EUDAT

D9.8: Final Report on Provision of Staging Support

Author(s)	Peter Krauß, Michal Jankowski, John A. Kennedy
Status	Final
Version	v1.0
Date	06/02/2018

EUDAT receives funding from the European Union's Horizon 2020 programme - DG CONNECT e-Infrastructures. Contract No. 654065

Document identifier: EUDAT2020-DEL-WP9-D9.8	
Deliverable lead	KIT
Related work package	WP9
Author(s)	Peter Krauß, Michal Jankowski, John A. Kennedy
Contributor(s)	
Due date	28/02/2018
Actual submission date	06/02/2018
Reviewed by	Jos van Wezel, Shaun de Witt
Approved by	PMO
Dissemination level	PUBLIC
Website	www.eudat.eu
Call	H2020-EINFRA-2014-2
Project Number	654065
Start date of Project	01/03/2015
Duration	36 months
License	Creative Commons CC-BY 4.0
Keywords	Data staging, request scheduling, service architecture, HPSS, TSM, storage, archive, long-term preservation, tape storage

Copyright notice: This work is licensed under the Creative Commons CC-BY 4.0 licence. To view a copy of this licence, visit https://creativecommons.org/licenses/by/4.0.

Disclaimer: The content of the document herein is the sole responsibility of the publishers and it does not necessarily represent the views expressed by the European Commission or its services.

While the information contained in the document is believed to be accurate, the author(s) or any other participant in the EUDAT Consortium make no warranty of any kind with regard to this material including, but not limited to the implied warranties of merchantability and fitness for a particular purpose.

Neither the EUDAT Consortium nor any of its members, their officers, employees or agents shall be responsible or liable in negligence or otherwise howsoever in respect of any inaccuracy or omission herein.

Without derogating from the generality of the foregoing neither the EUDAT Consortium nor any of its members, their officers, employees or agents shall be liable for any direct or indirect or consequential loss or damage caused by or arising from any information advice or inaccuracy or omission herein.

TABLE OF CONTENT

EXECUTIVE SUMMARY 4
1. INTRODUCTION 5
1.1. Challenges 5
1.2. Scope of the Task 5
2. RELATED WORK 6
2.1. TReqS 6
2.2. ERADAT and DataCarousel 6
3. REQUEST SCHEDULING 7
3.1. FIFO-based Request Scheduling 7
3.2. Throughput-based Request Scheduling 7
3.3. QoS-based Request Scheduling 8
4. RECOMMENDATIONS 10
4.1. File Sizes 10
4.2. Massive Pre-Staging 10
4.3. Large Directories 10
5. SERVICE ARCHITECTURE 12
5.1. Previous Work 12
5.2. Architecture Extensions 12
5.2.1. Request Queuing 13
5.2.2. Request Scheduling 13
5.3. Exemplary Request Routing 13
5.4. Implementation 13
5.5. Integration in EUDAT 14
6. FURTHER ENHANCEMENTS 15
7. CONCLUSION 16
ANNEX A. GLOSSARY 17
ANNEX B. REFERENCES 18

LIST OF FIGURES

Figure 1: Performance comparison of different scheduling strategies 8
Figure 2: Comparison of different scheduling strategies with regards to QoS 9
Figure 3: Evaluation of the duration of a directory listing 11
Figure 4: Architecture overview 12

EXECUTIVE SUMMARY

In this document we present an extension to the service architecture proposed in the EUDAT deliverable D9.4 [EUDAT D9.4] that enables a user to get easy access to different storage systems such as IBM TSM (see glossary for abbreviations) or HPSS. Hereby we focus on features for integrity checking based on check-summing and scheduling of staging requests to achieve the best possible quality of service.

The architecture provides a front-end API following the REST paradigm over HTTP. Beside the features presented in the previous deliverable, the architecture is now extended to schedule and reorder slow, asynchronous operations based on predefined strategies.

The architecture was prototypically implemented to work within the B2SAFE context and was successfully tested in defined testing environments.

1. INTRODUCTION

Scientific and cultural organizations, international collaborations and projects have a need to preserve and maintain access to large volumes of digital data for several decades. Existing systems which support these requirements cover the range from simple databases at libraries, to complex multi-tier software environments developed by large-scale scientific communities. All communities can see an increasing volume of digital data that must be stored efficiently and economically. Today, this can involve a dynamically managed combination of storage on magnetic disks and on magnetic tapes. While often accessed and changed data can be held on hard disks, rarely changing data can efficiently be stored on tape drives: Tape storage can easily be expanded by increasing the number of tapes and tape drives and is comparably cheap during operation due to its high energy efficiency. Further tape drives can be replaced in case of mechanical failure and are usually several backwards compatible for generations of tape technology. However new challenges arise from the asynchronous character of accessing data stored on tape as described in the next section.

1.1. Challenges

The EUDAT project is developing an infrastructure for secure and reliable archival storage based on such storage that functions as a uniform platform for multiple scientific domains and international projects. In this context, access to the actual storage in a data center is enabled through an abstracted bit-preservation layer that offers features selected for long-term storage such as special metadata tags. To support this functionality, complex storage systems such as the High Performance Storage System (HPSS) by the HPSS Collaboration or IBM Spectrum Protect for Space Management (SPSM, formerly known as Tivoli Storage Manager/TSM) are often used. Usually these storage systems are hard to setup and to operate and are targeted at specially educated administrative staff. Commonly, each storage systems offers its own, proprietary programming interface (API) to access data and metadata and to control the system. With the collaborations growing, and more and more data centers becoming involved, the need to support multiple storage systems arises.

In addition to the complex user interfaces, there are other problems to cope with: the aforementioned storage systems are usually not designed to interface with end users but instead with intermediate service users and administrators. As a result, requests to files and metadata are purely prioritized on technical properties, like the amount and the size of the requested files or simply in a first come/first serve (FIFO) way. Ignoring the physical position of the data on tapes will lead to bad reading performance due to the overhead of winding and loading tapes. Further, tape systems are often funded by several different user groups the need for a higher level request scheduling implementing a fair usage for different groups based on non-technical information with detailed monitoring and metering arises. In addition, tape system owners might want to distribute requests and thus the mechanical load on the tape hardware in a way that maximizes the lifetime of the devices in exchange for a reduced but still acceptable quality of service.

1.2. Scope of the Task

In task 9.3 of EUDAT we developed an architecture that helps to solve the aforementioned problems by putting an middleware between the back-end storage system and the actual users. While the middleware can be used by versed humans through scripting for example it is primarily targeted at being used by machines in form of the API. The main structure of this architecture is subject to the previous deliverable D9.4 “Final Report on Efficient Integrity Checking” [EUDAT D9.4] and will thus be covered only briefly in this document. However the added components to enable new functionalities are presented in detail.

In addition, we will discuss several aspects that arise while offering tape storage to different user groups on a large scale. Our experiences show that common users are not always aware of all the technical and functional implications of asynchronous storage and thus additional guidance is required. For storage providers it is crucial to cope with this in advance to avoid later intervention that might lead to suffering service quality at later stages.

2. RELATED WORK

2.1. TReqS

At the Institut national de physique nucléaire et de physique des particules[1] a software called Tape Request Scheduler[2] is developed. TReqS is a scheduler for file requests to a High Performance Storage System system. It interacts between a client asking for HPSS files and an HPSS instance through the HPSS API. The initial aim of TReqS is to enable HPSS for nuclear physics experiments needs.

TReqS is based on a client-server architecture with a REST-based front-end and a binary protocol binding on the back-end. The core functionality aims to enable efficient file staging from tape to online storage to make the data available for further use. The methods used are based on organizing file stage requests in queues and reordering them based on the physical location of the data. In our task we reused some of these ideas but extended them in several ways:

While we keep the idea of a scheduling queue, we do not use a single waiting queue per drive but three. The three queues are prioritized and are meant for administrative tasks (highest priority), user tasks (medium priority) and automated background tasks (lowest priority). Another aspect is that we want to enable this system for different tape back-ends such as the tape storage of IBM[3]. To enable that, we added an abstraction layer to the back-end as described in the previous deliverable D9.4 “Final Report on Efficient Integrity Checking”.

2.2. ERADAT and DataCarousel

ERADAT and DataCarousel [TSO2011] are other tools targeting to provide better staging support for tape systems. Both tools were developed at the Brookhaven National Laboratory[4]. While DataCarousel does not implement any logic and solely acts as an user interface, it is not of much use for EUDAT. However the second component is interesting for the project since it implements several functions to handle staging requests in bulk. ERADAT offers features such as resource sharing and guaranteed resources, request reordering based on different metrics as well as over-subscription methods. The latter mechanism is used to share allocated tape drives between users to minimize overhead due to tape loads or winding even if it would violate an otherwise strictly enforced fair-share policy. This mechanism seems promising and could improve the service quality. It is planned, however, not yet part of our proposed infrastructure.

[1] L’Institut national de physique nucléaire et de physique des particules. 2017. IN2P3. Available at: http://www.in2p3.fr/

[2] IN2P3. 2017. TReqS project webpage. Available at: http://www.in2p3.fr/https://forge.in2p3.fr/projects/treqs-project

[3] IBM. 2017. IBM Tape Storage. Available at: https://www.ibm.com/storage/tape

[4] Brookhaven National Laboratory. 2017. BNL webpage. Available at: https://www.bnl.gov/world/

3. REQUEST SCHEDULING

For task 9.3.2 we wanted to reuse this architecture and extend it. To enable efficient staging support, it is necessary to understand what possible optimization criteria are and which aspects can be optimized when it comes to dealing with tape technology.

While data on online storage, such as hard disks and solid state drives, can be read with a close to negligible overhead, this is not true for tape-based storage. For hard drives, data is physically aligned in two dimensions on several magnetic layers and read by a reading head that can move in two dimensions. For solid state drives, data is stored on a flash memory and can directly be addressed electronically. However for tape drives, the data on cartridges is physically aligned in one dimensional, linear blocks on a magnetic tape that can only be read sequentially. This alignment allows tape drives to read data sequentially at high speed but leads to severe performance loss in case of non-sequentially accessed data: with every reading operation a winding operation is necessary to move the reading head to the desired position. In addition, data that is spanned across multiple tapes, requires the tape system to load the different tapes which yields further performance loss. While the reading operation of the data itself depends on the reading performance of the tape drive itself, the reducing the overhead of winding and tape loads is desirable.

Beside suitable scheduling of requests to improve the performance of a tape system, the optimization of the data and its layout on tape is necessary. Usually large tape systems handle data layout on their own with none or only very few options for an operator to influence the behavior. It is however up to the user to avoid certain mistakes that directly affect the performance.

In the next sections, we will describe the three most common scheduling principles and point out some mistakes that should be avoided to achieve the best possible performance.

3.1. FIFO-based Request Scheduling

Processing requests the way simply in that order as they arrive at the system is called first-in-first-out (FIFO) scheduling. This is the most trivial way to process a set of requests, since no queuing is required. Without further sorting, this method yields the bad performance due to large winding and loading overhead as request arrive and are thus are de facto processed in a random order.

It is however important to offer this kind of behavior to allow users to create their own scheduling methods. A user can sort the requests in a for his use-cases suitable way based on metrics and data sources the scheduling software would not have implemented or cannot access. The FIFO mechanism then makes sure, the user-defined order is not changed.

3.2. Throughput-based Request Scheduling

The most common scheduling mechanism aims to scheduling requests in a way that data throughput is maximized. This is achieved by minimizing winding and loading times by sorting pending requests based on tape, then based on their physical layout on the tapes. Our prototype further takes the currently loaded cartridges into account and prioritizes requests for files on loaded cartridges over requests for files on currently not loaded cartridges. While HPSS and TSM provide the physical location of files using the API, some tape systems might not. For those systems, it is a good estimation to sort files based on their modification timestamp. In many cases this reflects the creation time of a file and is pseudo-proportional to the file’s location on tape: older files are usually at the beginning of a tape, newer ones are stored at the end.

While this mechanism is also used by TReqS and ERADAT, it does however only maximize the current throughput of the tape system. Other metrics are completely ignored resulting in probably unfair treatment of different users. TReqS offers no solution to this since it is targeted to be used by service accounts and management personnel. TReqS leaves it to the tape system’s operator to reserve tape drives exclusively to different user groups to enable fair usage. In many cases this leads to drives idling.

ERADAT copes with this by sharing idle drives between all active users independently from previous association with a certain user group. This behavior is referred to as “resource sharing”. Both approaches are suitable for environments where there is no other source of information available to the scheduling system beside the tape system’s metadata.

Since our architecture is also incorporating its own user database we can also use that information to allow more fine-grained scheduling policies, for example to limit bandwidth for single users and user groups.

3.3. QoS-based Request Scheduling

While throughput might be the obvious optimization criteria, depending on the use case, it might not be necessary to answer user requests as fast as possible but within a certain deadline. For these use cases, requests can be collected over a much longer period of time. Using this technique, the set of requests that can is optimized is much larger resulting in less tape loads, less winding, and thus an overall better quality of service for the whole user base at the cost of single users having to wait longer for their requests to be processed. We call this scheduling mechanism Quality-of-Service-scheduling (QoS). Hereby the quality of service is measured in the percentage of requests that are delivered successfully, which means within a predefined period of time (e.g. one hour). The purpose of this parameter is to quantify the responsiveness of the system and the observation of this parameter is directly relevant to our problem.

Measurements show that in many cases, QoS-based scheduling yields shorter average waiting times (“time to first byte”) and higher QoS success rates but a weaker overall performance compared to FIFO and throughput-based scheduling.

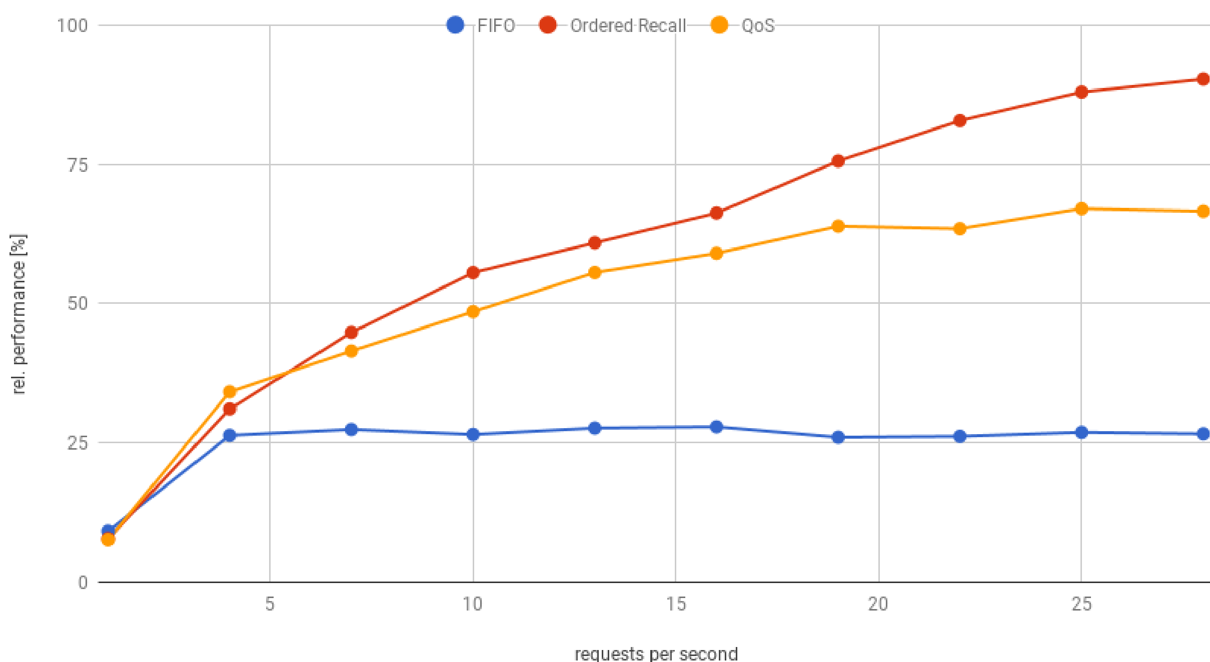

Figure 1: Performance comparison of different scheduling strategies

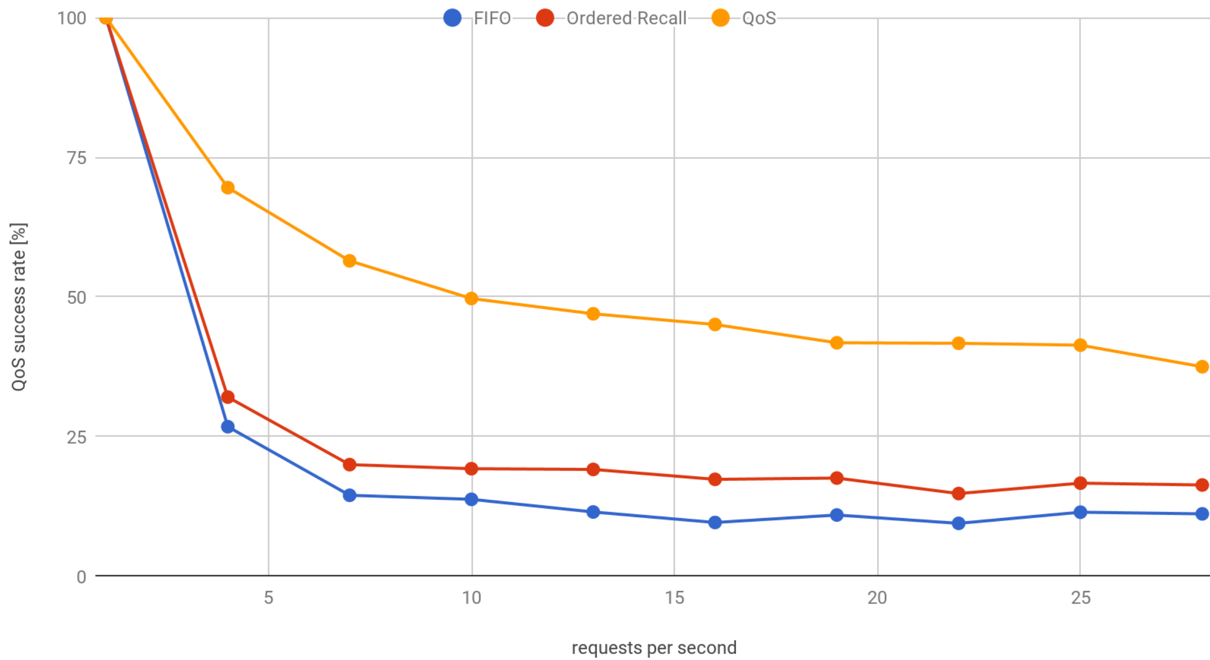

Figure 2: Comparison of different scheduling strategies with regards to QoS

The testing was done using the HPSS installation at the Karlsruhe Institute for Technology (KIT) exclusively. We successively increased the amount of concurrent reading requests for files stored only on tape and measured the reading performance in MB/s as well as the amount of successfully answered requests after a configured timeout for each of the three scheduling mechanisms (FIFO, Ordered Recall/throughput optimized, QoS-optimized). The results show a quick saturation with regards to performance for the FIFO-based scheduling. While the QoS-based scheduling can keep up with regards to performance, throughput-optimized scheduling achieves better results for larger amounts of requests. However, the QoS success rate is much higher for the QoS-scheduler compared to the other two algorithms. This is the expected behavior.

4. RECOMMENDATIONS

Beside choosing a suitable strategy for request handling, it is also important to avoid some things if possible.

4.1. File Sizes

Tape storage systems do not work well with small files. While the writing process is buffered its performance does not suffer as much as when it comes to the retrieval of a large number of small files. Due to the mechanical characteristics of tape media explained above, long delays between file retrieval will occur. Even with proper scheduling and a perfect data layout, one should try to avoid storing small files.

Instead, we recommend to store small files either exclusively on disks or in a larger single file using container file formats such as the TAR file format [TAR2016]. In general a file size of around 100 gigabytes should be targeted if feasible. Larger files are prone to transfer errors, can be hard to work with and their retrieval takes excessively long increasing the risk of transfer interruptions. It should be noted, that the TAR format has some limitations with regards to the maximum number of member files or the size of a single TAR archive.

Some tape systems encourage the usage of their own container formats, such as HPSS's HTAR. Beside combining many small files to one larger file, it also does automatic indexing to speed up retrieval of member files.

4.2. Massive Pre-Staging

When data on tape is requested, the tape library first moves the data to an attached disk cache for further accessing. One should keep in mind, that this cache is limited and managed by the tape library. A file that was moved to cache will get purged again as soon as the cache reaches its limit. As the cache is shared among other users, it is not always visible to the requesting user, for how long his data will stay cached. Further, if the data read is larger than the cache, it will be purged immediately.

Both situations result in a performance penalty as data is read twice from tape. Instead it is recommended that global disk space is used to stage large data volumes instead of the tape system's disk cache.

4.3. Large Directories

Even though all tape systems nowadays come with an attached database to manage metadata, one of the most expensive operations are verbose directory listings (e.g. through GNU coreutils' 'ls -la').

Since file listing is one of the most common operations, we recommend limit the amount of files per directory to the lower thousands as the listing operation showed non-linear scaling with the amount of files in a directory:

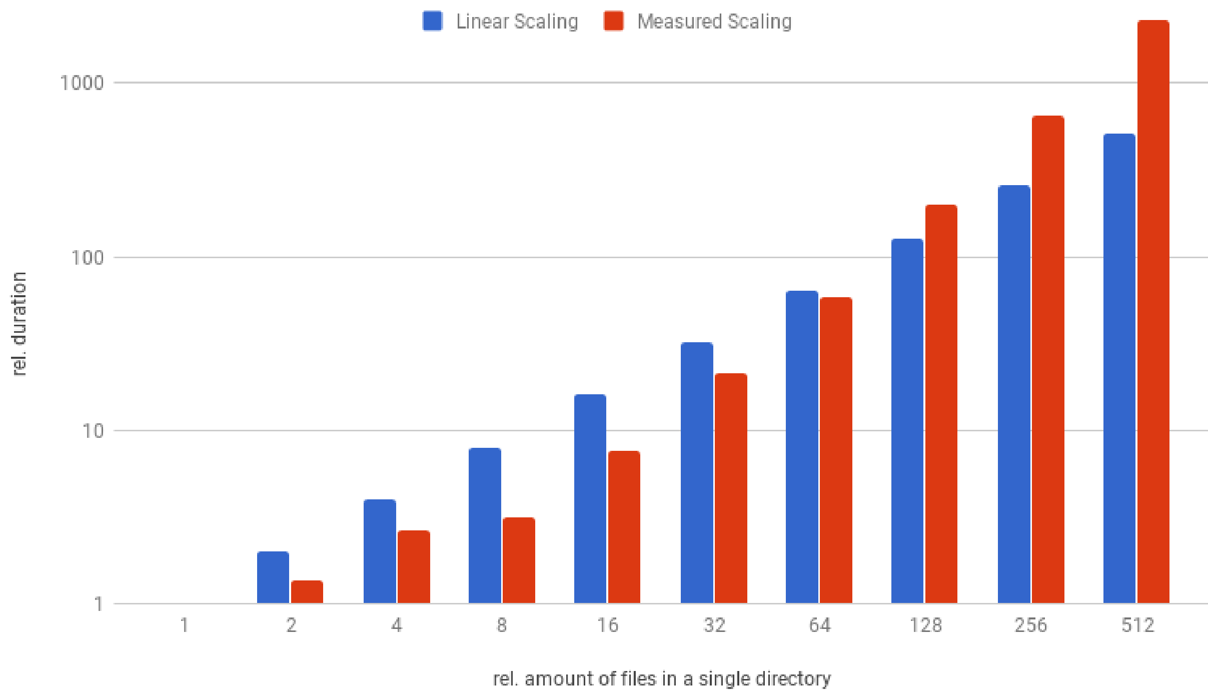

Figure 3: Evaluation of the duration of a directory listing

5. SERVICE ARCHITECTURE

5.1. Previous Work

To enable efficient staging support for tape systems we extended our architecture developed in WP9.3, presented in D9.4. "Final Report on Efficient Integrity Checking" [EUDAT D9.4]. While the core of the architecture does not change, some features have been added (see following section below).

The old architecture was based on five main components:

- Front-end web server to enable load-balancing and to terminate SSL encryption
- a component based on JWT (JSON Web Tokens [RFC7519]) and plug-able user database for authentication and authorization
- a middleware to retrieve and pre-process a user's requests
- a global message queue to collect and distribute requests from the middleware
- a set of distributed workers that are capable of validating the authentication context and of processing the requests from the message queue using back-end specific drivers

With these components, we enabled the following features:

- REST-interface for client-side communication
- API-bindings to tape back-ends for metadata access and administrative operations
- Periodic check-summing based on a policy as described in D9.4
- Authentication and authorization based on an internal database

5.2. Architecture Extensions

To enable request scheduling, we extended the architecture to incorporate two new components. First, we added requests queuing, a component that not directly sends incoming requests to the back-end, but instead acts as an ordered buffer. Second, a scheduler is added, that follows a user-defined strategy. For now, this scheduler implements all three scheduling mechanisms as described in Section 3.

The extended architecture is shown in the following graphic:

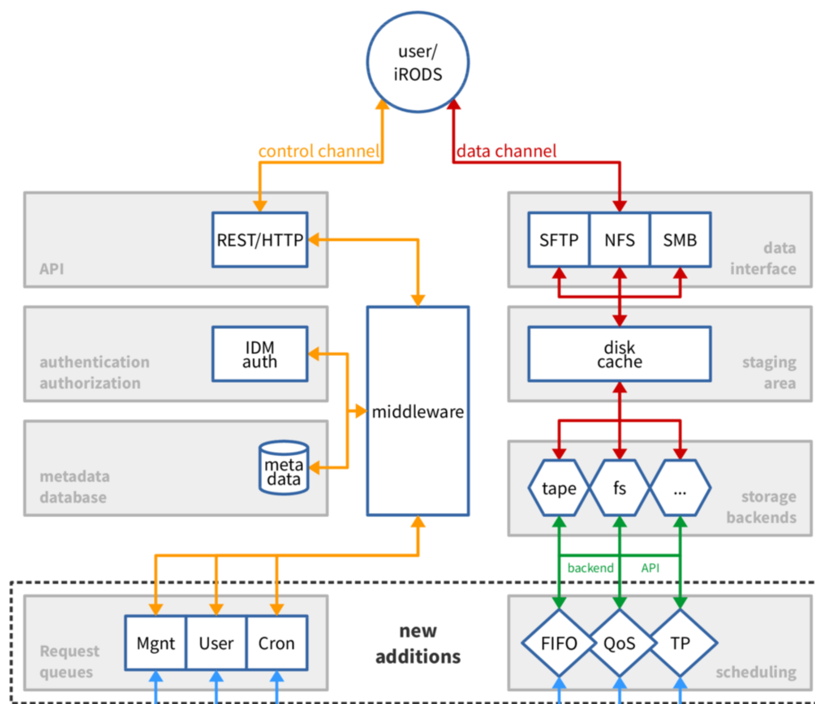

Figure 4: Architecture overview

5.2.1. Request Queuing

In our architecture, we decided to only enqueue requests that are related to files on tape, because access to the metadata database does not significantly suffer from multiple concurrent accesses rendering scheduling unnecessary. Requests for files that are already on cache are also considered to be successful and are thus immediately discarded.

Further, we added two more queues that a request could be added to: A high prioritized queue for administrative tasks and a low prioritized queue for automatic performed tasks, such as e.g. periodic check-summing. While it would also be possible to enable different prioritization of requests by adding an additional property to the request objects and then have them evaluated by the scheduler, we found that it is much more transparent and more easy to implement to have simply three separate queues that are then processed according to their priorities: if the administration queue contains items, those are always executed first. Then, if that queue is empty, the user requests are send to the tape system. Finally, if the other two queues do not contain any more items, those are executed.

5.2.2. Request Scheduling

In Section 3, we described three scheduling strategies. In our prototype, we use all of these scheduling mechanisms for the different queues: we apply a scheduler following the FIFO-strategy to the administrative queue, since we assume administrative requests are already ordered by the operator. In contrast, the user queue is scheduled following the QoS-focused strategy to optimize the requests for all users in total. Lastly, the third queue for automatic tasks is reordered following the high-throughput strategy. We have chosen that strategy to run as many requests as possible in a short period of time, since the requests are often interrupted by users due to the low prioritization.

5.3. Exemplary Request Routing

To further explain how the system handles requests, one could track an exemplary request through the system:

As soon as a REST request arrives at the system’s head node, the request content is parsed. In a first step, the request is passed to the authentication and authorization component. This component ensures the validity of the request by either checking login credentials or verifying the authentication token in the header. Next, the request body is analysed: does the request contain an action that is required to be scheduled on tape system level? If so, the request is added to one of the three waiting queues and a change notification is sent to the scheduler. If not, the request is added to the internal message system. The distributed worker farm can now receive this message and further process it on one if its workers. If a response can immediately be created, e.g. the request is a simple database query and a worker is available, an response object is sent back to the user. In case the request cannot be processed immediately or it’s processing takes too long, a task id is returned to the user. The requests response will then be hold back and delivered as soon as it is available and the user’s client is ready to accept it. The latter can be done in two ways, by either client-side polling using the task id or server-side pushing using WebSockets [RFC6455].

As mentioned above, some requests are added to the scheduling component due to them adding load to the tape drives. The scheduler is informed each time a request is added to one of the waiting queues and the configured strategy is applied. This makes sure the queues are always sorted in the desired way. For the user, those requests are treated like long running requests and thus the response object will be a task id to be used for polling.

5.4. Implementation

During the duration of the project, we created a functional prototype as a reference implementation of the described architecture. Since this is only of prototypical quality, it cannot be used for production but only for testing and proof-of-concept purposes.

Currently the implementation is based on the following technologies:

- Python (version 3) as the main programming language. We chose Python for development due to its large standard library, the simplicity of the language and the wide availability on different systems. Further, managed memory and bindings to all relevant system components lead to rapid development.
- For HTTP we rely on WSGI [PEP333] enabled by the commonly used Werkzeug library[5], implemented in the Flask microframework[6].
- In addition we make use of several Flask extensions, like Flask-RESTful and Flask-JWT to enable the corresponding technologies. A description of the used extensions can be found on the Flask project website available at http://flask.pocoo.org/extensions/.
- The internal database is driven by SQLAlchemy[7] allowing to swap out the underlying database system by implementing an abstraction layer (object-relational mapping, ORM).
- Message passing between the different components is done using the Advanced Message Queuing Protocol (AMQP) [AMQP2011], enabled by python-pika[8].
- The bindings to the tape library are enabled using the HPSS's binary system interfaces

This technology stack enables the operator to swap out critical components and reuse already existing ones to minimize the effort to use our software. For reference though, a currently running installation at the KIT uses MySQL[9] as database system, NGINX[10] as webserver and RabbitMQ[11] as message broker.

5.5. Integration in EUDAT

Our software was designed to be usable standalone as well as part of the EUDAT B2SAFE component. With B2SAFE using iRODS, we propose to connect the components using iRODS micro services in conjunction with a suitable resource plug-in, such as the iRODS HPSS Resource Plugin[12] for HPSS. This allows a the system to have access to data in a direct way through the resource plug-in while enabling high-level functions using our software, such as the check-summing (described in deliverable D9.4) and the scheduling strategies as described above.

Our code is already part of the EUDAT B2SAFE releases and we work on continuing to update it. In accordance with the project proposal we will use EUDAT CDI for reporting, which is however not yet conclusively done.

[5] Armin Ronacher. 2017. The Werkzeug project webpage. Available at: http://werkzeug.pocoo.org/
[6] Armin Ronacher. 2017. The Flask project webpage. Available at: http://flask.pocoo.org/
[7] SQLAlchemy authors and contributors. 2017. SQLAlchemy project website. Avialable at: https://www.sqlalchemy.org/
[8] Python-Pika. 2017. Project website. Available at: https://github.com/pika/pika
[9] Oracle Corporation. 2017. The MySQL database website. Available at: https://www.mysql.com/
[10] NGINX Software Inc. 2017. NGINX webserver website. Available at: https://nginx.org/en/
[11] Pivotal Software. 2017. "RabbitMQ – Messaging that just works". Available at: https://www.rabbitmq.com/
[12] The University of North Carolina at Chapel Hill. 2017. iRODS Resource Plugin on GitHub. Available at: https://github.com/irods/irods_resource_plugin_hpss

Copyright © The EUDAT2020 Consortium PUBLIC

6. FURTHER ENHANCEMENTS

During development, we found some aspects to further improve the software component described in this document. While our current implementation delivers good results regarding the check-summing and staging of files, we believe that adding support for ordering write requests and maybe an automatic repacking of file sets could be helpful for users. Currently it is up to the user to not store a large amount of small files in a single directory (see SEC[mistakes to avoid]). We think that our component could be extended to detect such cases and reorganize files in way that is more suitable for tape drives completely transparent for the user.

Further, we are aware of corner cases in the scheduling component that cause a so-called livelock[13]: a user's requests might never be processed if the scheduler decides to constantly favor other requests over those of that user. While these cases are rarely happen in real application, we still see room for improvement. We propose to research other scheduling algorithms and to improve the priority system to avoid such cases.

Additionally, the system's design can be used for any kind of asynchronous storage. During the development, the idea arose to not only limit the back-end plug-ins to tape storage but also allow developers to further add connectors for web storage technologies, such as S3 and Swift. However due to the lag of time and the completely different scope, we did not further work on this subject.

[13] Wikipedia Page for livelock. 2017. Available at: https://en.wikipedia.org/wiki/Deadlock#Livelock

7. CONCLUSION

Using our prototypical implementation, we can show that our architecture can be useful in many ways:

- We provide an easy to use interface to other services, end-users and administrators to high-level functions of tape libraries, that would otherwise not be available
- We implement an automatic mechanism for continuous fixity checking to ensure data integrity
- Our architecture greatly improves the quality of tape storage services by providing configurable request scheduling
- The system is based on widely used protocols making it easy to use and to run.
- The modular system makes it future-proof: outdated components can be replaced by more modern ones, while new ones can be added to implement new features.
- The back-end abstraction layer provides a standardized way to add new storage back-ends.

While all these aspects justify the efforts put into this technology, we still see some problems already mentioned in the last deliverable. Our service improves the usage of tape systems, but vendor-backed solutions would still be preferred. To achieve this, we need continued dialog with archive vendors to agree on uniform APIs and a standardized set of features. As for now, the decision of buying a tape library is still accompanied with the phenomenon of “vendor lock-in”.

Beside the technical and administrative implications of running a standalone service architecture as the one described in this document, policy discussions are necessary. While we can only provide a set of recommendations and tools to implement them, it becomes more and more clear that users need to be made aware of the tape technology itself. This does not only apply to end users such as researchers of different disciplines but also to operators of e.g. research facilities. We consider providing proper information about the technology, associated wording and definition as well as suitable workflows as the most important part when it comes to work with tape storage since it helps a user to decide where to store and how to organize his data.

Copyright © The EUDAT2020 Consortium PUBLIC

ANNEX A. GLOSSARY

Term	Explanation
AMQP	Advanced Message Queuing Protocol
API	Application program interface consisting of a set of routines, protocols, and tools for building software applications.
Authentication token	An authentication token is a digital signature generated from authentication information to identify a user or entity.
Checksum	small-size datum from a block of digital data for the purpose of detecting errors which may have been introduced during its transmission or storage.
DPM	Data Policy Manager
FIFO	First In First Out
FUSE	Filesystem in Userspace
HPSS	High-Performance Storage System
HSI	Hierarchical Storage Interface
HSM	Hierarchical Storage Manager
HTAR	HPSS Tape Archiver
HTTP	The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems.
IBM	International Business Machine (Corporation)
iCAT	iRODS Catalogue -metadata database for iRODS
iRODS	Integrated Rule-Oriented Data System: distributed data-management system for creating data grids, digital libraries, persistent archives, and real-time data systems.
KIT	Karlsruhe Institute for Technology
Metadata	Additional information added to a data record to describe the actual data. Commonly metadata includes information about the data's origin, the data's lifecycle or the data's content.
ORM	Object-Relational Mapping
PID	Persistent identifier associated to a digital object or to a whole collection
QoS	Quality of Service
Rest (API)	Architectural style: RESTful implementations make use of standards, such as HTTP, URI, JSON, and XML. Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs.
SPSM	Spectrum Protect for Space Management
SSL	Secure Socket Layer
TSM	Tivoli Storage Manager
WSGI	Webserver Gateway Interface

Copyright © The EUDAT2020 Consortium

ANNEX B. REFERENCES

[RFC7519] Internet Engineering Task Force RFC7519, “JSON Web Token (JWT)”, M. Jones, Microsoft, J. Bradley, Ping Identity, N. Sakimura, NRI, Maz 2015, https://tools.ietf.org/html/rfc7519

[EUDAT D9.4] EUDAT deliverable D9.4 “Final Report on Efficient Integrity Checking”, Peter Krauß, Marion Cadolle Bel, John A. Kennedy, Michal Jankowski, 2016

[TSO2011] Journal of Physics: Conference Series, volume 331/4 “Tape Storage Optimization at BNL” (42-45), David Yu and Jérôme Lauret, 2011

[TAR2016] Free Software Foundation, Inc. “GNU tar 1.29: Basic Tar Format”, Lionel Cons, Karl Berry, Olaf Bachmann, et. al., 2016, available at: https://www.gnu.org/software/tar/manual/html_node/Standard.html

[RFC6455] Internet Engineering Task Force (IETF), “The WebSocket Protocol”, I. Fette, A. Melnikov, 2011, available at: https://tools.ietf.org/html/rfc6455

[PEP333] Phillip J. Eby, “Python Web Server Gateway Interface v1.0”, 2010, available at: https://www.python.org/dev/peps/pep-0333/

[AMQP2011] AMQP authors, “AMQP Specifications”, 2011, available at: http://www.amqp.org/sites/amqp.org/files/amqp.pdf