2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Access to the DARIAH Bit Preservation Service for Humanities Research Data

Danah Tonne
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: danah.tonne@kit.edu

Jedrzej Rybicki
Forschungszentrum Jülich
Jülich, Germany

Stefan E. Funk
Göttingen State and
University Library
Göttingen, Germany

Peter Gietz
DAASI International
Tübingen, Germany

***Abstract*—Sustainable management of large amounts of research data is gaining in importance for research projects all over the world. The European project DARIAH aims to address this topic for the arts and humanities community.**

The DARIAH Bit Preservation, as a part of an archiving system for the arts and humanities, allows for a high performance, sustainable, and distributed storage of research data as basis of virtual research environments. A great challenge in designing such a service is to provide a standardized, consistent yet easy-to-use API for accessing the data that remains stable even if backend technology changes over time.

As a solution, this paper presents the RESTful API of the DARIAH Bit Preservation which includes an administrative extension, and which is secured by an Authentication and Authorization Infrastructure (AAI) based on SAML. An exemplary implementation illustrates that the API offers distributed access by usage of the HTTP protocol and is able to handle a high number of files. Data transfer rates of up to 45 MB/s were achieved for uploading large files in the local network.

I. Introduction

DARIAH (Digital Research Infrastructure for the Arts and Humanities) [1] is a European project on the ESFRI roadmap (European Strategy Forum On Research Infrastructures) [2]. It aims to support and enhance digitally-enabled research across the digital humanities community, wherefore a sustainable, distributed research infrastructure is built and maintained. An essential component of the infrastructure is a long-term storage service serving a wide variety of disciplines and accounting for their special requirements. Some examples of research endeavors DARIAH deals with on every-day basis are:

- A musicological project provides a complete overview of the work of one composer including scores, letters or recordings of an orchestra.
- A scholar working in Jewish studies analyzes an old Jewish graveyard. Therefore he has to deal with inscriptions which have to be translated and access maps or chronicles from different decades.
- A digitization project establishes a virtual library comprising of manuscripts that have been spread all over the world.
- An archaeologist virtually reconstructs buildings from their remains. The data from results of the excavations will be used to create 3D models of the landscapes and buildings.

All these different research initiatives have one thing in common: they rely on accessible, reliable long-term data storage. The concrete requirements posed by each of the projects can be, however, very different. The data from these projects and scholars differs in size (a few kilobytes for a text file containing a letter or several gigabytes for a film record of an opera), quantity (a few image files of a rare and valuable manuscript up to several millions of image files of a whole library) and type as there is a variety of different formats for text, image, audio, and movies. This list is non-exhaustive but the examples illustrate the heterogeneity of the data to be handled in the DARIAH project and the digital humanities in general. Humanities disciplines nowadays generate and analyze an increasing amount of data. At least parts of their research process therefore become more and more data-intensive and have to be supported by emerging research infrastructures.

Long-term storage as a basic service should be as generic as possible to satisfy the respective requirements of all humanities scholars, such as those described in the examples above. The DARIAH Bit Preservation (DBP) aims to design and implement a system for a sustainable, safe and persistent storage of research data. In this case the term Bit Preservation differs from the common understanding as it includes the following features:

- The humanities scholars store files long-term and only administrative meta data is handled.
- Heterogeneous data is handled, independent of size, format or content.
- Mostly create and read operations are performed, updating and deleting is possible but not used that often.
- Mechanisms to ensure data integrity are provided.
- A distributed system with both human and machine interface offering high-performance access is needed.
- The system is secured by an Authentication and Authorization Infrastructure (AAI).

Figure 1 depicts a high-level model of the DBP. The service is organized in layers and components to get a clear separation of functionalities. The undermost layer is formed by resources like disk storage, tape storage, and data bases. Storage resource federations are handled on this level to be completely hidden from the end-users. On top of the resources are generic Basic

Data Services, namely Storage Virtualization and the Meta Data Service which use the existing resources to store the data and the meta data. The Storage Virtualization abstracts the resources by providing a logical namespace and offers mechanisms to ensure the data integrity. The Meta Data Service stores administrative meta data for each file independent from the underlying data base. The Data Management and Repository Service in the layer above orchestrates the Basic Data Services and includes mechanisms for AAI. The Access Layer offers APIs for accessing the storage and parts of the AAI. Clients represent the topmost layer. User applications, a standalone web browser or DARIAH High Level Services offer more specific functionalities and use the DBP as a storage backend.

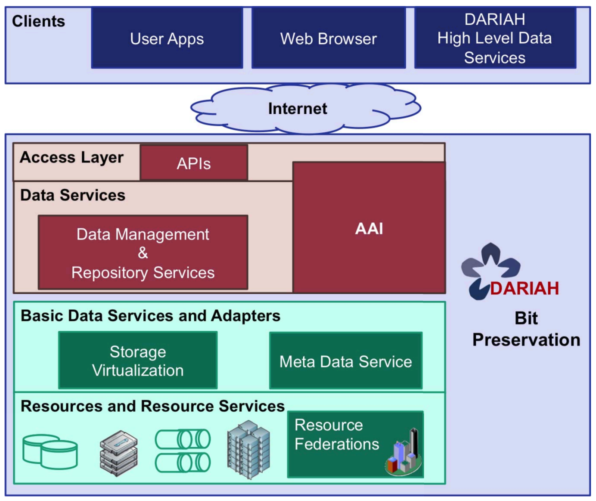


Fig. 1: Architectural overview of the DARIAH Bit Preservation service

II. Related Work

In order to build a sustainable storage infrastructure it is essential to provide a standardized API for the user access. This API has to be consistent, since every change needs to be adopted on the client-side. In terms of the DBP service the API is part of the Access Layer. Due to the usage in the arts and humanities community several additional requirements have to be met by such an API:

- The API should only cover basic functionalities for file handling.
- The API should be as easy-to-use as possible.
- The API should hide the complexity of the storage system.

In principle it is possible to use existing, well-established data-access APIs, for example Amazon Simple Storage Service (S3) [3], Cloud Data Management Interface (CDMI) [4], Merritt Storage Service Interface [5] or the DataOne API [6]. They all offer methods for uploading, updating, downloading, and deleting files as needed, but they allow users to directly access storage nodes of the underlying system and offer functionalities for replication. As stated in the requirements the complexity of the storage system and the responsibility for the replication has to be hidden from the user to provide an interface as easy-to-use as possible. Additionally the afore mentioned APIs offer functionalities which are not part of the DBP such as versioning, searching, listing of files, user rights management, persistent identifier (PID) [7] handling or format information. They are contrary to the limitation of basic file handling and will be offered by DARIAH High Level Services. Conceptually the DBP constitutes a basis for more specific services and therefore includes only the most elementary functions.

Data management systems like iRODS [8] or dCache [9] already offer most of the functionalities desired for the DBP. Their APIs can be theoretically adapted for the use in the Access Layer but they also reveal too much of the complexity inside the storage system and are too comprehensive. For example in the case of iRODS information about different storage resources or methods for replication are provided, in case of dCache a buffer can be used and altered by the client. Again, both systems and APIs offer functionalities which are not part of the DBP. Additionally they are not consistent enough: They can change in future development of the data management system or they can even be no longer maintained. A sustainable DBP service has to keep any changes hidden from the user by allowing access to the data in the storage via an consistent interface, independent from the storage backend used or changes in the backend's API.

The usage of an existing API therefore is not an option for the DBP as not all requirements are met. Thus a new API had to be specified, namely the Storage API.

III. Methods of the Storage API

This section provides a user-driven overview of the API, a more detailed description of the programming interface can be found in its publicly available specification [10].

The DARIAH Storage API is implemented in RESTful style [11] and based on the HTTP protocol. It particularly conforms with RFC 2616 [12] with regard to error codes and method semantics. The DBP naturally uses the notion of resources and representations of resources, therefore using a RESTful, resource-oriented architectural style is logical. The rationale behind the decision of using HTTP is the wide spread, the usability from every location and the easy implementation with many techniques.

By using HTTP the API is bounded to a list of methods with certain semantics. The methods POST, PUT, GET, HEAD, DELETE, and OPTIONS are used within DBP and the following description is mainly about defining the resources on which the methods can be called upon. To efficiently handle the heterogeneous data the notion BLOBs (Binary Large Objects) is used. The Storage API provides a set of methods to manipulate a "flat collection" of such BLOBs. It is not the intention to provide an abstraction of hierarchical collections as such abstractions will be managed by the DARIAH High Level Services (see figure 1) on top of the DBP.

Each BLOB has a unique identifier which is used in the URL to define a resource the user wants to manipulate. Therefore the URL scheme for the API has the form http(s)://<storage-service>/<objectID>. The *objectID* must be unique across the service to avoid conflicts between users. The simplest way of assuring this condition is to delegate the assignment of the *objectID* to the service and not the user herself. In the DBP the *objectID* for the new content is therefore assigned upon upload.

A new file is uploaded to the service by issuing a HTTP POST request on the service URL and providing the content in the message body. If the user is not authenticated or not authorized, the service will reject the request and respond with a proper error code (401 - Unauthorized). The authentication and authorization process will be described in section IV-B; for the following it is assumed that the user is authenticated and has the rights to upload content. In this case the DBP service will accept the content, generate a unique object identifier and respond with 201 - Created. The header of the response will also include a field *Location* pointing to the URL including the *objectID* generated where the content uploaded can be accessed by subsequent GET requests.

In order to modify content of an existing object in the storage, the user needs to follow the REST approach. She first has to retrieve content with a GET, modify its representation locally and upload the new version with a PUT to the same location. According to the HTTP specification PUT is an idempotent method i. e., a series of the same PUT operation on a given resource will have the same result as one PUT operation. This feature is often used to achieve the reliability of a RESTful interface. In case of an error the user might not receive a proper response from the server (201 - Created), she can then simply resend the same request to assure that the content is uploaded.

Removal of an object from the service is done in a very similar way to the updates. The user issues a DELETE request on a resource URL. The server responds with 204 - No Content. DELETE is also an idempotent method, thus it is possible to send multiple DELETE requests using the same *objectID*. The server will always respond with 204, even if the object has been removed in a prior attempt.

The two remaining HTTP methods are OPTIONS and HEAD. HEAD is very useful when dealing with larger data objects. It allows to get information about an object without retrieving its content. The response for a HEAD request is exactly the same as for a GET request with one significant difference: there is no response body, only the header is sent back. Thus the client can for instance first get the header of the file and check the last modification date or the size before deciding whether or not to retrieve the content. The other possible usage is to send a HEAD request to find out whether a content with a given *objectID* exists on the server. There is an almost unlimited number of valid URLs as specified by the API and the API will return well-defined responses. A request for a non-existing content, done either by GET or HEAD, will result in the response 404 - Not Found. The OPTIONS method offers a simple way of checking which methods are implemented for a given resource. For each proper URL of the service an OPTIONS request can be sent and the server responds with a list of methods allowed. For example executing an OPTIONS request on the location of an existing object will result in a list composed of OPTIONS, GET, HEAD, PUT, and DELETE. POST is only allowed on the service root URL.

IV. Extensions

As the Storage API is providing mere storage functionalities an additional API is needed to address Bit Preservation mechanisms provided by the DBP implementations. Additionally both APIs need to be secured as potentially sensitive data and information is provided.

A. Bit Preservation Admin API

The Bit Preservation Admin API is an administrative interface to determine selected Bit Preservation configurations and to get information about the data stored. This section gives a brief overview of the API, its complete specification is publicly available.

The Admin API is also based on the REST architectural style and the HTTP protocol to be compliant with the Storage API. The general form of request is http(s)://<storage-service>/admin/<objectID>, whereat admin specifies the entry point for the Admin API. The HTTP methods PUT and GET are provided to interact with the Bit Preservation part of the storage.

The configuration of Bit Preservation mechanisms offers three possibilities to modify the parameters of the DBP. A user has the opportunity to determine the Bit Preservation level of the data stored, which translates to the number of replicas stored, to different checksum algorithms and to the interval for integrity checks. These levels are defined by the hosting institution of the DBP implementation. The user can additionally mark her data as archivable and thereby accept longer access times as the data can then be moved to near-line or off-line storage. An administrator of the DBP implementation is able to trigger additional data integrity checks as needed.

For these features an HTTP PUT request is used. All values are set to a default value while ingesting the data with the Storage API. The PUT request must contain the service addressed, the *objectID* of the file, a configuration file for which an XML schema can be found in the specification, and the content-type for the configuration file sent.

Different information about the data and the Bit Preservation mechanisms can be accessed by a user or administrator of the DBP implementation: the Bit Preservation level assigned to the file, the number and location of the replicas, the archivability of the file, the checksum of the file and the algorithm used, the interval of data integrity checks, and the date of the last one.

For this information an HTTP GET request is used. The request must contain the service addressed and the *objectID* of the file for which the information should be retrieved. Optional is the statement which content-type of the information requested will be accepted. The request http(s)://<storage-service>/admin/<objectID> returns all information stored about the file. Requests for individual elements are realized by using extensions of the request URL. A complete list of the extensions and an XML schema for the information delivered can be found in the specification.

B. Authentication and Authorization Infrastructure

The DBP is integrated with the DARIAH Authentication and Authorization Infrastructure (AAI), which is based on the SAML standard [13] that allows federated identity management. Thus not only users registered in the DARIAH user management can access the DBP, but any user who can authenticate within a higher education federation via her campus account and who is authorized to use the service. Here the general issue had to be overcome, that within the campus management no information is stored that can be used for authorization purposes, for example service-specific roles. Therefore the concept of a Virtual Organization [14] that origins from a PKI-based security infrastructure of Grid computing has been adopted to the SAML-based infrastructure by using SAML 2.0 attribute queries and the attribute aggregation feature of the Shibboleth Service Provider (SP).

The DARIAH SPs first collect the authentication information and a persistent ID from the campus Identity Provider (IdP). It then queries additional authorization attributes from the DARIAH IdP. At the IdP side privileges are being managed via group memberships in the LDAP [15] based backend and released by the IdP as entitlement attributes. Through the SP the DARIAH application - in this case the DBP implementation - gets all information as an aggregation of attributes, upon which it can decide whether the user is authorized to use the service or not.

The current group management allows for read and/or write privileges for the whole service. If the single data objects are to be protected by the AAI, an external Policy Decision Point (PDP) can be deployed. The DBP as the Policy Enforcement Point (PEP) can ask the PDP whether a particular user is allowed to perform a particular API method on a particular object.

In addition to the browser based SAML Web SSO profile the SAML Enhanced Client Proxy Profile (ECP) [16] has been implemented for cases when the user does not directly access the DBP via a browser based GUI, but via a web service based user application. Currently other methods than ECP for integrating web services into a SAML based infrastructure are being evaluated.

V. Implementation

For implementing the Storage and the Admin API different components were chosen to fulfill the requirements of the elements of the DBP architecture. The implementation is realized in Java to be independent from the operating system and therefore one criterion for the components chosen, besides being wide-spread and open-source, was to be addressable via a Java API.

A. Components

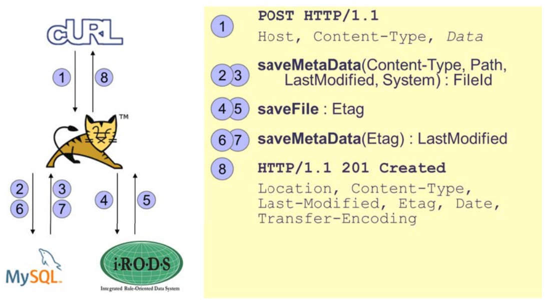
iRODS, the integrated Rule-Oriented Data System [8] is an open-source data management system. It offers a logical namespace independent from the underlying storage and is therefore used as Storage Virtualization Service. An iRODS instance can be accessed via Jargon [17], a Java client API which offers methods to handle files according to the needs of the DBP. *MySQL* [18] is an open-source data base and used as Meta Data Service to store all administrative meta data for the files. The connection is established with JDBC [19] to abstract from the specific data base. *EclipseLink* or the Eclipse Persistence Services Project [20] is the open-source reference implementation of the Java Persistence API (JPA) 2.0 specified in JSR 317 [21]. JPA offers a standard interface for building and managing persistent Java objects and is in this case used to read or write the administrative meta data to the data base. *Jersey* [22] is the open-source reference implementation of the JAX-RS (JRS 311) [23] standard. Among other things it offers special annotations and a testing environment for a simple way of creating RESTful web services. *Apache Tomcat* [24] is an open-source implementation of the Java Servlet [25] and JavaServer Pages [26] specifications and includes a full HTTP web server. The opportunity to execute Java code on this server is used to provide the implementation to the clients.

The DBP architecture and the implementation itself are designed in a way that all components described are principally substitutable. Therefore the implementation is not restricted to the components mentioned above but the developer can use her software preferred. Every web server which is able to execute Java code can be used instead of the Apache Tomcat without any modifications in the implementation. iRODS can for example be substituted by dCache [9], or even a file system or a tape system. Instead of a MySQL instance another data base like PostgreSQL [27], Oracle [28], or again a file system can be used instead. These modifications would only result in slight changes in the implementation as the code is modular and includes abstraction layers.

B. Implementation

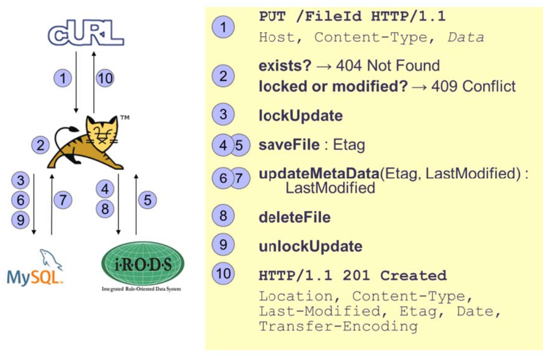
According to the six HTTP methods described in section III the implementation provides six methods which are tagged by the specific annotation @POST, @PUT, @GET, @HEAD, @DELETE or @OPTIONS to offer the HTTP functionalities. The annotation @PATH specifies the relative path for each method and therefore determines where the method is provided.

The POST method (figure 2a) creates a new file on the server. At first administrative meta data is saved in the data base, namely the content-type of the file provided by the request, the storage virtualization system used, the logical file path depending on the storage virtualization system where the file will be stored, a suffix for the updating operation, the last modified time set as the current system time, the persistent



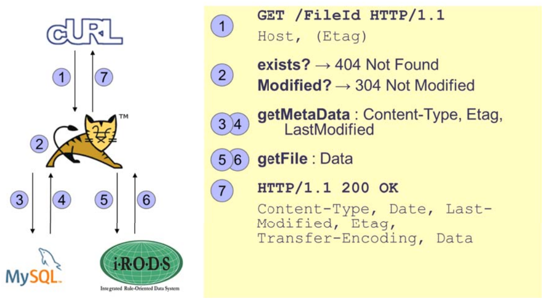
```
① POST HTTP/1.1
  Host, Content-Type, Data
②③ saveMetaData(Content-Type, Path,
     LastModified, System) : FileId
④⑤ saveFile : Etag
⑥⑦ saveMetaData(Etag) : LastModified
⑧ HTTP/1.1 201 Created
  Location, Content-Type,
  Last-Modified, Etag, Date,
  Transfer-Encoding
```

(a) Implementation of POST



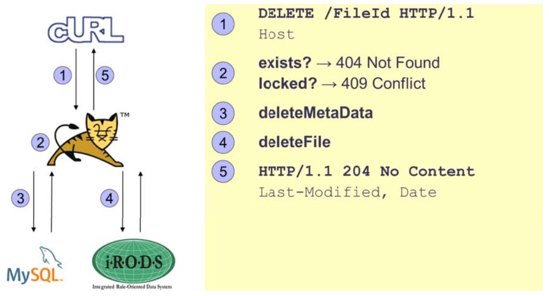
```
① PUT /FileId HTTP/1.1
  Host, Content-Type, Data
② exists? → 404 Not Found
  locked or modified? → 409 Conflict
③ lockUpdate
④⑤ saveFile : Etag
⑥⑦ updateMetaData(Etag, LastModified) :
     LastModified
⑧ deleteFile
⑨ unlockUpdate
⑩ HTTP/1.1 201 Created
  Location, Content-Type,
  Last-Modified, Etag, Date,
  Transfer-Encoding
```

(b) Implementation of PUT



```
① GET /FileId HTTP/1.1
  Host, (Etag)
② exists? → 404 Not Found
  Modified? → 304 Not Modified
③④ getMetaData : Content-Type, Etag,
     LastModified
⑤⑥ getFile : Data
⑦ HTTP/1.1 200 OK
  Content-Type, Date, Last-
  Modified, Etag,
  Transfer-Encoding, Data
```

(c) Implementation of GET



```
① DELETE /FileId HTTP/1.1
  Host
② exists? → 404 Not Found
  locked? → 409 Conflict
③ deleteMetaData
④ deleteFile
⑤ HTTP/1.1 204 No Content
  Last-Modified, Date
```

(d) Implementation of DELETE

Fig. 2: Implementation of four of the HTTP methods described

identifier (PID) if provided by the request, and a flag to lock the file which is set to "false" by default. The data base is in charge of creating and returning a unique id, which is used as a filename for the file uploaded. The file is then stored in the storage virtualization system which returns the checksum which is currently computed with the MD5 algorithm. This checksum is additionally used as an ETag to determine if the file was changed. Finally 201 - Created is returned to the user including the header fields specified.

The PUT method (figure 2b) updates a file on the server. At first it is checked whether the file exists on the server, if not 404 - Not Found is returned. If the file is already locked or has been modified, 409 - Conflict is sent to the user. The file is then locked via the flag in the data base that no concurrent update or delete can take place. The file is stored in the storage virtualization system with a different suffix than the file to be updated. If the transfer completes without failure, administrative meta data, namely the suffix, the last modified date, the ETag, and if provided, the PID are updated. A GET request will now return the file updated. The old version is deleted from the storage and the file is unlocked. Finally 201 - Created is returned to the user including the necessary header fields. If an error occurs during the operation, the file is unlocked automatically and the user has the chance to update the file again.

The GET method (figure 2c) downloads a file from the server. At first it is checked whether the file exists on the server, if not 404 - Not Found is returned. Another check determines whether the file has been updated since the last GET by comparing the ETag or last modified date if provided by the request. If no update has taken place, 304 - Not Modified is sent and the file is not transferred. Otherwise the file and the meta data are fetched from the storage virtualization system and the data base to be provided to the user with 200 - OK.

The HEAD method downloads the meta data of a file from the server. It is implemented in the exact same way as the GET method with the difference that getting the file from the storage virtualization system and delivering it to the user is skipped.

The DELETE method (figure 2d) deletes a file on the server. At first it is checked whether the file exists on the server, if not 404 - Not Found is returned. If the file is currently updated, the status code 409 - Conflict is sent. Otherwise the administrative meta data and the file in the storage virtualization system are then deleted and a 204 - No Content is returned with the necessary header fields to the user.

The OPTIONS method returns the implemented methods for the Bit Preservation implementation, which is included by default in Jersey and needs no further implementation.

VI. Performance

The DBP has been deployed on a dual quad core server with 96 GB RAM and 8 TB of storage to test the implementation without AAI. The clients used run on a standard workstation at the same location connected by a 1 gigabit ethernet network to the server.

Table I concentrates on the four most important methods for handling files. Several 1 MB files were uploaded and further handled, at least two dozen times each, using the client cURL for the Storage API and the iRODS client iCommands. The comparison to the iRODS system was chosen because the current implementation of the Storage API uses such a system as storage virtualization. In this scenario the respective method is examined without concurrent operations as this is the main use case for the arts and humanities. The time between sending the request and receiving the response was measured and averaged. Using a standalone iRODS system is in general faster than using the Storage API as the Storage API includes additional meta data handling and Bit Preservation mechanisms such as computation of the checksum for the upload and update operation. For the download operation the time only differs slightly as just minimal meta data operations are executed.

TABLE I: Average times for uploading, updating, downloading and deleting a 1 MB file via the Storage API and the iRODS API

Method	Storage API	iRODS
upload	113 ms	67 ms
update	141 ms	66 ms
download	138 ms	135 ms
delete	71 ms	54 ms

To analyze the administrative overhead caused by checks, meta data handling, Bit Preservation mechanisms, and building the proper responses as described in V-B several files with different file sizes were uploaded via cURL and the Storage API. The time needed for administrative operations is measured via time stamps within the code and is related to the time needed for storing the file in the iRODS system which is measured in the same way.

TABLE II: Average times for uploading files with various sizes via the Storage API split to administrative overhead and storage

File Size	Administrative	Storage	Overhead [rel.]
10 KB	11 ms	65 ms	17%
100 KB	12 ms	68 ms	18%
1 MB	14 ms	101 ms	14%
10 MB	26 ms	274 ms	9%
100 MB	37 ms	2075 ms	2%
1 GB	27 ms	22129 ms	0,1%

Regarding various file sizes the relative administrative overhead decreases with increasing file sizes as shown in table II because the time needed for storing the file in the iRODS system increases rapidly and gets the main part of the overall time. The additional time for meta data handling measured shows a large variance for large files. A few are comparable to the times needed for small files but the average time needed increases unexpectedly. For small files (10KB and 100KB) the overall time needed only differs slightly. Small outlier due to the network or system processes have a huge impact here and probably cause the differences. The time for storing the file does not increase with the same factor as the file sizes. This is due to the fact, that iRODS itself produces an overhead by creating logical file names and the mappings to the physical file names.

VII. Conclusions

This paper presents the DARIAH Storage API, a novel interface for storing research data in the Bit Preservation service of the DARIAH project. This API offers a standardized and easy-to-use way to access data even if backend technology changes over time. A first version including all six HTTP methods and a basic version of the Admin API have been implemented and secured by the DARIAH AAI. Tests with about 50 thousand files of various sizes were successfully conducted.

The DBP implementation offers a long-term storage with mechanisms like replication and checksums to ensure data integrity. Due to the usage of BLOBs heterogeneous data with different sizes, formats, and contents can be handled. Only administrative meta data for data management purposes is stored in the internal data base, all scientific meta data for the scholar's research is treated as a file. By offering the HTTP methods CRUD operations (create, retrieve, update, delete) can be executed easily and the system can be accessed through the web. All complexity and technology changes inside the system are hidden to achieve an interface as easy-to-use and persistent as possible. Additionally the exchangeability of the hidden components improves the sustainability of the system, which is further supported by the complete integration into the DARIAH infrastructure.

The design of the DBP results in a distributed, generic system which will be reused by other research communities, for example in the Large Scale Data Facility (LSDF) [29] and Large Scale Data And Analysis (LSDMA) projects, which aim to support data-intensive research projects of various communities.

Even though the DBP implementation includes an overhead due to administrative operations according the performance evaluation and the HTTP protocol limits the transfer rates, the time needed for file operations in the DBP are acceptable. As a new functionality to deal with larger resources even more efficiently, partial GET can be supported in the future. A client will then be able to request only a subset of bytes for a given resource. Additionally since the process of ingesting data can take more time, the API can support asynchronous uploads. In such cases the server will response to a POST request with 202 - Accepted and the URL for the resource which will be created later.

The DBP implementation which offers the methods specified by the API is an important part of the humanities research process and therefore for all humanities research projects, but is not yet sufficient. Additional features such as usage of persistent identifiers, format identification, technical meta data extraction, versioning, listing of files or searching need to be provided by Higher Level Services which have to be implemented as a next step. The DBP as a hidden basic service will be integrated seamlessly in such DARIAH High Level Services to provide a comprehensive storage service to the humanities scholars.

Acknowledgements

The authors wish to thank all people and institutions involved in defining and setting up this project. Especially this work has been supported by DARIAH-DE which is partially funded by the German Federal Ministry of Education and Research (BMBF) under the D-Grid initiative by agreement 01UG1110A-M and by DARIAH-EU funded by the European Union under the Seventh Framework Programme, contract number RI211583. Additionally this work is supported by the Helmholtz Portfolio Extension ”Large Scale Data Management and Analysis” with contributions of the Data Life Cycle Lab ”Key Technologies” and the ”Data Services Integration Team”.

References

[1] (2012, August) DARIAH-EU. [Online]. Available: {http://www.dariah.eu}
[2] (2012, August) ESFRI. [Online]. Available: {http://ec.europa.eu/research/infrastructures/}
[3] (2012, August) Amazon S3. [Online]. Available: {http://awsdocs.s3.amazonaws.com/S3/latest/s3-api.pdf}
[4] (2012, August) CDMI. [Online]. Available: {http://snia.org/sites/default/files/CDMI\%20v1.0.2.pdf}
[5] (2012, August) Merritt Storage Service. [Online]. Available: {https://confluence.ucop.edu/download/attachments/16744547/Merritt-storage-service-latest.pdf?version=7&modificationDate=1320361737000}
[6] (2012, August) Data One. [Online]. Available: {http://mule1.dataone.org/ArchitectureDocs-current/apis/REST_overview.html}
[7] H. Neuroth, A. Oßwald, R. Scheffel, S. Strathmann, and M. Jehn, *nestor-Handbuch: Eine kleine Enzyklopaedie der digitalen Langzeitarchivierung*, 2009, ch. 9.4: Persistent Idetifier (PI) - ein Überblick, Available: http://nestor.sub.uni-goettingen.de/handbuch/.
[8] R. Moore and A. Rajasekar, “iRODS: Integrated Rule-Oriented Data System,” September 2008, (White Paper).
[9] M. Ernst, P. Fuhrmann, M. Gasthuber, T. Mkrtchyan, and C. Waldman, “dCache, a distributed data storage caching system,” in *Computing in High Energy and Nuclear Physics (CHEP 2001)*, Beijing,China, September 2001.
[10] Available: http://hdl.handle.net/11858/00-1734-0000-0009-FEA1-D.
[11] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” dissertation, University of California, Irvine, 2000.
[12] R. Fielding, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and B. T. Lee, “RFC 2616, Hypertext Transfer Protocol – HTTP/1.1,” June 1999. [Online]. Available: {http://www.ietf.org/rfc/rfc2616.txt}
[13] S. Cantor, J. Kemp, R. Philpott, and E. Maler, “Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0,” Tech. Rep., Mar. 2005. [Online]. Available: {http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf}
[14] I. Foster, C. Kesselman, and S. Tuecke, “The Anatomy of the Grid: Enabling Scalable Virtual Organizations,” *International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, Aug. 2001. [Online]. Available: {http://www.globus.org/alliance/publications/papers/anatomy.pdf}
[15] K. Zeilenga, “Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map,” June 2006, IETF RFC 4510. [Online]. Available: {http://www.ietf.org/rfc/rfc4510.txt}
[16] S. Cantor et al., “SAML ECP profile schema,” OASIS SSTC, March 2005, Document ID saml-schema-ecp-2.0. [Online]. Available: {http://www.oasis-open.org/committees/security/}
[17] (2012, August) Jargon. [Online]. Available: {https://www.irods.org/index.php/Jargon}
[18] (2012, August) MySQL. [Online]. Available: {http://www.mysql.com/}
[19] (2012, August) JDBC. [Online]. Available: {http://www.oracle.com/technetwork/java/overview-141217.html}
[20] (2012, August) EclipseLink. [Online]. Available: {http://www.eclipse.org/eclipselink/}
[21] L. Demichiel, “JSR 317: Java Persistence API, Version 2.0,” Tech. Rep., 2008. [Online]. Available: {http://jcp.org/aboutJava/communityprocess/final/jsr317/}
[22] (2012, August) Jersey. [Online]. Available: {http://jersey.java.net/}
[23] “JAX-RS: Java API for RESTful Web Services,” September 2009. [Online]. Available: {http://jsr311.java.net/}
[24] (2012, August) Apache Tomcat. [Online]. Available: {http://tomcat.apache.org/}
[25] “JSR 315: Java Servlet 3.0 Specification,” December 2009. [Online]. Available: {http://www.oracle.com/technetwork/java/index-jsp-135475.html}
[26] (2012, August) JavaServer Pages. [Online]. Available: {http://www.oracle.com/technetwork/java/javaee/jsp/index.html}
[27] (2012, August) PostgreSQL. [Online]. Available: {http://www.postgresql.org/}
[28] (2012, August) Oracle. [Online]. Available: {http://www.oracle.com/us/products/database/overview/index.html}
[29] R. Stotzka, V. Hartmann, T. Jejkal, M. Sutter, J. van Wezel, M. Hardt, A. Garcia, R. Kupsch, and S. Bourov, “Perspective of the Large Scale Data Facility (LSDF) Supporting Nuclear Fusion Applications,” in *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, Februar 2011, pp. 373 –379.