

Abstract. The ATLAS Distributed Data Management system stores more than 150PB of physics data across 120 sites globally. To cope with the anticipated ATLAS workload of the coming decade, Rucio, the next-generation data management system has been developed. Replica management, as one of the key aspects of the system, has to satisfy critical performance requirements in order to keep pace with the experiment's high rate of continual data generation. The challenge lies in meeting these performance objectives while still giving the system users and applications a powerful toolkit to control their data workflows. In this work we present the concept, design and implementation of the replica management in Rucio. We will specifically introduce the workflows behind replication rules, their formal language definition, weighting and site selection. Furthermore we will present the subscription component, which offers functionality for users to proclaim interest in data that has not been created yet. This contribution describes the concept and the architecture behind those components and will show the benefits made by this system.

1. Introduction

The ATLAS[1] collaboration creates and manages vast amounts of data. Since the detector started data taking, Don Quijote 2 (DQ2)[2], the collaboration's distributed data management system, is responsible for managing Petabytes of experiment data on over 750 storage end points in the Worldwide LHC Computing Grid[3]. DQ2 organizes, transfers and manages not only the detectors RAW data, but also the entire lifecycle of derived data products for the collaboration's physicists. This is done in accordance with the policies established in the ATLAS Computing Model.

As laid out in [4], it is very difficult to extend DQ2 to satisfy new high-level use cases and still cope with the load the system is facing today. For this reason and to cope with the load of the next decade, the next generation of the ATLAS DDM system, called Rucio, was introduced.

In this article we describe a new way of managing replicas, called replication rules. This concept, which is implemented in Rucio, gives the system users great flexibility to control their replica placement workflows, while at the same time it gives the system more freedom to save

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

resources. We also introduce Rucio subscriptions, which offer the user a way to define replica placement for future data.

The paper is organized as follows: In Section 2 we describe the high-level concept of Rucio and give a short overview of its architecture. Section 3 continues to introduce replication rules, their architecture and workflow as well as the evaluation of the component. Section 4 does the same for subscriptions. Finally we conclude in Section 5.

2. Rucio

This Section describes the general concept of Rucio as well as giving an overview of the architecture of the system.

2.1. Concept

A Rucio account is the unit of assigning permissions in Rucio. An account can represent individual ATLAS users, a group of users or an organised production activity. Every account has a data namespace identifier called scope. The scope is used to partition the data namespace, to easily separate production data from individual user data. In general, accounts can only write to their own scope, but privileged accounts (like production accounts) can be authorized to write into foreign scopes. Credentials, such as username/password, X509 certificates or Kerberos tokens are used to authenticate with Rucio. Such credentials can map to multiple accounts, for example, when a user is authorized to do operations on behalf of a group account.

The ATLAS Collaboration creates and administers large amounts of data which are physically stored in files. For Rucio, these files are the smallest operational unit of data. Files, however, can be grouped into datasets and moreover, datasets can be grouped into containers. We consequently refer to files, datasets or containers as data identifiers (DID), as all three of them refer to some set of data. A data identifier is a tuple consisting of a scope and a name. In Rucio each $(scope, name)$ tuple is unique. Datasets as well as containers may be overlapping in the sense that their constituents may be part of other datasets or containers.

To address and utilize storage systems in Rucio, the logical concept of the Rucio Storage Element (RSE) is introduced. An RSE is a container of physical files (replicas) and is the unit of storage space within Rucio. Each RSE has a unique name and a set of attributes describing properties such as protocols, hostnames, ports, quality of service, storage type, used and available space, etc. Additionally, RSEs can be assigned with meta attributes to group them in many logical ways, e.g. all Tier-2 RSEs in the UK, all Tier-1 RSEs, etc. Quotas assigned to accounts specify how much data volume an account can use on a specific RSE or a set of RSEs.

The replica management concept in Rucio is based on replication rules and subscriptions. Replication rules define the workflow for replication of existing data, while subscriptions are a concept which defines how the system manages future data. Both concepts and components will be explained in Sections 3 and 4.

2.2. Architecture

The Rucio software stack (see figure 1) is separated into three horizontal layers and one orthogonal vertical layer. It is implemented in Python 2.6.

The **Rucio clients** layer offers a command line client for users as well as application programming interfaces which can be directly integrated into user programs. All Rucio interactions are transformed by the client into https requests which are sent to the REST[5] interface of the Rucio server. Consequently, external programs can also choose to directly interact with the REST API of the server (e.g. by libcurl).

The **Rucio server** layer connects several Rucio core components together and offers a common, https based, REST API for external interaction. After a request is received by the REST layer, the *authorization* component checks the used credentials. If permitted,

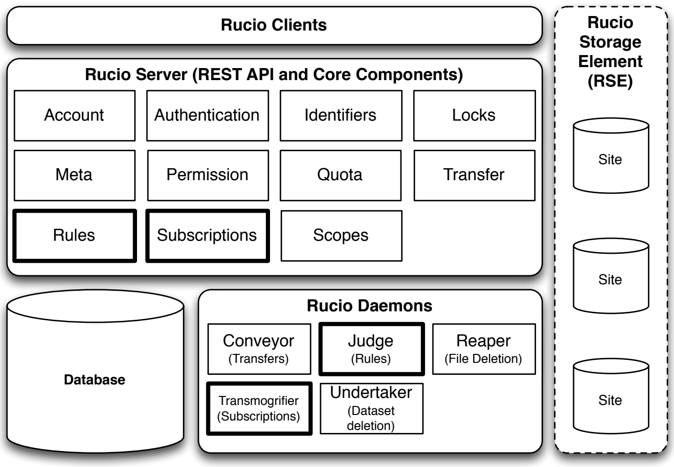

Figure 1. Overview of the Rucio architecture (the components featured in this article are highlighted)

the permissions of the account to execute the given request are checked by the *permission* component. If allowed, the request is passed to the responsible core component for execution. Rucio core components are allowed to communicate with each other, as well as with the *Rucio storage element* abstraction.

The **Rucio Storage Element** (RSE) abstraction layer is responsible for all interactions with different Grid middleware tools which interact with the Grid storage systems. It effectively hides the complexity of these tools and combines them into one interface used by Rucio. The abstraction layer is used by the clients, the server as well as the Rucio daemons.

The **Rucio daemons** are used to asynchronously operate on requests made by users or by the Rucio core. This can be transfer requests, executed by the *Conveyor*, expired replicas or datasets deleted by the *Reaper* or *Undertaker* as well as rule-re-evaluations and subscriptions performed by the *Judge* and *Transmogrifier*.

The **Database** is used to persist all the logical data as well as for transactional support. Only the Rucio server or daemons directly communicate with the database. Rucio uses SQLAlchemy[6] as an object relational mapper for performance as well as for development reasons.

3. Replica Management in Rucio – Replication Rules

Replica management in Rucio is based on replication rules. The general idea of replication rules is that instead of defining a specific destination for data to be replicated to, the user expresses the intention behind the replication request. Consequently, the system is able to interpret those requests and choose the appropriate destinations while preserving system resources, like storage space and network bandwidth. Replication rules can be as specific as specifying a single RSE. To give a more specific example: a user wants to replicate a dataset to two Tier-2 RSEs in the United Kingdom. The user can create a replication rule with 2 copies and the RSE expression `'tier=2&country=uk'`. The string `'tier=2'` represents the set of all Tier-2 RSEs and the string `'country=uk'` the set of all RSEs in the United Kingdom. The set-union operator `'&'` is used to create the set-union of both sets (The RSE expression language will be discussed in detail in section 3.1). Thereupon Rucio picks two ideal destinations based on existing and queued replicas.

A replication rule can be created for any data identifier in Rucio, independent of the scope or creator of the data identifier. When specified on a dataset or container, the rule will affect all contained datasets or containers. Subsequent changes to these datasets or containers will be considered by the replication rule.

Internally, Rucio processes replication rules and creates a replica lock for each replica created or covered by the rule. Replicas with at least one replica lock are exempt from the deletion procedure. Quota calculations are based on replica locks and not actually created replicas. Therefore quotas are not based on the amount of physically created replicas. For example, a single replica can have multiple replica locks from different rules and all accounts will pay from their quota for the file, not only the account who was responsible for creating the replica in the first place.

Once a replication rule is removed, the associated replica locks are removed as well. Replicas without any replica lock are flagged to be picked up by the deletion service.

Possible parameters for a replication rule are:

- `dataidentifier`: The data identifier the rule affects.
- `owner`: The account owning the replication rule. RSE permissions and quotas are checked based upon this account.
- `copies`: The number of replicas the replication rule should cover (Replication factor).
- `RSE Expression`: The RSE expression defining a set of RSEs the site selection algorithm should consider. The RSE Expression formal language definition is covered in Section 3.1.
- `grouping`: When a replication rule is specified for a dataset or container, the replicas can be grouped in three different ways: The `NONE` grouping defines that no grouping should be used and the selection algorithm selects an RSE randomly for every file. The `ALL` grouping option means that all files are replicated to the same RSE. The `DATASET` grouping defines that all files within the same dataset are replicated to the same RSE.
- `weight`: The RSE selection algorithm can respect weights assigned to the RSEs as meta attributes. The weight option defines the name of the weight to be used.
- `lifetime`: Lifetime of the replication rule. Once expired, the rule gets deleted and the replicas, if not covered by another rule, are marked for deletion.
- `locked`: A lock flag which additionally protects the rule from deletion.
- `subscription_id`: If the replication rule is created by a subscription, the subscription id will be stored with the rule.

Additionally, every replication rule is in one of four states:

- `OK`: All replicas have been created, no further actions are needed.
- `REPLICATING`: The rule is currently replicating at least one file to its destination.
- `STUCK`: At least one file could not be replicated to its target destination. Rucio will try to repair Stuck replication rules and select a different target destination, if possible.
- `SUSPENDED`: This manually set state defines that no further actions are taken to satisfy the rule.

3.1. RSE Expression formal language definition

RSE Expressions are strings based on the RSE Expression language defined in this Section. The expressions of each rule are interpreted by Rucio and result in a non empty set of RSEs. For example, to define a replication rule considering all German and French Tier-2 sites, the suitable replication rule would be "`tier=2&(country=FR|country=DE)`", which is equivalent to the set of all Tier-2s intersected with the set of all french and german sites.

The formal language $\mathbf{L}_{rseexp}$ is defined by the formal grammar $\mathbf{G}_{rseexp} = (N, \Sigma, P, S)$, with $N = \{RSE, ATTR, PRIM, B\}$, $\Sigma = \{A, \ldots, Z, a, \ldots, z, 0, \ldots, 9, (,), \cap, \cup, \backslash\}$, S is the start symbol and the following production rules P:

(i) $S \rightarrow B$

(ii) $S \to PRIM$
(iii) $B \to RSE$
(iv) $B \to ATTR$
(v) $PRIM \to B \circ B$ where $\{\circ \in \{\cap, \cup, \setminus\}\}$
(vi) $PRIM \to B \circ (PRIM)$ where $\{\circ \in \{\cap, \cup, \setminus\}\}$
(vii) $PRIM \to (PRIM) \circ B$ where $\{\circ \in \{\cap, \cup, \setminus\}\}$
(viii) $PRIM \to (PRIM) \circ (PRIM)$ where $\{\circ \in \{\cap, \cup, \setminus\}\}$
(ix) $PRIM \to (PRIM)$

The symbol B is an auxiliary symbol, to reduce the amount of production rules.

The RSE symbol corresponds to the name of a specific RSE. Its ASCII representation in Rucio is defined by the regular expression "`[A-Z0-9]+([_-][A-Z0-9]+)*`".

The $ATTR$ symbol corresponds to match of a specific RSE attribute with a value. These attributes are meta tags assigned to RSEs. Its ASCII representation in Rucio is defined by the regular expression "`[A-Za-z0-9\.]+=[A-Za-z0-9]+`". An attribute match results in, possibly empty, set of RSEs.

The $PRIM$ symbol corresponds to operation primitives. Operation precedence can be given with brackets. The classical set operations $\cap, \cup, \setminus$ are defined (in ASCII they correspond to the characters "`&`", "`|`" and "`\`").

3.2. Architecture & Workflow

The replication rule architecture is split into two components: The replication rule core component residing in the Rucio core on the server and the replication rule daemon, called Judge. The core component synchronously processes `create`, `delete` and `list` replication rule requests and immediately gives a result to the caller. The daemon asynchronously operates on events, which due to their nature cannot be done synchronously, like deleting expired replication rules. It also re-evaluates rules on datasets and containers which have been changed, which technically could be done synchronously in the core, but we chose to separate this workflow from the dataset/container change workflow for performance reasons.

The workflow of replication rule creation is defined as follows:

(i) Parse the RSE Expression of the rule and transform it into a set of RSEs.
(ii) Resolve the data identifier into a list of containing files, datasets and containers as well as their replica locks.
(iii) Check quotas and permissions of the account.
(iv) Based on the chosen grouping, for each group of files (or all of them) choose the optimal RSE, which is based on existing or scheduled replica locks as well as the weights specified by the weighting option.
(v) Create the replica locks and, if necessary, create transfer requests if replicas have to be created.

The RSE selection algorithm is set to minimize the amount of created transfers. However, the algorithm can be replaced by a different algorithm which follows a different policy. Step (ii) of the workflow row-locks the considered replica locks in the database, as the workflow has to make sure that no replicas get deleted while they are under consideration for new rules.

The Judge is actually split up into three parts. The rule-cleaner, which deletes expired replication rules; the re-evaluator, which re-evaluates rules for datasets/containers which have been changed; and the rule-repairer, which tries to repair erroneous rules, by selecting different replica destinations, if allowed by the RSE expression. The daemons are configurable in amount of processes to run and amount of threads per process. This way the request queue gets processed by as many separate threads as necessary for the workload.

3.3. Evaluation
To evaluate the scalability of Rucio and the replica management component, Rucio was engaged in a stress test with nominal and twice the peak workload measured on DQ2. For detailed information on the stress test environment and setup, see [7]. It was difficult to come up with a target value for the rule creation rate, as the concept does not exist in DQ2. However, the nominal amount of created replicas is equivalent to the one in DQ2. Figure 2 shows the frequency of `delete-rule` calls on the core as well as rule deletions due to rule expiration. Figure 3 shows the frequency of re-evaluate operations carried out by the Judge (1 process with 60 threads). The component managed to cope with the workload very well without accumulating any backlogs.

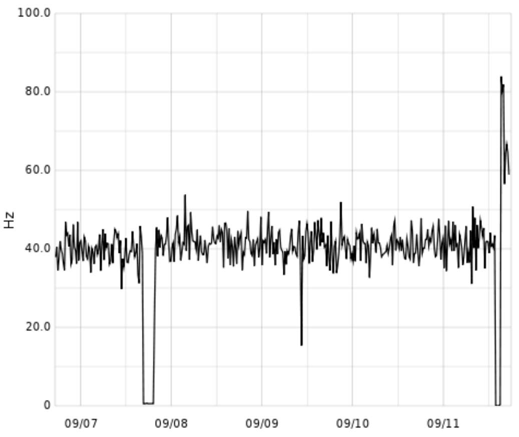

Figure 2. Delete replication rule calls (rules on files/datasets/containers)

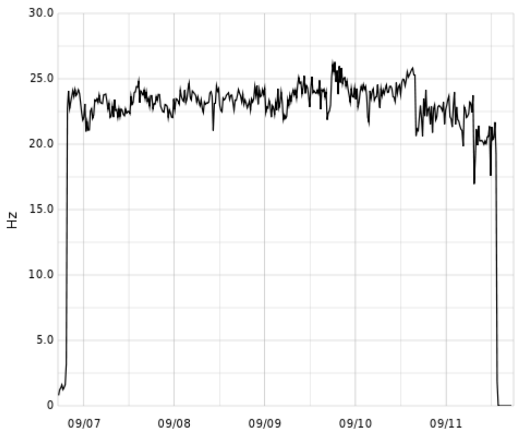

Figure 3. Judge re-evaluate operations for datasets/containers

4. Subscriptions
While replication rules are responsible for the replica management of existing data, subscriptions are responsible for the replica management of not yet existing data. They are mainly used to represent collaboration policies for data replication. Subscriptions generate replication rules for newly created data identifiers, by matching their meta data at registration time. Subscriptions are owned by an account and specify a set of replication rules as well as a set of meta data which they use for matching. An example for the meta data match would be ”`project=data12_8TeV,dataType=RAW,stream=physics,DIType=dataset`”. The subscription definition only allows to connect the meta data attributes by the logical and operator `','`.

4.1. Architecture & Workflow
The core component of the subscriptions is responsible for simply creating, deleting and listing subscriptions. As subscriptions are not matched synchronously during data identifier creation, the corpus of the work is done asynchronously by the Transmogrifier daemon, which is split into a polling agent and a pool of workers. Whenever a data identifier is registered in Rucio, it gets flagged as `new` by the Rucio core. The workflow of the Transmogrifier daemon is shown in figure 4 and defined as follows:

(i) The Poller reads a bulk of new data identifiers from the database and also reads all currently active subscriptions.
(ii) The Poller creates work packages (subset of the data identifiers & all subscriptions) and sends them to the Gearman server.
(iii) The Gearman server distributes each work package to one of the idling workers.

(iv) Each worker matches each data identifier's meta data with the meta data of the subscriptions. If a successful match is made, it generates replication rules for the data identifier and stores them to the database. The created replication rules are interpreted by the rule component momentarily.

(v) After matching, the worker removes the `new` flag from the data identifier on the database.

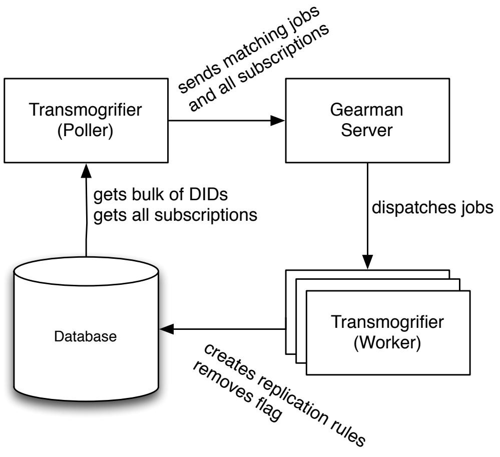

Figure 4. Workflow of the Subscription daemon

4.2. Evaluation

The subscription component was also engaged in the same stress test as mentioned in Section 3.3. Twice the nominal peak load of container/dataset/file creation was used with a representative set of subscriptions from DQ2. The Transmogrifier daemon and its workers passed this stress test without any performance problems.

5. Conclusion

DQ2, the data management system of the ATLAS experiment has provided the collaboration with a well functioning system during the last years. However, due to conceptual limitations in the original design, it is very difficult to extend the current system while handling the workload of the coming years. For this reason, Rucio, the next generation data management system of the ATLAS experiment, was introduced. In this article, we introduce replication rules which give users a powerful toolkit to manage their replication workflows, while at the same time giving Rucio more liberty to save resources, like bandwidth and storage. We also present the subscription component, which matches meta attributes of new data with existing subscriptions and generates replication rules for matching data identifiers. We show the concept and architecture of these components and evaluate them by engaging them in a stress test.

References

[1] Jones R and Barberis D 2008 The ATLAS computing model *Journal of Physics: Conference Series* vol 119 (IOP Publishing) p 072020
[2] Branco M, Zaluska E, De Roure D, Lassnig M and Garonne V 2010 *Concurrency and Computation: Practice and Experience* **22** 1338–1364 ISSN 1532-0634
[3] Bird I, Bos K, Brook N, Duellmann D, Eck C, Fisk I, Foster D, Gibbard B, Girone M, Grandi C *et al.* 2008 LHC computing grid *EGEE, Mar* vol 18
[4] Garonne V, Stewart G A, Lassnig M, Molfetas A, Barisits M, Beermann T, Nairz A, Goossens L, Megino F B, Serfon C *et al.* 2012 The ATLAS distributed data management project: Past and future *Journal of Physics: Conference Series* vol 396 (IOP Publishing) p 032045
[5] Fielding R T and Taylor R N 2002 *ACM Transactions on Internet Technology (TOIT)* **2** 115–150
[6] Bayer M 2013 *URL http://www.sqlalchemy.org/. Accessed on the 2013-09-12*
[7] Vigne, Schikuta, Garonne, Stewart, Barisits, Beermann, Lassnig, Serfon, Goossens and Nairz 2013 *Journal of Physics: Conference Series* **to appear**