The Research Data Management System at the University of Groningen (RUG RDMS): architecture, solution engines and challenges

A. Tsyganov, S. Stoica, M. Babai, V. Soancatl-Aguilar, J. Mc Farland, G. Strikwerda, M. Klein, V. Boxelaar, A. Pothaar, C. Marocico, J. van den Buijs
University of Groningen,
Groningen, Netherlands
rdms-support@rug.nl

ABSTRACT

RUG Research Data Management System (RUG RDMS) is powered by iRODS. It is developed at the University of Groningen to store and share data, and allow collaborative research projects. The system is developed based on open-source solutions, providing access to the stored data by means of command line, WebDAV, and a web-based graphical user interface.

The system provides a number of key functionalities and technical solutions such as metadata template management, data policies, data provenance, and activity auditing. It uses some of the existing iRODS functionalities and tools like the iRODS audit plugin to track the data operations or the iRODS rule engine plugin for Python to implement a set of custom developed rules. It allows users to configure and tune metadata templates for different research areas. Furthermore, iRODS advanced rule and composable resource functionalities have been used to automate metadata extraction on demand and allow long term storage on magnetic tapes. In addition, a set of custom system policies that is used for data handling has been developed. This provides a flexible environment on top of the current iRODS rules. Data replication on the storage level guarantees full data recovery.

The system is accessible via the local HPC facilities and provides a landing zone to support data to compute. The architectural design of the system allows both vertical and horizontal scalability with the potential of unifying different iRODS installations and facilities.

Keywords

RUG RDMS, metadata templates, data policies, iRODS Python rule engine plugin, iRODS auditing module.

INTRODUCTION

The general lack of centralized data storage and management within research institutions increases the risk that valuable data disappears or is more difficult to find. Moreover, datasets are increasing in volume, complexity, heterogeneity, and speed of generation [1]. This is why the demand for high quality and intuitive scientific data management tools is high and most of the funding agencies nowadays require a data management plan in an early stage of a research project. As a research output, data are often compared to a public good that should be made available to the community for re-use if possible. There is a growing demand for research data integrity, quality, public availability, and reusability, the FAIR principles [2]. In the context of research data management, the documentation of research data is an essential duty. In order to ensure the interpretability, understandability, shareability, and long-lasting usability of the data, any data management system should support standardized, and interoperable metadata. Effective data management can actually contribute to the increased pace of the research process, contribute to the

iRODS UGM 2021, June 8-11, 2021, Virtual

soundness of research results, and meet funding agency requirements by making research data easy to manage and share over the long term. A good example of such a system is YoDa, created at Utrecht University, Nederlands [3][4].

The RUG Research Data Management System (RUG RDMS) system has been developed to support the scientific research community at the University of Groningen [5] and University Medical Center Groningen (UMCG) [6] in the process of handling its research data. It provides an inhouse solution for the data storage, data access management, in addition to data management policies.

ARCHITECTURE

The main idea behind the architecture of the RUG RDMS is to split the system into a number of services that do not depend on the implementation. Figure 1 illustrates a number of logical components and the decisions that were made in the design of the RUG RDMS to make it a scalable and robust system. The system has been designed to store vast amounts of data, while keeping everything on-premises. The backend is based on iRODS [7][8], an open-source data management software that supports collaborative research effectively. iRODS is data-grid middleware that virtualizes access to data regardless of which physical device the data is stored on. It accomplishes this by mapping physical files and directories to logical data objects and collections. Users can make use of various interfaces to search for and retrieve data. In addition to facilitating data discovery and retrieval, iRODS has a robust security system to implement fine grained access control and facilitate robust activity auditing. The defining concept of the RUG RDMS is using iRODS' powerful metadata storage and discovery functionality to document the data stored in it.

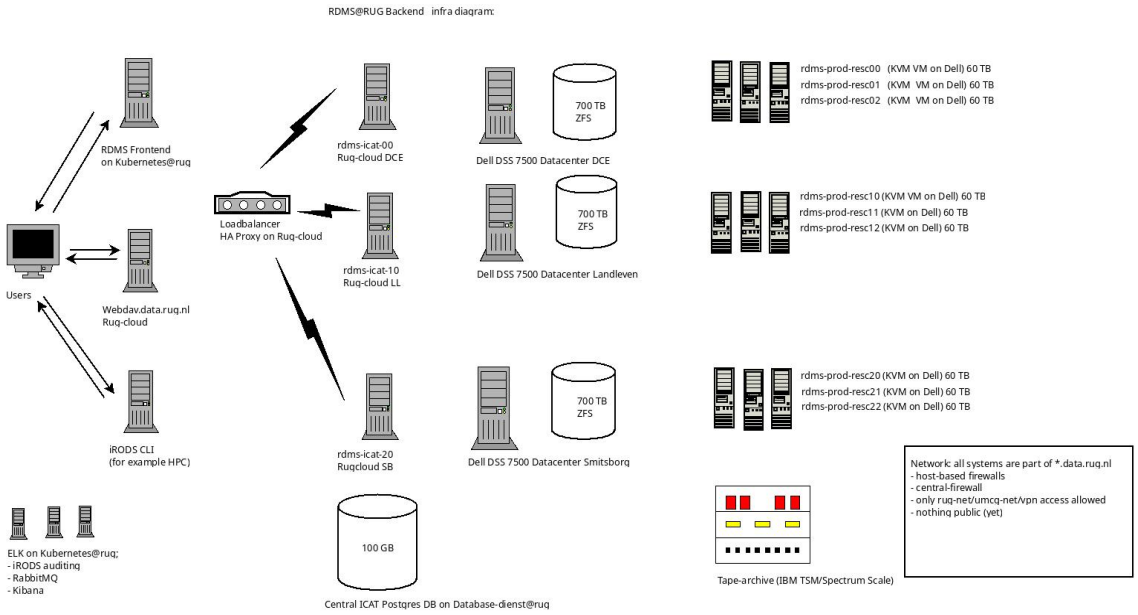

Figure 1. Users interface with the RUG RDMS in one of three ways: the web front-end, a WebDAV connection, or the iRODS CLI. Three iRODS catalog providers access a single PostgreSQL database and sit behind an HAProxy. The nine current resource servers (iRODS consumers) have multiple volumes "mounted" from a local storage server. The RUG RDMS storage is composed of 3 big storage servers located in three different data centers. These servers utilize a ZFS filesystem pool with built-in support for data-deduplication, compression, 'self'-healing, and software RAID among other features.

The RUG RDMS is configured such that it should survive broken data disks (spare and hot swappable), broken volumes, broken servers, one downed data-centre, small scale network outages, etc. Maintenance can also be performed easily, since resource virtual machines are small and can be moved or migrated if needed. There are several iRODS zones with the possibility to add new zones when necessary. Currently, next to the iRODS zones dedicated for the development, the production iRODS zones include one for the majority of the university projects and one for

the collaboration with the UMCG. In addition to the online storage resources, there is a nearline, long-term archive resource utilizing a tape storage system.

APPLICATION DESIGN AND WEB INTERFACE

Illustrated in Figure 2, the whole system is divided into a number of components intended to be independent services with their own setup and configuration:

- Proxy: the main and only entry point for the Web Interface. The proxy service is implemented using Nginx [9], the most popular proxy server.
- Web Server: The RUG RDMS uses Django as a base framework to deliver web content in combination with RUG css templates and Bootstrap 4 libraries. Django [10] is a high-level Python Web framework that encourages rapid and clean development and powers many of the Web's sites.
- Backend: iRODS with the rule engine plugin for Python and custom iRODS rule bases dedicated to the RUG RDMS functionality. Such rules include blocking of select operations on (meta)data, automatic metadata extraction, notification messages, etc.
- REST-like services within a private network: all other services that RUG RDMS uses in the intermediate level are based on REST APIs [11] or have a direct connection to the mid-tier database. This design choice makes internal communication between different modules of the system independent from the particular programming code of a service. For instance, communication with the iRODS backend is done via the Java Jargon API using Jersey REST API [12] and the Apache Tomcat Web server [13]. However, in the near future, there is a plan to switch to Python iRODS client implementation. This transition is especially easy because the implementation of the REST calls from the other parts of the system to iRODS will be the same.
- Davrods [14] and NFSRODS [15]: DavRods and NFSRODS are two implementations of a standard file system protocol to mount iRODS data as a filesystem directly on the client side.
- iCommands [16] access for the users: collectively, the iCommands are a default iRODS tool that is used to manage data on the server and on the client side.

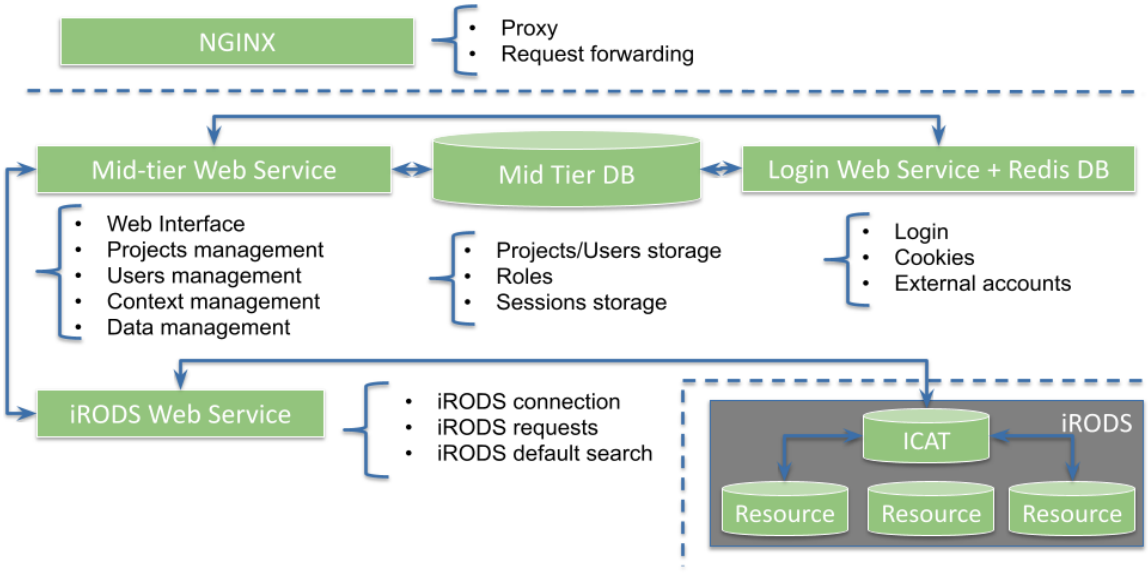

Figure 2. RUG RDMS Logical Architecture

Since the beginning of the project, there have been several versions of the web interface. Our current solution has many similarities to typical cloud storage interfaces' look and feel as shown in Figure 3.

1. The first logical type of the data is in the user's data area. It is a collection in which the user is the sole owner and manager of the data.
2. The second logical type is in the team drive. A team drive is a shared area that can be created and managed by data managers. Each scientific group can have their own data managers. Within RUG RDMS, data managers are users with the iRODS groupadmin role. Users with this role are allowed to create and manage iRODS groups. The idea behind the team drives is to provide users with a collaborative working area with flexible permission management.
3. The third logical data type resides in the project area. It contains all the facilities to fully support the data lifecycle management. Project collections can be created and managed by another special group of users: project managers. When a new project is created, three dedicated user groups are immediately attached to it. These groups have read, write and ownership of the data, respectively. Every user who is a member of a project will be at least a member of the read user group. Users that are considered owners of the data will be added to all three groups. A user that has write access to a project will be included in the write and read groups. This approach allows for fine grained permissions of the inner project data and archived data.

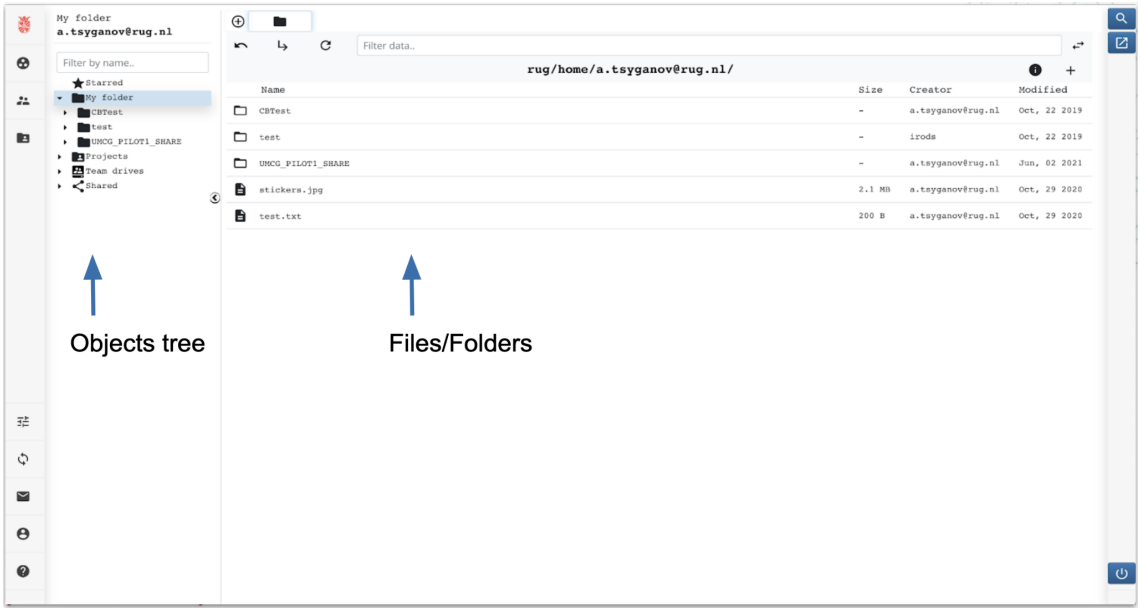

Figure 3. Main user landing page at RUG RDMS with its inbuilt data browser. The left panel shows the directory (collection) browser. The right panel shows the contents of a selected collection: data objects and sub-collections.

The metadata and metadata handling constitute another important aspect of the interface and its data management. RUG RDMS supports standard iRODS metadata operations enriched with custom rules. One such custom rule is automatic metadata extraction from standard data formats such as JPEG, TIFF, FITS, PDF, etc. This rule is triggered manually from the web interface or from the command line. It extracts all available metadata from a physical file having the supported format. With the evolution of the system the list of supported formats will only grow and the expansion of metadata templates will be used to choose what metadata is extracted and how it is stored. The interface workflow for running this metadata extraction is presented in Figure 4.

The automatic metadata extraction runs as a delayed rule on the iRODS backend that is submitted as a background job. While the results of the extraction may sometimes not be directly visible on the web interface, the user can follow the status of their submitted jobs on a dedicated page. Before such a background process starts, the process is first registered in the RUG RDMS mid-tier database. By using the mid-tier REST database service, the custom Python rule reports to the RUG RDMS about the process status.

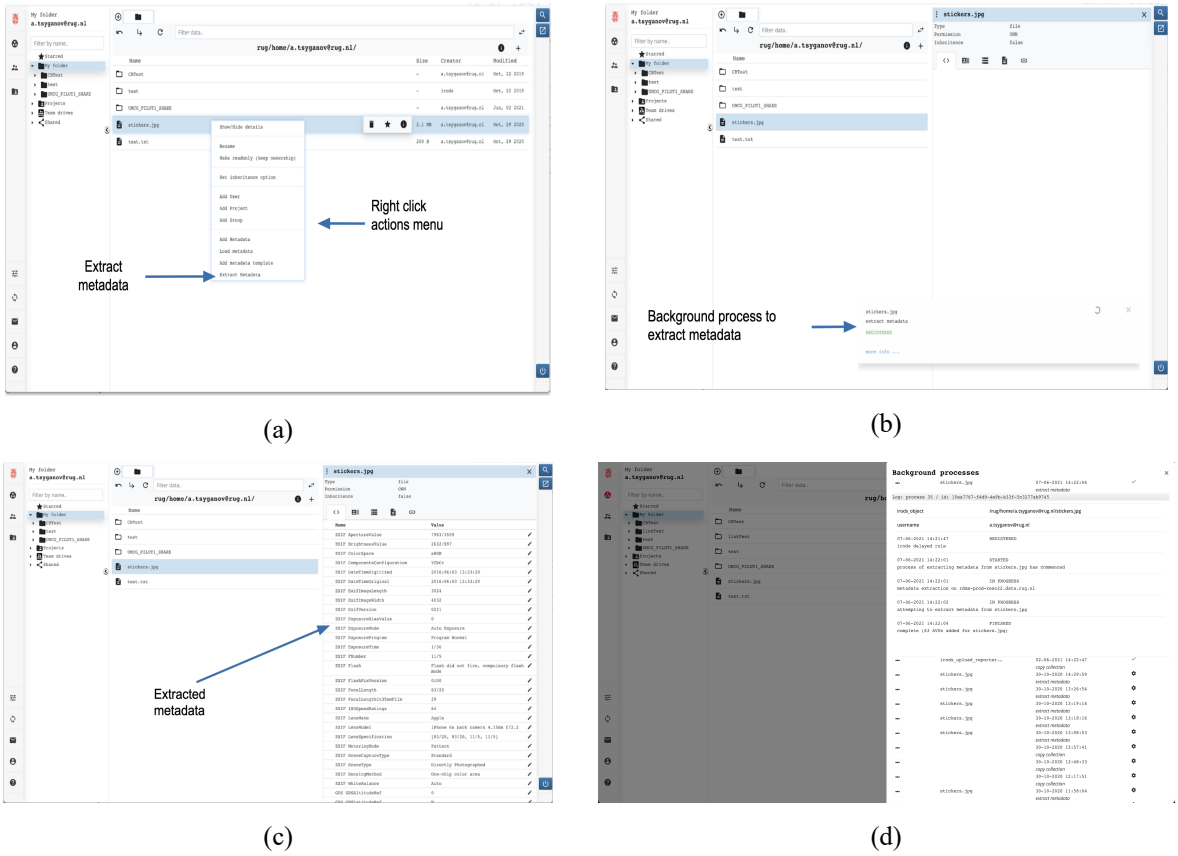

Figure 4. RUG RDMS web interface automatic metadata extraction. The four panels show (a) the right click action menu, (b) a screen with the background extraction process running, (c) the final result of the extracted metadata, and (d) extra information about the background extraction process.

CUSTOM POLICY ENGINE

Implementation of user workflows is one of the major requirements for the RUG RDMS. During the requirements engineering phase, it became clear that different research groups apply their own field-specific data processing, storage formats, and operational algorithms. A custom policy engine has been developed for this purpose. It utilizes the iRODS rule engine plugin for Python in combination with the mid-tier RUG RDMS architecture and iRODS metadata on particular collections and data objects. One needs to add special metadata to an object to initiate a custom policy.

iRODS metadata consists of three values: key, value, unit. Key – to identify metadata; value – to store actual metadata value; unit – an extra field that can be used for different purposes. Below there is an example of the metadata policy attached to the project collection «/testZone/home/Projects/project0_5n1»:

```
Name : sysmdt_rdms_policy_2c7197f0a89e1c842180756537534a81a069be79e8ec6fa1473af21c
Value: {
       "policy_name"      : "project_user_participation_enddate",
       "policy_creator"   : "atsG",
       "input_parameters" :
                    {
                        "user_name"   : "atsG",
                        "end_date"    : "10/06/2021 11:52",
                        "date_format" : "%d/%m/%Y %H:%M"
                    }
       }
Unit : POLICY|PROJECT|TORUN|atsG
```

To make a key-value pair unique, the object name is transformed into a non-reversible hash value and this is added to the key name. In the example above "/testZone/home/Projects/project0_5n1" equals to hash:

```
2c7197f0a89e1c842180756537534a81a069be79e8ec6fa1473af21c
```

The unit field is used to make this policy correctly searchable and executable on the iRODS server side. It also stores the type of the policy, status and the name of the user who initiated the policy.

The value field contains a JSON string with the information about the policy. This string is used to run Python code constructed from the "policy_name" and "input_parameters". Here is a code example to run such a JSON string:

```
from .policy_project_user_participation_enddate import
       run_policy_project_user_participation_enddate

C_AVAILABLE_POLICIES = {
"project_user_participation_enddate" : run_policy_project_user_participation_enddate
}

# method to run policy that was fetched from the metadata of the object
def run_policy(self):
   l_function_name, l_parameters = self.parce_policy()# parse JSON
   if self.policyIsValid():
       if self.c_namespace.C_AVAILABLE_POLICIES.has_key(l_function_name):
           # execute code
           self.c_namespace.C_AVAILABLE_POLICIES[l_function_name](self, l_parameters)
```

After the policy metadata is set it is picked up by the server cron process illustrated in the diagram in Figure 5.

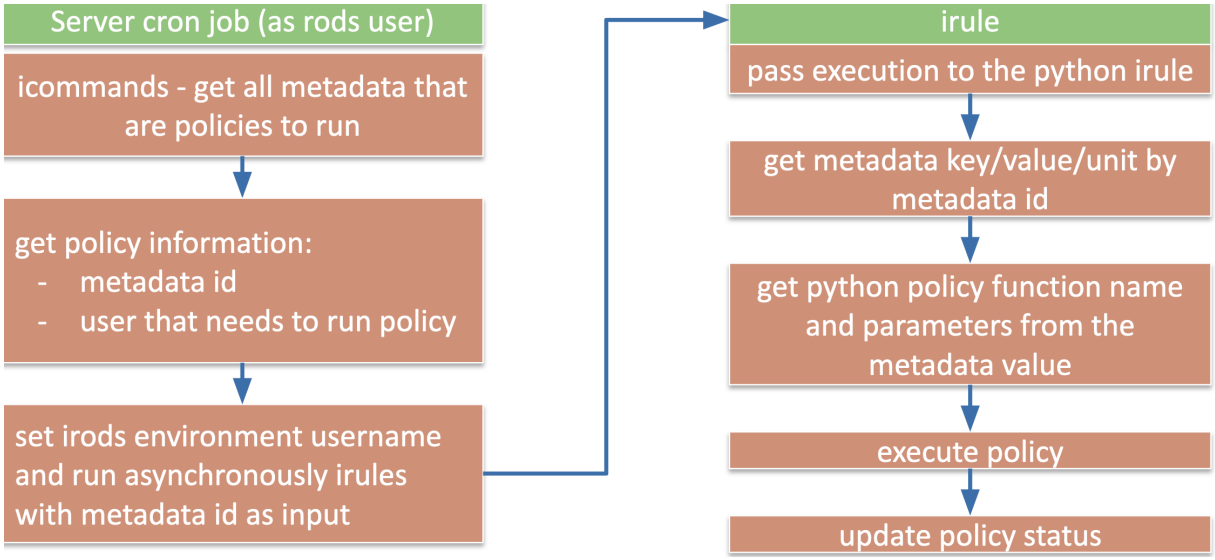

Figure 5. Shows a schematic view of the policy engine's workflow.

The Python code shown above could in principle allow users to inject malicious code into the JSON string. This will trigger server-side processes to operate on the data residing in the RUG RDMS. To avoid such an unauthorised operation, a policy validation mechanism is implemented in the mid-tier layer of the RUG RDMS. When a policy metadata is added to an object, the system generates an unique non-reversible hash value associated with the newly introduced policy. These values are stored in the mid-tier database and are used as guards to validate the user-requested policies. When a policy tag is added by a user, its hash value is generated and compared to the ones stored in the database. The process is triggered only when there is a match between the stored and computed hash values.

METADATA TEMPLATES

In recent years, researchers' capability of gathering data have increased a great deal. Descriptive metadata is used by many fields and stakeholders, and has evolved from different communities with diverse discipline-specific objectives and backgrounds. Metadata can be seen as a formally structured and documented collection of information about data that reveals minimally what is in the data, where the data originated from, who produced them, when they were produced and modified, why they were produced, and how the data can be obtained. Metadata characterizing a dataset generally contains all the attributes that are relevant to the specific research domain, but might as well contain general measurement attributes such as number of instances, dataset dimensionality, or circumstantial information. Besides describing the data being stored, metadata can also be used as triggers to configure and customize automatic workflows.

In the process of setting up RUG RDMS we have received several requests for metadata templates in different research domains, ranging from social sciences to archeology or microbiology. There is no one-size-fits-all solution, as each research domain has specific metadata attributes, and even within the same research domain there are large variations. By allowing users to define their own metadata schemas, we leave the definition of the metadata, which requires domain specific knowledge, to the domain experts. Metadata schemas define the general format of the metadata, that is, which elements are allowed, the number and order of their occurrences, of which data type they should be, which elements are required, etc. Because the technical skill of the user is also extremely diverse, RUG RDMS supports the use of JSON Schema templates together with a metadata template builder (Figure 6).

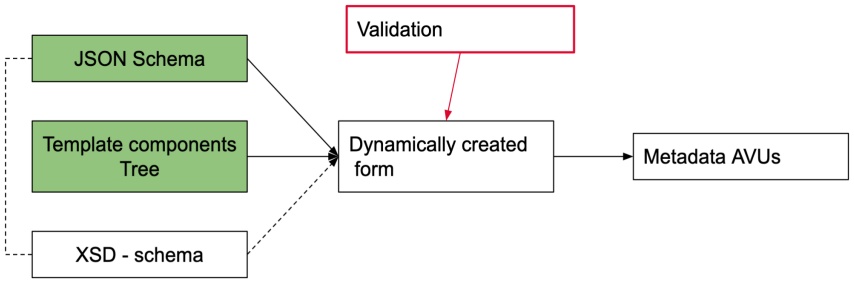

Figure 6. Working principle of the metadata templates in the RUG RDMS system.

This allows users to define their own tree-like types and type restrictions. These types of metadata attributes can be freely combined to build domain, and sometimes even user specific metadata templates. The definition of the metadata template tree of components is stored in the PostgreSQL mid-tier database. Based on the template definition an input form is dynamically built. The form handles the validation of the input values based on the template definition. When data has been validated, the form entries are serialized and stored as key-value-unit sets in iRODS. The same form is used to update the values of the metadata template fields.

THE AUDITING MODULE

The auditing module is using the iRODS audit plugin to store data operations in an elasticsearch database. This module contains functions to query operations on collections and data objects such as uploading, removing, moving, and downloading data. Figure 7 displays the auditing pipeline for the RUG RDMS system. The iRODS audit rule engine plugin can emit a single AMQP message for every policy enforcement point (PEP) encountered by the iRODS server which gets further pushed to the RabbitMQ service. AMQP stands for Advanced Message Queuing Protocol and it is an open standard application layer protocol for message-oriented middleware. The AMQP message emitted by the iRODS plugin has the information related to that particular operation, including username, filepath, filesize, etc. RabbitMQ is a message-queueing software also known as a message broker or queue manager. It is basically software where queues are defined, to which applications connect in order to transfer a message or messages. It also serves as a temporary location for messages by storing them while the destination application is busy or not connected. The messages in RabbitMQ are pulled by the ELK stack and can be displayed to the user. ELK is the acronym for three open source projects: Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like Elasticsearch. Logstash can dynamically unify data from disparate sources and normalize the data into several destinations. Any type of event can be enriched and transformed with a broad array of input, filter, and output plugins, with many native codecs further simplifying the ingestion process. Kibana lets users visualize the data in Elasticsearch with charts and graphs.

Figure 7. Auditing pipeline for the RUG RDMS system.

The data stored in Elasticsearch contains information on the PEPs being triggered in the iRODS backend. There are custom queries to answer questions such as: who, when, and what operation was performed and on which data objects. To prevent the Elasticsearch database from rapidly growing out of control, configuring a regular expression for filtering becomes necessary so only the output of desired PEPs is retained. The exact regular expression used is:

```
"(^audit_pep_api_((?!gen_query|auth_response|auth_request).)*_(pre|post)$|audit_pep_api_auth_.*_except)"
```

This states that the system is interested in logging activity from the api plugin (while specifically excluding authentication PEPs and general queries), but specifically including any authentication errors that may occur. Additionally, empty fields are filtered using a Ruby script in the Logstash configuration. We have observed that using these filters, around 95% of the messages are excluded from being stored in Elasticsearch. The filters do not affect the data operation queries.

In order to comply with the General Data Protection Regulation (GDPR) [17] and funding agencies requirements, the system is required in some cases to store the data including its audit trail for up to ten years. Despite the fact that the amount of data being stored in the ELK stack has been considerably reduced, the amount of data from the audit trail is still large. As a consequence, Index-Lifecycle-Management (ILM) policies for managing the indices have been implemented. In a cluster under this architecture, different types of nodes can be configured to balance performance and capacity.

The exact Elasticsearch architecture is still a work in progress. We are considering a hot-warm-cold architecture, where the exact number of shards is yet to be decided. The four stages in the index lifecycle are:

- Hot—the index is actively being updated and queried.
- Warm—the index is no longer being updated, but is still being queried.
- Cold—the index is no longer being updated and is seldom queried. The information still needs to be searchable, but it's okay if those queries are slower.
- Delete—the index is no longer needed and can safely be deleted.

Additionally, we have considered using Snapshot-Lifecycle-Management policies to back up iRODS indexes periodically in a snapshot repository.

CONCLUSION

The RUG RDMS is a system for the research data storage and management. It is designed and built to meet the requirements of our local researchers and research groups. Using iRODS as a storage backend it delivers flexible and robust infrastructure. There are many challenges ahead. At the moment there are approximately seven pilot use cases employing RUG RDMS. The range of the projects is wide, from collaboration with library and publishing departments to tape storage of a large amount of data for astronomy. The RUG RDMS development team is looking forward to helping researchers at the university and working hard to deliver the system to the scientific community as an open-source project.

ACKNOWLEDGMENTS

The authors would like to thank researchers from various departments for their involvement and valuable input during the requirement engineering and test phase of the system. Furthermore, we want to thank the project's steering board, our colleagues from the Center for Information Technology, Research Data Office, University of Groningen Library and CMS team, acknowledge everyone who is and was involved in the project and had helped us to design and test the system.

REFERENCES

[1] Agarwal, R. and Dhar, V.: Big data, data science, and analytics: The opportunity and challenge for IS research. Information Systems Research. Vol. 25, issue 3, 443--448 (2014)

[2] Wilkinson, Mark D et al.: The FAIR Guiding Principles for scientific data management and stewardship. Scientific data vol. 3 160018. (2016)

[3] Ton Smeele and Lazlo Westerhof: Using iRODS to manage, share and publish research data: Yoda, iRODS UGM 2018 proceedings (2018)

[4] YoDa – a research data management service, **https://www.uu.nl/en/research/yoda**

[5] The University of Groningen, **https://www.rug.nl/**

[6] The University Medical Center Groningen (UMCG), **https://www.umcg.nl/**

[7] iRODS, **https://irods.org/**

[8] iRODS documentation. **https://docs.irods.org/**

[9] NGINX, **https://www.nginx.com/**

[10] Django, **https://www.djangoproject.com/**

[11] Cesare Pautasso, Olaf Zimmermann, Frank Leymann: RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision, 17th International World Wide Web Conference (2008)

[12] Eclipse Jersey, **https://eclipse-ee4j.github.io/jersey/**

[13] Apache Tomcat, **http://tomcat.apache.org/**

[14] Davrods - An Apache WebDAV interface to iRODS, **https://github.com/UtrechtUniversity/davrods**

[15] NFSRODS, **https://github.com/irods/irods_client_nfsrods**

[16] iCommands, **https://docs.irods.org/master/icommands/user/**

[17] General Data Protection Regulation, European Commision. April 27, 2016. url: **https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679**