Prototype tape library for advanced tape subsystems
Milestone MS8

This project has received funding from the European Union's
Horizon 2020 Research and Innovation Programme
under Grant Agreement No 675191

Lead Beneficiary:
Science and Technology Facilities Council (STFC), Neil Massey, Matthew Jones

Other contributing authors:
Deutsches Klimarechenzentrum GmbH (DKRZ), *Jakob Lüttgau, Julian Kunkel*
The University of Reading (UREAD), *Julian Kunkel, Bryan Lawrence*

Type: *Production of a prototype*

Dissemination level: *PU*

Delivery date Annex 1 (DoA): *project month 36*

Means of verifications: *Source code for the prototype has been produced, a test deployment has been made and preliminary functionality tests have been carried out*

The source code is available, with an open source license, from a repository on GitHub:
https://github.com/cedadev/django-jdma_control/releases/tag/0.1.6

The client application is also available to download from a repository on GitHub:
https://github.com/cedadev/jdma_client

Achieved: Yes

If not achieved, indicate forecast achievement date: *NA*

Comments: *This document includes details of the implementation and deployment of a multi-tiered storage library.*

Summary

The Exploitability work package within ESiWACE contains the objective of:

(3) *Developing* ***New tape access strategies and software customised for Earth system data*** *by first modelling and simulating possible strategies, and then developing a new software library which provides both higher bandwidth to tape storage and increased storage redundancy.*

We implemented and released a simulator for hierarchical storage as part of deliverable D4.1 "Business Model with Alternative Scenarios" (Chapter 6). The developed simulator was used to model a tape library with a cache system together with a simple network topology. The system allows to run traces recorded on existing tape archive and has been used to predict the DKRZ tape archive using real traces. Additionally, the deliverable provided a high-level discussion of the models.

We have since developed new software, consisting of a multi-tiered storage library which provides a single API to users to move data to a number of different storage systems, query the data that they have stored on those systems, and retrieve the data. These interactions are carried out using a common user interface, which is a command line tool to be used interactively, and a HTTP API to be used programmatically. The command line tool essentially provides a wrapper for calls to the HTTP API.

The source code for the multi-tiered storage library can be downloaded from:
https://github.com/cedadev/django-jdma_control/releases/tag/0.1.6

This prototype meets the requirements for milestone 8: Prototype Tape Library

Prototype library for advanced tape systems

Introduction

The Exploitability work package within ESiWACE contains the objective of:

(3) *Developing* ***New tape access strategies and software customised for Earth system data*** *by first modelling and simulating possible strategies, and then developing a new software library which provides both higher bandwidth to tape storage and increased storage redundancy.*

The simulation part of this objective was met within deliverable D4.1 "Business Model with Alternative Scenarios" (Chapter 6). In this milestone report, we address the development of a prototype storage library with support for tape systems. We have done this by developing a multi-tiered storage library which provides a single API to users to move data to a number of different storage systems, query the data that they have stored on those systems, and retrieve the data. These interactions are carried out using a common user interface, which is a command line tool to be used interactively, and a HTTP API to be used programmatically. The command line tool essentially provides a wrapper for calls to the HTTP API.

The goal of providing a multi-tiered storage library is motivated by the fragmenting of storage systems that are now in use at data-centres and, more specifically, the wide range in latency when recovering data from these devices. Users wishing to store or retrieve data on these systems are confronted by a heterogeneous array of tools with different workflows. For example, one system may provide asynchronous transfer, while another may be synchronous and expect the user to run a tool until the transfer has completed. If the system needs a long time to perform operations (high latency) this may lead to errors in the data transmission, in terms of system uptime (the calling system having to maintain a connection to the storage system) or human error (the user closing the lid of their laptop and dropping the connection).

Figure 1 shows the ascending latency of the storage systems in use by JASMIN, the super-data cluster analysis platform funded by NERC and hosted by STFC *[5]*. Users can exploit this hierarchy of storage systems in their workflow by storing and retrieving data to and from the storage systems based on the timeliness of their need for the data. For example, during a simulation run, the user would write the data to the POSIX file system. They will have a small quota on this expensive, low latency, storage system and so will need to migrate the data to the other, higher latency, storage systems. The simulation produces both daily and monthly data, and the user wishes to check the simulation by analysing the daily data as the simulation progresses, produce monthly means and eventually produce yearly means and decadal means. Therefore, they will want to migrate the daily data to the object store, as it will be quick to retrieve when they do their daily and monthly analysis, and they will want to migrate the monthly data to the tape system, so that it can be analysed when more data is available to do the yearly and decadal analysis.

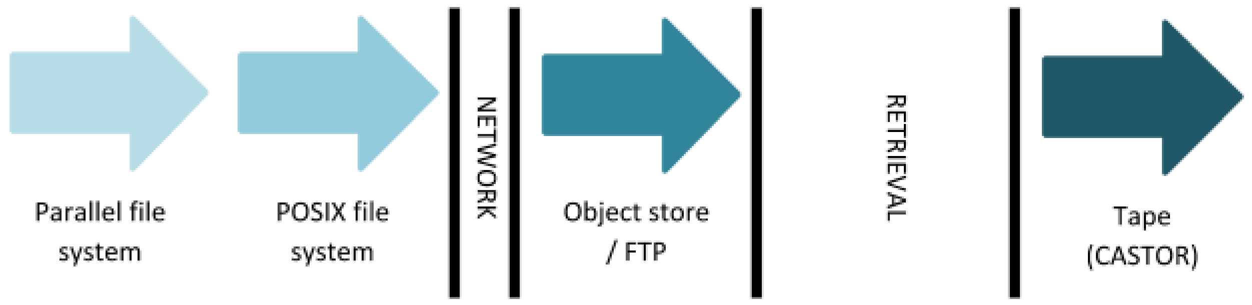

Figure 1: increasing latency in file systems

We have designed and implemented a multi-tier storage library that improves and unifies the user experience across these different storage systems, with the ability to migrate data to object stores (or any service with a S3 interface), FTP (and, therefore tape systems which have FTP as a frontend) and Elastic Tape – a proprietary tape system in use at STFC which is based upon CASTOR *[6]*.

This library has been designed for deployment at any site, but in order to provide focus, it is being piloted at JASMIN where it will enter production in the near future. For this reason, it is known as the **JASMIN Data Migration Application** (henceforth **JDMA**). While the initial purpose was to move data from POSIX disk to tape, many features have been added.

Design Criteria

JDMA was designed with the following criteria in mind:

- The user experience for moving data, regardless of the underlying storage systems, should be identical.
- The user should not be responsible for maintaining the connection to the storage system in the case of asynchronous transfers.
- The user should receive notifications when the transfers are complete.
- Users should be able to transfer data from one storage system to another
- The JDMA system should be distributable, both across computing nodes and across cores on each node.
- Extra storage systems should be able to be added to the JDMA system without extensive re-engineering.

Implementation

JDMA consists of four main components:

1. A webserver application with a HTTP API which accepts requests from users to store or retrieve data and to query the status of the data. This webserver application error checks the requests, writes these requests to a **database** and provides information about the requests, but does not process the requests itself.
2. A command-line client tool which users can interact with. The client tool then forms the HTTP API calls and dispatches them to the webserver.
3. A number of worker programs which monitor and process the requests in the database to move data to and from the storage systems.
4. A number of “storage backend” plug-ins. These are used by the worker programs to handle the data transfers to and from the storage systems. They are written to have a

common API (using object oriented programming) so that adding a new storage system involves only writing a new backend to target that system.

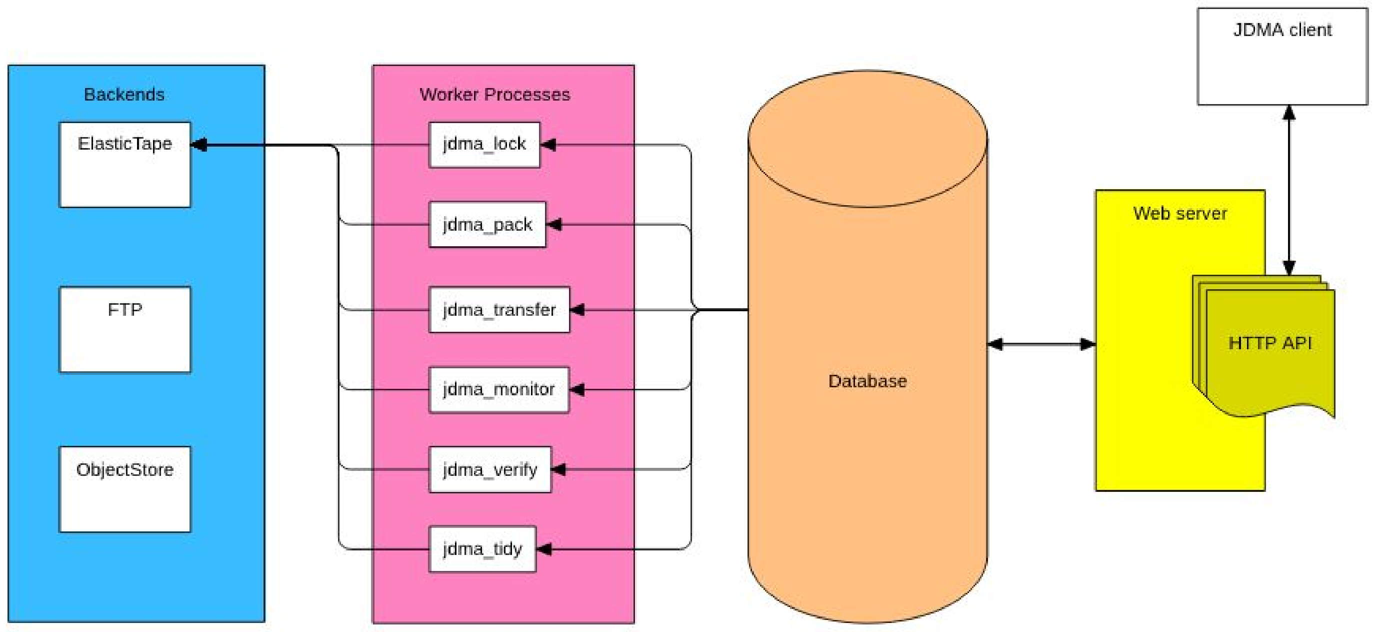

Figure 2: System structure of the JDMA showing the client, HTTP API residing on the webserver, the central database containing information about the migrations and requests, the worker processes which carry out the migrations and the storage backends.

Database schema and data model

JDMA is implemented in Django *[1]*, a Python library which enables the development of database driven web applications. Django requires a data model with a collection of classes which describe the database schema used by the application. The classes, and therefore schema, of **JDMA** consist of:

- **MigrationRequest:** the requests made by the users to store or retrieve data.
- **Migration:** the data (files) stored on the various storage systems. Contains one or more **MigrationArchives**, which in turn contain one or more **MigrationFile**s. See the section **MigrationArchives, piecewise uploading and packing** below
- **User:** The users who have access to **JDMA**.
- **GroupWorkspace:** The unified workspace area for a group of users. See the section **JASMIN, users and group workspaces** below.
- **StorageQuota**: The quota for each group workspace on each storage backend.

Figure 2 shows the data fields of each class and the relationship between these classes. In Django, these relations are expressed via the ForeignKey datatype, for one-to-one relationships, and the ManyToManyField datatype for many-to-one and many-to-many relationships.

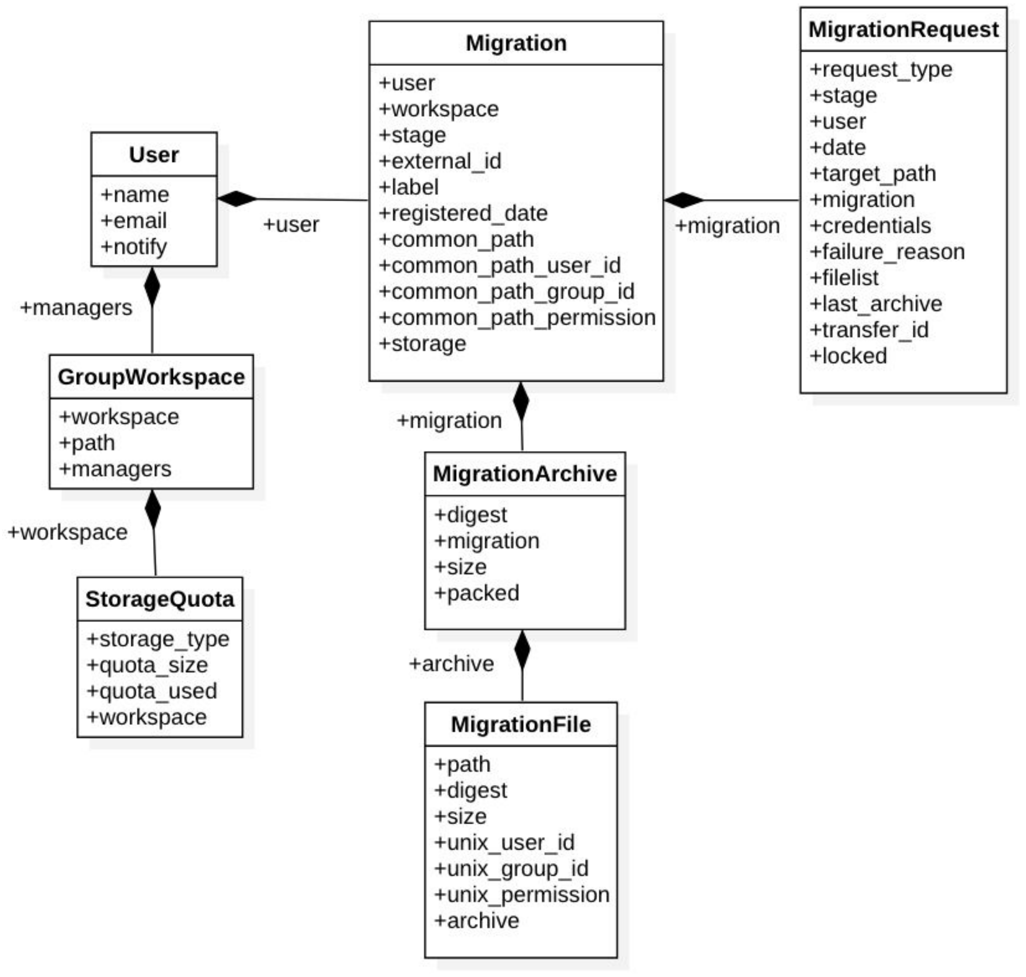

Figure 2 showing the relationships between the 7 classes in the datamodel for JDMA

JASMIN, users and group workspaces

JDMA follows a model of users, workspaces and storage that has been implemented in the user areas and Elastic Tape system on JASMIN. Users are regular UNIX users with a user id and may belong to several groups, each with a group id.

Group workspace are portions of disk allocated for particular projects to manage themselves, enabling collaborating scientists to share network accessible storage on JASMIN. At another location these could be different file systems where users have quotas.

Users can pull data from external sites to a common cache, process and analyse their data, and where allowed, exploit data available from other GWSs and from the CEDA archive. Users, via their groups, can be associated with one or more group workspaces. On JASMIN these group workspaces also have an allocation on Elastic Tape (a proprietary tape library based on CASTOR). In **JDMA** a group workspace can be thought of as a common working area for a group (a unix group with associated gid), which has some allocation of space on a POSIX spinning disk storage system and some corresponding allocation on one or more storage backend.

JDMA replicates this model of users, group workspaces and storage, as can be seen in Figure 2, but adds more storage systems. In **JDMA**, a user belongs to one or more group workspaces and each of these group workspaces has some space allocated on one or more of the storage systems. Whether a user can read or write to a group workspace is not handled in the data model, instead a LDAP server is used for authentication.

HTTP RESTful API and user client

Users make requests to **JDMA** to store (migrate) data, to retrieve data and to query the status of migrations and retrievals. These requests are handled via a HTTP RESTful API, which presents a number of URL endpoints, and each endpoint can have a GET, PUT or POST request made to it. These URLs are closely aligned with the classes in the data model shown in Figure 2. From the base server URL, the URL endpoints are:

URL	**Methods**	**Purpose**
`api/v1/user`	`GET, POST, PUT`	Create a User, update and query User settings.
`api/v1/request`	`GET, POST`	Create a MigrationRequest to migrate or retrieve data and query the status of a MigrationRequest. There is no PUT method as MigrationRequests cannot be modified.
`api/v1/migration/`	`GET, PUT`	Get the status of a Migration and modify a Migration. There is no POST method as Migrations are created when a MigrationRequest is made to migrate data.
`api/v1/archive/`	`GET`	Get the status of a MigrationArchive. There are no PUT or POST methods as the MigrationArchives are created by the jdma_lock worker process, and they cannot be modified.

`api/v1/file/`	GET	Get the status of a MigrationFile. There are no PUT or POST methods as the MigrationFiles are created by the jdma_lock worker process, and they cannot be modified
`api/v1/list_backends/`		Get a list of the backends currently supported by / installed in JDMA

Migrating, retrieving and displaying information about data are carried out by calling the URL endpoints with a GET, PUT or POST method along with a JSON document containing the information required for the operation. The HTTP response is in JSON format, along with a return code indicating whether the call was successful or not.

To allow users to interact with **JDMA**, without having to learn how to make requests to a HTTP API, a command line client program (**jdma_client**) is provided and available in a GitHub *[2]* repository. This client wraps calls to the HTTP API into a command line interface which runs like a regular UNIX / Linux command line program but generates the JSON documents needed to carry out the command and calls the HTTP API using the requests Python library. The JSON returned from the API is then interpreted and displayed to the user. The **jdma_client** also reads in a configuration file (`.jdma.json`) from a user's home directory. This contains information about the user's default backend, as well as any credentials that are required to connect to each backend.

MigrationRequests and quotas

A request to migrate, retrieve or delete data is made via the endpoint `api/v1/request` with a `PUT` request. In the JSON document that accompanies the request is the request type, which is either GET (retrieve data), PUT (migrate data), MIGRATE (migrate data then delete the original copy) or DELETE (remove data). Data is stored in a batch which maps to a list of files. The request to PUT data contains a list of files, and a request id will be returned. A batch id will be created later, when the data is migrated to the external storage. The request to GET data contains a batch id and (optionally) a list of files. If the list of files is supplied then the batch must contain those files and only those files will be retrieved. The request to DELETE data contains just a batch id: only whole batches can be deleted.

Calling `api/v1/request` with a `GET` request requires a request id. This will return information about the request, including which stage it is at.

Each group workspace has a quota on each storage backend, which are independent from the quotas for the other backends. This is realised by the many to one mapping of StorageQuota and GroupWorkspace shown in Figure 2. The storage_id stores the external storage id that the StorageQuota pertains to. When a user requests to migrate some data to a storage backend, the **JDMA** webserver checks if there is any quota remaining. As the webserver does not do any processing, it does not total the file sizes of the requested migration as this would entail too much processing. Therefore, a group workspace could be quite far over their quota if a large migration is made when already close to the quota limit.

MigrationArchives, piecewise uploading and packing

Producing a system that can write to multiple storage backends presents two problems when migrating the data. Firstly, some systems may require all files to be presented as an upload at the same time, such as with tape systems. Other systems may have the ability to upload single files one at a time, such as an object store. In **JDMA,** the latter is termed *piecewise* upload and the backend plugin class must implement a method called `piecewise_upload` which indicate whether the external storage requires all the files at once (returns `False`) or one at a time (returns `True`).

Secondly, some systems may be optimised to upload single files, including very small files whereas others may be optimised to upload large files only. In the latter case it makes sense to form TAR archives of the smaller files and to upload those in place of the individual files. In **JDMA** a backend must implement a method called `pack_data`, which indicates whether data should be packed before being uploaded to the external storage. In conjunction with this, a method called `minimum_object_size` should be implemented if `pack_data() == True.` This indicates the smallest size an archive should be, and files will be added to a TAR archive during a migration to meet this minimum size.

To support backends both with and without packing, a multi-tiered approach to storing file details is used. As can be seen in Figure 2, a Migration contains a number of MigrationArchives and each of these contains a number of MigrationFiles. During the *jdma_lock* worker program, MigrationFiles are added to a MigrationArchive until the total size of the files exceeds the `minimum_object_size,` then a new MigrationArchive is created, added to the Migration and MigrationFiles are added to it. If a Migration is made to a backend which requires the data to be packed, then each MigrationArchive will have its packed field marked as True and a TAR file will be created containing the MigrationFiles in the MigrationArchive. These TAR files are stored in a "staging directory" and removed during the jdma_tidy worker program. Additionally, a checksum digest is calculated for the TAR file that contains the MigrationFiles.

If a Migration is made to a backend which does not require packing then no digest is calculated and the packed field is set to False. The worker programs process MigrationArchives that are packed and unpacked, with the packing and unpacking of tarfiles occurring in a staging directory.

Verification of data

Verifying the integrity of the data on the external storage system is very important for **JDMA** as the original data may be deleted, either by choosing a MIGRATE request or by deleting the data after the transfer has completed. To ensure that the data is not corrupt, two stages are carried out:

1. During the *jdma_lock* worker program, the checksum digest of each file is calculated. This is stored in a MigrationFile object, along with the path, size and unix permissions of the file.
2. After the data has migrated to the external storage, the jdma_transfer worker program retrieves the data to a verification directory. The checksum digests for the files are calculated again and a comparison is made to those stored in the corresponding MigrationFile object in the database. If they differ then the Migration is marked as FAILED.

Worker programs

JDMA follows a design methodology where the user makes requests to a HTTP API, which makes entries into a database but doesn't do any actual processing of the requests beyond some error checking. The actual transfer of data is carried out by a number of worker programs, running in parallel, which query the database to determine which Migrations and MigrationRequests to process, based on their stage in the state machine. These worker programs are detailed below:

Worker program / daemon	**Purpose**
jdma_lock	● Locks the files the user wishes to migrate, or the target directory for a retrieval. ● Creates the Migration in the database (from the MigrationRequest). ● Creates the MigrationArchives and the list of files in each MigrationArchive. ● Calculate checksum digests for each file.

jdma_pack	● If the backend requires data to be packed before transfer, this program creates, in a staging directory, TAR files for each MigrationArchive containing the list of files in each.
jdma_transfer	● Transfer data to or from the storage backends, or delete files from the storage backends. ● Restore owner and group to downloaded files
jdma_monitor	● Monitor which transfers have been completed and transition the state machine for those that have
jdma_verify	● Verify files have downloaded correctly by checking their checksum digest with the one calculated in **jdma_lock**
jdma_tidy	● Clean up after the migration or retrieval by deleting files in the staging directory and verification directory ● Delete the original files if the data has been destructively migrated ● Remove the MigrationRequest as it has now completed

As shown in the table, there are six worker programs running concurrently, all of which act upon a MigrationRequest and could, in fact, all act upon the same MigrationRequest at the same time. This could cause race conditions and errors due to the state of the MigrationRequest being changed out of order. To prevent this, when a worker process acts upon a MigrationRequest it is locked, via an entry in the database, to prevent the other worker processes acting upon it.

Handling connections, multiple requests and authentication credentials

JDMA is effectively acting as a number of users, carrying out data transfers on behalf of those users. This could have a detrimental effect on user's productivity if the transfers are carried out in a serial manner, i.e. one transfer per user, one at a time, with no transfers acting in parallel. JDMA has been designed to allow parallel transfers in a number of ways:

1. The worker processes can be run on different servers for each external storage backend, while connecting to a central database. This allows for a cluster-like architecture, allowing,for example, the transfers of the Elastic Tape to occur on one server, those for Object Store on another server and those for FTP on yet another.
2. Each worker process on each server processes multiple transfers, either from one user or multiple users. **JDMA** launches a separate thread for each transfer, using the Python

multiprocessing library *[7]*, allowing multiple MigrationRequests to occur in parallel. Each backend can set a maximum number of threads that are allowed, and this number can be set to match the server capabilities.

3. Each external storage backend can launch its own threads, drawing from the same maximum number of threads setting as in Point 2. This allows, for example, transfers to and from an Object Store for a single MigrationRequest to occur in parallel.

The scheme for allowing parallel transfers above is limited by the maximum number of available cores on a single server, per external storage backend. For example, if the Elastic Tape backend is running on its own server, then the number of transfers is limited to the number of cores on that server. A further extension to **JDMA** would be to distribute the transfers for each storage backend across a number of servers, via a load balancing mechanism.

A large overhead for transferring data is incurred when connecting to the external storage systems. To mitigate this overhead, a pool of connections is kept within a ConnectionPool list. When a connection to an external storage system is required as part of the migration or retrieval process, a connection object is requested from the ConnectionPool, passing the user's credentials and external storage system required as part of the request. If no useable connection is available, then a new connection is made and a record is made in the ConnectionPool along with the information that it is in use. When a connection is no longer required then the connection object is closed. However, this does not close the connection with the external storage. Instead the connection object is marked as available for use and can be reused when new connections are requested. By reusing connections in this way a minimal number of connections are made when transferring data, even if the transfers occur in parallel.

Each of the backends may require a set of user credentials to authenticate, transfer and retrieve data from it. As **JDMA** is acting as an intermediary for the users, these credentials need to be passed from the user, to **JDMA** and then onto the external storage system. **JDMA** provides a basic authentication system for this. The first component is that each of the backend plugins must implement a method called `required_credentials`, which returns a list of keywords which the user must provide corresponding values for. The keyword / value pairs are supplied in the `.jdma.json` file, which the **jdma_client** reads and then attaches to the JSON document sent to the server when a request is made to one of the HTTP API endpoints. These keyword / value pairs could be, for example: `{"user" : "nrmassey", "password" : "topsecret"}` for external storage that requires a username and password.

At the moment, these credentials are sent "in the clear", i.e., in plain text. This is not ideal, and some private / public key encryption could be used to make this transfer of credentials more secure. When the credentials arrive at the server they are stored in the corresponding

MigrationRequest. However, before storage they are encrypted with a symmetric key encryption (AES encryption) which improves security should the database be compromised. MigrationRequests are ephemeral: they only exist until the data transfer is complete, whereupon they will be deleted as part of the `jdma_tidy` worker process. The symmetric key is stored on the server, and is generated during the installation. This also improves security as, if the database is compromised, the intruder will only be able to recover a subset of user credentials which will have been encrypted. As an added layer of security, all communication with the server has to be carried out via https.

Storage backends

To provide a flexible architecture, and to allow new external storage to be used with a minimum of re-engineering, **JDMA** implements the storage backends as a set of plugins. These plugins all have a common interface and, due to **JDMA** being implemented in Python, the backend plugins inherit from a common Super class: *Backend*. To implement a backend plugin a new class must inherit from the *Backend* class and implement these methods:

Member function	**Purpose**
`available`	Determine whether the storage backend is available, i.e. online and ready to accept storage requests
`monitor`	Returns which **GET**, **PUT** and **DELETE** requests have finished since the last call to the monitor function.
`pack_data`	Return **True** if the backend requires data to be packed into a TAR file before sending to the backend
`piecewise`	Return **True** if the backend allows files to be uploaded one by one, and **False** if it requires files to be uploaded all at once in a batch
`create_connection`	Create a connection to the backend server and return a connection object
`close_connection`	Close the connection
`download_files`	Make a request to the storage backend to download a list of files
`upload_files`	Make a request to upload a list of files
`delete_batch`	Make a request to delete a list of files
`user_has_put_permission`	Return **True** if the user has permission to write to the storage backend.
`user_has_get_permission`	Return **True** if the user has permission to retrieve from the storage backend.
`user_has_delete_permission`	Return **True** if the user has permission to delete from the storage backend.
`user_has_put_quota`	Return **True** if the user has remaining quota on the storage backend.

`required_credentials`	Return a list of keywords which must be present in the user's configuration file, along with the corresponding values.
`minimum_object_size`	The minimum size an archive object should be, when *pack_data()* == **True.**
`get_name`	Get the name of the storage backend for display purposes.
`get_id`	Get the id of the storage backend. This should be the name without any spaces or non-alphanumeric characters.

Migration process

To carry out a transfer of data to a storage backend, termed a migration in **JDMA**, a user submits a PUT or MIGRATE request via either the HTTP API or the command line client (which interfaces to the HTTP API). The difference between a PUT and MIGRATE request is that, with a PUT request the data will be left in its original location, whereas with a MIGRATE request it will be deleted by the *jdma_tidy* worker program after the migration completes successfully. Users can migrate data to a storage system if the backend function `user_has_put_permission()` evaluates to True.

The migration process is as follows:

1. The user makes a request to migrate data to **JDMA**, choosing the backend they wish to move the data to. This can either be a directory or a list of files. This creates a MigrationRequest in the database, as well as an (empty) Migration.
2. *`jdma_lock`* runs on one of the servers in the background and processes the request:
   a. A list of files is built. This is either the list of files passed in or a full depth traversal of the directory.
   b. The MigrationArchives are created in the database, along with the MigrationFiles that are part of the archive, containing the checksum digest, the file size, file owner, group and permissions.
   c. The list of files is locked by changing the ownership to root and the permissions to only be read / write / execute by root.
   d. The files are assigned to MigrationArchives by grouping files together so that the sum of the file sizes is less than `minimum_object_size` for the backend.
   e. A checksum digest is calculated.
3. *`jdma_pack`* is run:
   a. If the archive does not required packing (as the backend the data is to be migrated to does not support packing) then this step is skipped.

   b. Otherwise, for each MigrationArchive a tarfile of the MigrationFiles in that archive is created. These TAR files reside in a "staging directory" which is configurable, on the server, for each backend. A checksum digest is calculated for the tarfile and added to the MigrationArchive information in the database.
4. *`jdma_transfer`* is run:
   a. For backends that do not support piecewise uploading: a thread is created that uploads all of the files at once, using the backend `upload_files` method.
   b. For backends that do support piecewise uploading: the current number of available threads is found, and the MigrationArchives are grouped so that there is an equal number per thread. A list of files is built for each thread by concatenating a list of the MigrationFiles in each group of MigrationArchives. Each thread is then created to upload these concatenated lists of files.
5. *`jdma_monitor`* is run:
   a. Each backend monitors the migrations that it is carrying out and marks those that have completed by transitioning the state machine.

The data has now been migrated to the external storage but, to be confident that the data has transferred correctly, a verification process is now carried out automatically:

6. *`jdma_transfer`* is run:
   a. A list of files is produced by looping over all the MigrationArchives in a Migration. If the MigrationArchive is packed, then the TAR file for the MigrationArchive is added to the list of files. Otherwise all of the MigrationFiles in each MigrationArchive is added to the list of files to download.
   b. If the backend does not support piecewise downloading then a single thread is created to download all the files using the `download_files` backend method.
   c. If the backend does support piecewise downloading then a list of files for each available thread is created in the same manner as in 4a. For each list a thread is created to download that list of files using the `download_files` backend method.
7. *`jdma_monitor`* is run:
   a. Each backend monitors the retrievals that it is carrying out (including the retrievals for verification) and marks those that have completed by transitioning the state machine.
   b. Note that for MigrationArchives that are packed, a TAR file will be downloaded, whereas for MigrationArchives that are not packed, a hierarchy of files will be downloaded.
8. *`jdma_verify`* is run:
   a. For each MigrationArchive in the Migration:

        i. If the MigrationArchive is packed then calculate the checksum digest on the downloaded TAR file.
        ii. If the MigrationArchive is not packed then calculate the checksum digest on every downloaded file in the file hierarchy
    b. Compare the calculated checksum digests with those stored in the MigrationArchive (for packed archives) and those stored in MigrationFile (for non-packed archives)
    c. If any of the digests do not match then mark the migration as failed. The user can then retry
9. *`jdma_tidy`* is run:
    a. Any intermediary files, such as those created by packing the files into TAR files, or downloading the files for verification, are deleted.
    b. If the user request was to MIGRATE the files then the original files are deleted.
    c. If the user request was to PUT the files then the user, group and permissions are restored to the original files.
    d. Delete the MigrationRequest from the database as it has completed.
    e. Subtract the uploaded amount from the group workspace's StorageQuota.
    f. Send a notification email to the user to inform them that their data transfer has completed successfully.

Retrieval process

To carry out a retrieval of data from a storage backend, a user submits a GET request for a Migration, by specifying the batch id, via either the HTTP API or the command line client (which interfaces to the HTTP API). The user retrieval differs from the verify process (Points 6 to 9 above) in that the user can request a subset of files from a single Migration. Users can retrieve data to a storage system if the backend function `user_has_get_permission()` evaluates to True.

The retrieval process is as follows:

1. *`jdma_lock`* is run:
    a. The target directory for the download is created and locked by transferring ownership to root
    b. A staging directory for the download is created for the backend
2. *`jdma_transfer`* is run:
    a. If a subset of files is specified by the user: a list of MigrationArchives to retrieve is created. This consists of the MigrationArchives that contain at least one file in the subset of files

    b. Otherwise, the list of MigrationArchives to download contains all of the MigrationArchives in the Migration.
    c. For each MigrationArchive:
        i. If the MigrationArchive is not packed then download each MigrationFile in the archive directly to the target directory.
        ii. If the MigrationArchive is packed, then the TAR file is downloaded for the archive to a staging directory.
    d. If the backend supports piecewise downloading, then the downloads are distributed across multiple threads, the number of which depends on how many threads are available.
3. *`jdma_monitor`* is run:
    a. Each backend monitors the retrievals that it is carrying out marks those that have completed by transitioning the state machine.
4. *`jdma_pack`* is run:
    a. If the archive does not required unpacking (as the backend the data has been retrieved from does not support packing), then this step is skipped.
    b. Otherwise, for each MigrationArchive the tarfile of MigrationFiles is unpacked from the staging directory to the target directory.
5. *`jdma_transfer`* is run:
    a. The owner, group and permissions are restored using the information stored in MigrationFiles.
6. *`jdma_tidy`* is run:
    a. Any files in the download staging directory are deleted.
    b. The MigrationRequest is deleted.
    c. An email notification is sent to the user to inform them that their retrieval has finished.

Delete process

Users can delete data on a storage system if the backend function `user_has_delete_permission()` evaluates to True. Deletion can be run at any time, even part way through a migration or retrieval. If it is run during a migration then all intermediary files in the staging or verifying directories are also deleted. If it is run during a retrieval then the retrieval is allowed to finish before the deletion occurs. The DELETE MigrationRequest will remain in the request queue until this time.

The delete process is as follows:

1. *`jdma_lock`* is run:

    a. Lock the Migration that is going to be deleted so that no further processing will occur, if it hasn't completed yet.

2. *`jdma_transfer`* is run:
    a. For each MigrationArchive in the Migration:
        i. If the MigrationArchive is packed then delete the tarfile.
        ii. If the MigrationArchive is not packed then delete each file.
    b. Again, if the backend supports piecewise deletion then the deleting of the MigrationArchives is distributed across available threads.
3. *`jdma_monitor`* is run:
    a. Each backend monitors the deletions that it is carrying out and marks those that have completed by transitioning the state machine.
4. *`jdma_tidy`* is run:
    a. The Migration is deleted. This will cascade through the data model and delete the associated MigrationRequest, MigrationArchives and MigrationFiles.
    b. The group workspace's StorageQuota is updated by adding the total size of the deleted files back to the quota.
    c. An email notification is sent to the user to inform them that their deletion has finished.
    d. Any intermediary files that may be present due to a partially completed migration are deleted.
    e. If the worker processes are part way through a migration, and the original files have been locked, then the permissions on original files are restored here.

Transfer state machines

The process of migrating, retrieving and deleting data from the storage backends is complicated and consists of multiple stages and multiple worker programs. This allows the worker programs to be carried out asynchronously, allows the processes to be distributed across cores and nodes, allows progress of the processes to be monitored and allows the continuation of any interrupted processes.

To keep track of which stage each process is at, finite state machines (FSM) are used. The state is stored in the MigrationRequest, for the progress of the processes and in Migration for the overall state of the Migration. These are outlined below and tie closely to the description of the processes in the previous section.

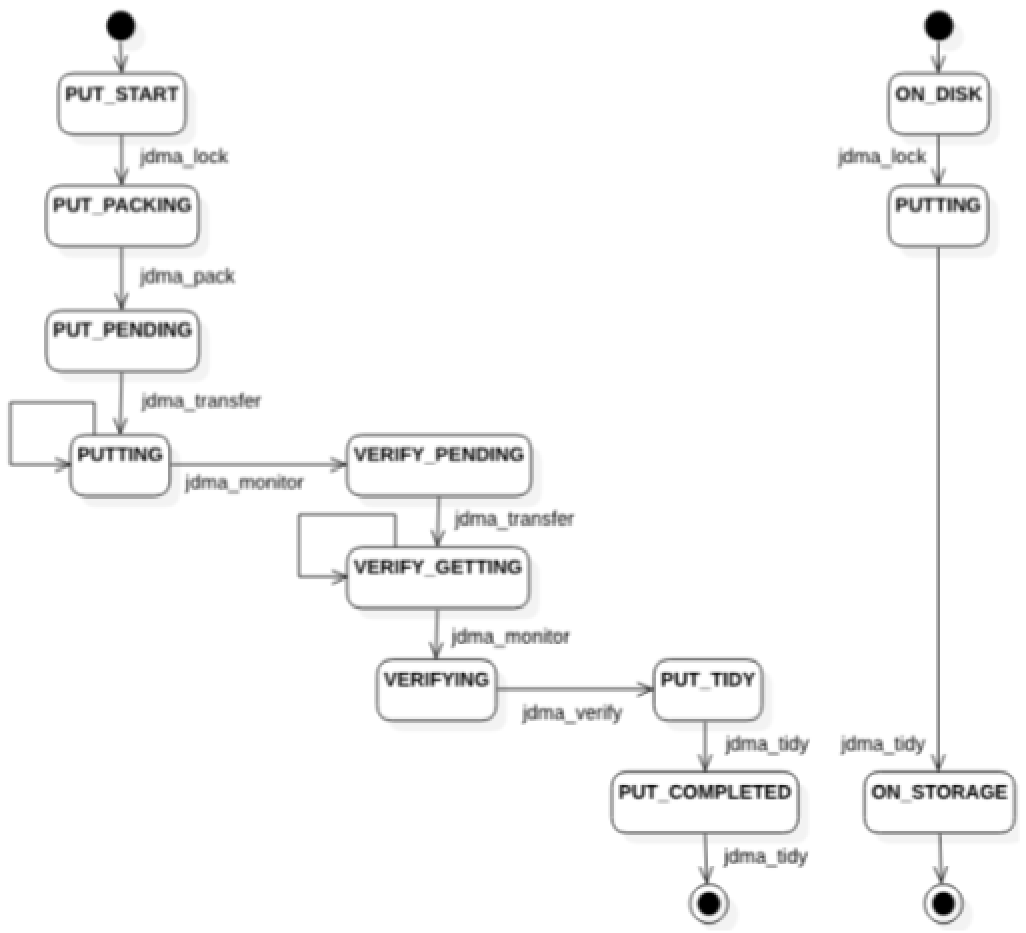

Figure 3: FSM for migrating data. Left: MigrationRequest FSM, Right: Migration FSM

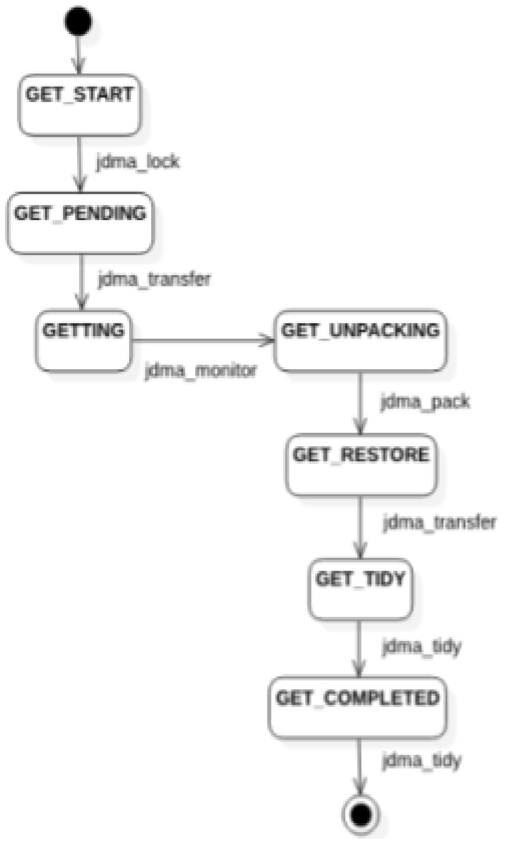

Figure 4: MigrationRequest FSM for retrieving data.

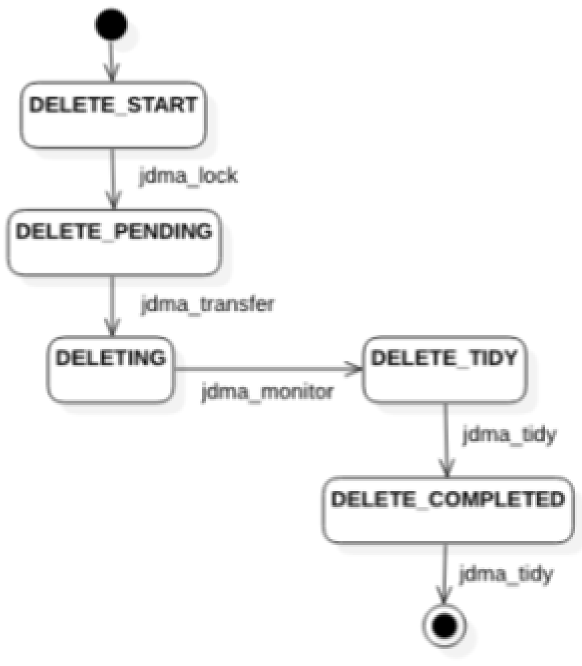

Figure 5: Migration request FSM for deleting data

Current Status

JDMA is currently deployed on JASMIN via a virtual machine in the JASMIN cloud environment. This allows full control over the virtual machine, including root access, making installation straightforward.

- Backends for FTP, Object Store (via Amazon S3) and Elastic Tape (CASTOR based) have been implemented and tested.
- **JDMA** has beta testers from NCAS Computational & Modelling Services.
- Installation is handled via an Ansible script. This makes deployment easy and consistent. Although the deployment is currently on a JASMIN virtual machine, the Ansible script will deploy to any RedHat or CentOS machine, whether that be a physical or virtual machine.

Future work

- Investigating deploying **JDMA** via a container (such as Docker *[3]*) to enable installation on machines where the installer does not have root access. This includes traditional HPC services like ARCHER. A Docker installation would allow a user to install JDMA into a user area without having to go through an installation process and, possibly, without a setup process.

- Installation via Docker would also allow for **JDMA** to be run via a container orchestrator, such as Kubernetes *[4]*. This would allow resources to be allocated to **JDMA** when they are needed and allow **JDMA** to run on a system where the compute may not be persistent - i.e. we cannot guarantee that the system has resources to continuously run the **JDMA** worker processes.
- Writing more backends to target other storage systems.
- Evaluating performance.
- Implementing the ability to move data from one storage system to another (for example from Object Store to tape) by the user issuing a single command and not having to download the files themselves. This would make **JDMA** a truly hierarchical file system and allow the user to choose the storage system based on the latency and their workflow, with a view to moving it to higher or lower latency storage systems at some point in the workflow.
- Investigating the ability to stream the data directly to memory, as a file object, rather than to disk, with the user opening the file object to read it. This would limit the user to using Python, and could be offered as an additional API on top of the HTTP API already implemented.
- Implementing a less naïve staging and caching system. At the moment, if the disk where staging directories are written to is full the migration or retrieval will fail.
- **JDMA** has been designed to be secure, by using https and encrypting authentication credentials in the database, and to mitigate problems caused by users moving or adding to data while a migration is underway, by locking directories. However, a problem remains that, when a directory is locked, the user can move that directory to a different name or location, which will cause errors with the data migration. A possible fix for this would be to move the entire directory to a directory owned by root before the migration.
- Additionally, we will carefully investigate the security of all use cases to harden the system further.
- Monitoring needs to be improved, including providing monitoring of requests to the user (some of this can be achieved currently via the command line tool and RESTful API), and monitoring of the worker programs. The worker programs write logs but more interactive monitoring would be useful.

References

[1]: Django Project, https://www.djangoproject.com/
[2]: GitHub, https://www.github.com/
[3]: Docker, https://www.docker.com/
[4]: Kubernetes, https://kubernetes.io/
[5]: JASMIN, http://jasmin.ac.uk/
[6]: CASTOR, http://castor.web.cern.ch/
[7]: Python multiprocessing, https://docs.python.org/3/library/multiprocessing.html