Journal of Physics: Conference Series **664** (2015) 062002 doi:10.1088/1742-6596/664/6/062002

Resource control in ATLAS distributed data management: Rucio Accounting and Quotas

M Barisits[1,2], C Serfon[1], V Garonne[3], M Lassnig[1], T Beermann[1], R Vigne[1] on behalf of the ATLAS Collaboration

[1] CERN, Geneva, Switzerland
[2] Vienna University of Technology, Vienna, Austria
[3] University of Oslo, Oslo, Norway

E-mail: martin.barisits@cern.ch

Abstract. The ATLAS Distributed Data Management system manages more than 160PB of physics data across more than 130 sites globally. Rucio, the next generation Distributed Data Management system of the ATLAS experiment, replaced DQ2 in December 2014 and will manage the experiment's data throughout Run 2 of the LHC and beyond. The previous data management system pursued a rather simplistic approach for resource management, but with the increased data volume and more dynamic handling of data workflows required by the experiment, a more elaborate approach is needed. Rucio was delivered with an initial quota system, but during the first months of operation it turned out to not fully satisfy the collaboration's resource management needs. We consequently introduce a new concept of declaring quota policies (limits) for accounts in Rucio. This new quota concept is based on accounts and RSE (Rucio storage element) expressions, which allows the definition of hierarchical quotas in a dynamic way. This concept enables the operators of the data management system to implement very specific policies for users, physics groups and production systems while, at the same time, lowering the operational burden. This contribution describes the concept, architecture and workflow of the system and includes an evaluation measuring the performance of the system.

1. Introduction

Rucio, the ATLAS [1] collaboration's distributed data management system manages about 160 petabytes of data on more than 750 storage endpoints in the Worldwide LHC Computing Grid[2]. Rucio replaced DQ2[3] in December 2014 and introduced many new features to the ATLAS users and applications. Rucio organises not only the RAW data from the detector, but also handles all data inputs and outputs from the collaborations users.

One important feature of Rucio is the control of storage resources to assure a fair usage of resources based on the collaboration's usage policy. These policies are expressed as quotas in the system.

In this article we describe how data was accounted and quota was managed in DQ2, how a first quota version was implemented in Rucio and then we introduce a new quota system for Rucio which will handle quotas throughout Run 2 of the LHC and beyond.

The paper is organised as follows: In section 2 we present key concepts of Rucio which are essential to follow this article. In section 3 we discuss quotas in general and give an evolution of storage quota systems in ATLAS distributed data management. We also point out how the

new quota system fits in here. In section 4 we explain the concept of this system, followed by section 5 where we describe the architecture and workflow. We continue in section 6 with an evaluation. Finally we conclude in section 7.

2. Rucio key concepts and architecture

2.1. Concepts

This section describes the key concepts of Rucio[4, 5] which are essential to follow this article.

In Rucio, every user, group or organised production activity is represented by an *account*. Accounts are also the unit of assigning permissions. 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.

Managing data is the primary function of any data management system. 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 used. 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.

To select a set of RSEs, *RSE Expressions* were introduced. RSE Expressions are strings based on the RSE Expression language defined in [4]. The expressions are interpreted by Rucio and result in a set of RSEs. For example, to specify an expression considering all German and French Tier-2 sites, the suitable expression 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 RSEs.

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 explicitly address a specific RSE, or be more generic such that they result in a list of RSEs, e.g. 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. Thereupon Rucio picks two ideal destinations based on existing and queued replicas.

A replication rule can be created for any data identifier in Rucio, independently 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 which was responsible for creating the replica in the first place. In the end, accounts are paying quota for the protection from deletion of a replica and not for the replica's creation.

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.

2.2. Architecture

The Rucio software stack is separated into three horizontal layers and one orthogonal vertical layer. It is implemented in Python 2.6[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[7] 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, 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. These 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[8] as an object relational mapper for performance as well as for development reasons.

3. Quotas and their evolution in ATLAS distributed data management

The main principle behind quotas is to partition storage resources for different users. Not only does production data need to have a guaranteed available partition separated from user data, but more importantly also user data needs to be partitioned somehow fairly to protect the storage from overload by a single user. The term *quota* is a long established term in storage administration and can be mainly differentiated in block (or usage) quota, which is based on volume (bytes) and inode (or file) quota, which is based on the number of files. Furthermore quotas can be categorized in hard and soft quotas, which allows users to temporarily violate their quota cap.

3.1. Quotas in DQ2

In DQ2 quota was based on replica accounting reports, which were generated once a day. After the report was generated, a script ran over the report comparing the actual usage against a list of quota caps. For violating quotas email alerts were generated to inform the system operators about the violations.

The advantage of this system was its very low complexity. As the accounting reports had to be generated, for other purposes, anyway, the actual comparison tool was very simple. The disadvantage of the system was that it was, de-facto, no real quota system. As the quota could be exceeded beyond any limit, this concept was not even a soft-quota. Also, once the quota was exceeded, clean-up usually required a non-negligible amount of work, as users had to be contacted on a case-by-case basis.

3.2. First quota version in Rucio

With the initial release of Rucio we also delivered a fully functional quota system. Each quota is defined as $(account, RSE, bytes)$ tuple, which defines a hard quota for an account on a RSE. If no quota is set, the account has no permission to acquire replica locks or transfer files to this RSE, thus the default quota is defined as $bytes = 0$. Consequently to give accounts unlimited access to an RSE the bytes value can be set to $bytes = \infty$.

The fundamental part of this implementation is a consistent, realtime accounting of the current amount of locked replicas which each account has on each RSE. In DQ2 the replica volume was calculated only once a day as part of the accounting report, but this was clearly not sufficient to support a system with hard quotas. Rucio uses a counter-based system to keep track of the number of files and bytes each account locks on an RSE. These $(account, RSE, \#files, \#bytes)$ tuples are stored in an index organized table in the database to allow fast lookup and updates. Due to potential row lock contention on the database when updating the table from many different processes (daemons and server), there is only one master process which updates this table. This process, residing in the daemon layer of Rucio, reads from a queue table and updates the counters in one transaction. Whenever a rule is created or removed the respective differences are put into this queue table. Thus the counters are only eventual consistent, but the master process is quick enough to ensure that the delay between queue and master table is only several seconds.

The advantage of this system is that both management of quota as well as lookups are very simple. Whenever a new rule is created, the workflow only has to make one primary key based lookup in the quota table and one primary key-based lookup in the counter table, to see if an account is within its quota. This is very performant and we have demonstrated that this system runs very well.

The disadvantage of this system is, that it does not conceptually represent what the collaborations policies on disk quotas actually are. For example, this concept allows a quota for an account for each scratchdisk RSE. However, what is really needed, policy wise, is one quota for the account spanning over all scratchdisk RSEs. This kind of quota definition is fundamentally different.

While it would be possible to dynamically adapt single RSE quotas to achieve this kind of behavior, we decided against that, as it would again be a quota system outside of the Rucio core which would lead again to different consistency issues, as with DQ2. Instead the decision was made to design and implement a new quota system within Rucio.

4. Quota concept

The main goal the new quota system has to achieve is to support quotas involving multiple RSEs. A potential conflict arises if two or more quotas involve the same RSE. It can be very simply argued that in these cases, the lowest or highest quota should count, but in any case the

question arises what has to be done to represent exceptions. These were the main difficulties which had to be solved conceptually, before solving the architectural challenges.

With the new quota system we also decided to introduce the possibility to have quotas on the number of files. This is due to the reason that some storage systems do not scale well with high numbers of small files. To discourage users from creating many small files, but instead group them into tarballs or other archives, the concept of file (or inode) quota is introduced.

With the new quota system each quota is defined as $(account, RSEexp, \#bytes, \#files, level)$ tuple.

- The **account** field is the account this quota is valid for.
- The **RSEexp** field holds an RSE expression describing the set of RSEs the quota is valid for.
- The **#bytes** field is the maximum number of bytes the account can write to this set of RSEs. The value can be ∞.
- The **#files** field is the maximum number of files the account can write to this set of RSEs. The value can be ∞.
- The **level** field is an integer value setting the level of this quota. The level is used to create a hierarchy between quotas which is necessary to resolve exceptions and will be described next in this section.

When it comes to overlapping quotas the following policy is used to resolve conflicts during the quota evaluation process:

(i) Quotas with higher level supersede quotas with lower level.
(ii) Quotas with the same level: It is sufficient that one quota is violated.

This conflict resolution gives the policy creator maximum freedom to declare quotas and still be able to define exceptions. A typical policy, including an exception could look like this:

- All accounts have a global quota of 20TB on all scratchdisk RSEs. The respective quota looks like this: $(account = \text{"jdoe"}, RSEexp = \text{"scratchdisk=1"}, bytes = 20TB, files = \infty, level = 0)$
- However, no user should have more than 2TB on a single scratchdisk: $(\text{"jdoe"}, \text{"CERN\_SCRATCHDISK"}, 2TB, \infty, 0)$
- But user jdoe is a power-user on BNL-OSG2_SCRATCHDISK, where he should have infinite quota: $(\text{"jdoe"}, \text{"BNL-OSG2\_SCRATCHDISK"}, \infty, \infty, 1)$

5. Architecture

The quota components are integrated into both server as well as daemon workflows in the architecture, as replication rules are created both from servers and daemons. Also the client and REST layers have to be adapted to allow system operators the manipulation of quotas.

The architecture of the new quota system in Rucio has to solve one major challenge: with every rule creation some RSE_x is picked to write replicas to. As RSE expressions are used to describe the quotas, the system has to be able to efficiently identify which quotas apply to RSE_x. As an account can have multiple quotas the realtime evaluation of all RSE expressions would be rather slow. As the result of RSE expressions only changes very rarely, we decided in favor of a caching strategy. Thus, each RSE expression is only resolved once every day. To speed up the process of the quota evaluation, a daemon populates the cache once a day, so in normal operation the standard quota evaluation workflow does not have to contact the database to resolve RSE expressions.

To identify if a single quota is violated multiple RSE counters have to be aggregated. However, as these counter lookups are very fast primary key lookups, the performance of this workflow does not change significantly.

The next paragraph describes each step in the workflow of the quota evaluation, which is shown in figure 1:

(i) The first step is actually part of the rule creation workflow, thus it would have to be executed even without a quota system. It resolves the $RSEexp$ of the replication rule, to get a list of eligible RSEs: $(RSE_1, RSE_2, \ldots, RSE_n) \in RSE_N$. If the rule is created by a privileged account, the whole quota evaluation is skipped and the workflow continues directly with the rule creation.

(ii) In the next step the usage counters for all eligible RSE_N are retrieved. As these are primary-key based lookups against an index organized table in the database, the lookups are very fast.

(iii) Afterwards all quotas $(quota_1, quota_2, \ldots, quota_n) \in quota_N$ of the account are retrieved from the database. As the quota table is also index organized by account, this lookup is also very fast. However, at this point no selection of applicable quotas in $quota_N$ has been made.

(iv) In the next step, the RSE expression of each quota in $quota_N$ is resolved from the cache. If the resolved RSE set overlaps with RSE_N the quota is applicable and has to be evaluated in the next step, if not, the quota is not applicable. This operation is also very fast as only fast cache lookups are involved.

(v) In the next step for each applicable quota the usage counter comparison is made. For each RSE out of the set RSE_N a decision is made if there is enough quota or not. This operation is very fast as no database interaction is involved. The process is purely CPU bound.

(vi) In the last step a set of RSEs with sufficient quota is reported back to the rule creation workflow. The set might be empty leading to a rejection of the replication rule.

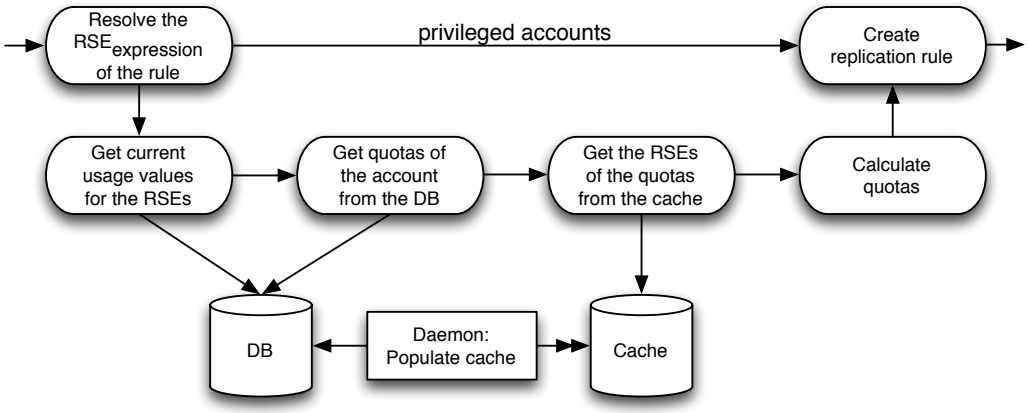

Figure 1. Quota evaluation workflow.

6. Evaluation

The evaluation of the new quota system is quite difficult as the current quota system in Rucio is fundamentally different to the new one. Also, as the new system is not in production yet, we

do not have a set of production-level quotas. Our approach to evaluate the system is as follows: Over a period of 24 hours we recorded all rule creations for scratchdisks on the production system of Rucio. For the new quota system, we created, for each account, a quota spanning over all scratchdisks and a quota for each single scratchdisk. For some selected accounts we also defined exceptional quotas (on a higher quota level).

We then replay all the rule creations on a test instance with the new quota system. For comparison, we recorded timings of the whole quota workflow (See figure 1) for the current and the new quota system. The timings are shown in figure 2.

In this evaluation the new quota system is about 25% slower than the current quota system, which is well explained by the higher complexity and added functionality. However, as the whole rule creation workflow takes 150ms, on average, an increase of 6ms is negligible.

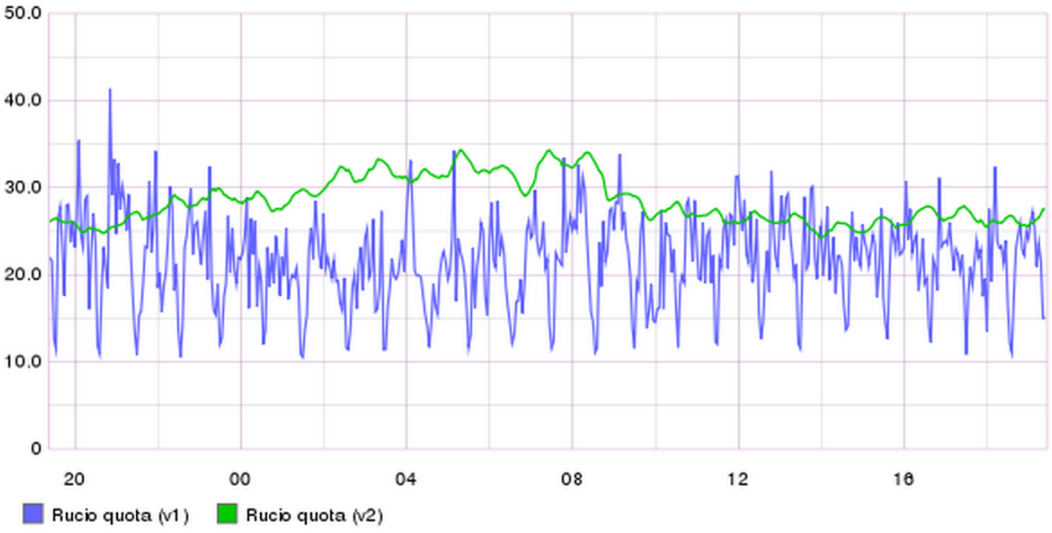

Figure 2. Quota evaluations in [ms] during a period of 24 hours.

7. Conclusion

Rucio, the data management system of the ATLAS experiment replaced DQ2 in 2014. Since then, Rucio provided many new features to the users and sub-systems of the Collaboration. Rucio was introduced with an initial quota system, but during operation it turned out to not fully satisfy the policy needs of the collaboration. In this article, we introduced a new quota system for Rucio. We first discussed replica accounting and quotas and their evolution in the different ATLAS distributed data management systems. We then introduced a new quota concept for Rucio and showed the architecture and workflow of the system. We also discussed how the performance critical parts of the workflow are efficiently designed. We then evaluated the new quota system by comparison with the current quota system of Rucio.

References

[1] ATLAS Collaboration 2008 *Journal of Instrumentation* **3** S08003 URL `http://dx.doi.org/10.1088/1748-0221/3/08/S08003`

[2] Bird I, Bos K, Brook N, Duellmann D, Eck C, Fisk I, Foster D, Gibbard B, Girone M and Grandi C 2008 *EGEE, Technical design report CERN-LHCC-2005-024*

[3] Branco M, Zaluska E, De Roure D, Lassnig M and Garonne V 2010 *Concurrency and Computation: Practice and Experience* **22** 1338–1364 URL `http://onlinelibrary.wiley.com/doi/10.1002/cpe.1489/full`

[4] Barisits M, Serfon C, Garonne V, Lassnig M, Stewart G, Beermann T, Vigne R, Goossens L, Nairz A, Molfetas A and on behalf of the ATLAS Collaboration 2014 *Journal of Physics: Conference Series* **513** 042003 ISSN 1742-6588
[5] Garonne V, Vigne R, Stewart G, Barisits M, Eermann T B, Lassnig M, Serfon C, Goossens L, Nairz A and on behalf of the ATLAS Collaboration 2014 *Journal of Physics: Conference Series* **513** 042021 ISSN 1742-6588
[6] Python Software Foundation 2015 *URL https://docs.python.org/2.6/ Accessed on the 23 of April 2015*
[7] Fielding R T and Taylor R N 2002 *ACM Transactions on Internet Technology (TOIT)* **2** 115—-150
[8] Bayer M 2015 *URL http://www.sqlalchemy.org/. Accessed on the 22 of April 2015*