Open LTFS design

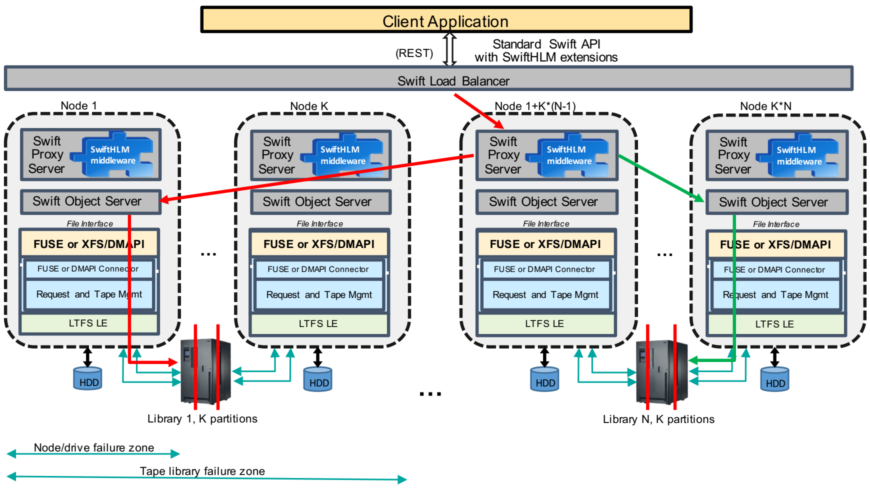
1. Introduction

IBM currently provides two ILM solutions on Linux to migrate data from disk to tape: Spectrum Protect HSM and Spectrum Archive. Both solutions are integrated within the ILM framework that Spectrum Scale provides. Furthermore, Spectrum Protect HSM requires a Spectrum Protect Server and Spectrum Archive is based on Spectrum Protect HSM code (a Spectrum Protect Server is not necessary in that case).
For users who do not want or cannot use Spectrum Scale there is no such solution available. Especially cloud object storage providers who build up their environment on standard hardware and software have little interest to use a highly advanced file system like Spectrum Scale. Clustering functionality that is provided by Spectrum Scale is not necessarily needed by since the clustering functionality already provided with e.g. OpenStack SWIFT.

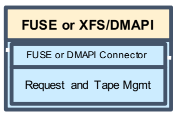
To implement a HSM solution we have identified two technologies to use:

- The file system type XFS is usually used for Object Storage installations. XFS provides an API to perform ILM operations which complies to the DMAPI standard. The DMAPI interface is also used by the two the Spectrum Archive implementations.
- The German company BDT Media Automation GmbH implemented an open source software that is based on a specifically implemented FUSE layer and provides HSM functionality. This software is based on LTFS SE (LTFS Single Drive Edition).

The following figure gives on overview on the design from a high level view with an integration into a SWIFT framework. SWIFT provides the clustering capabilities. A single Open LTFS node is not aware of other Open LTFS nodes beside the fact that the tapes need to be distributed between the different nodes and a single tape only can be used on a single node.



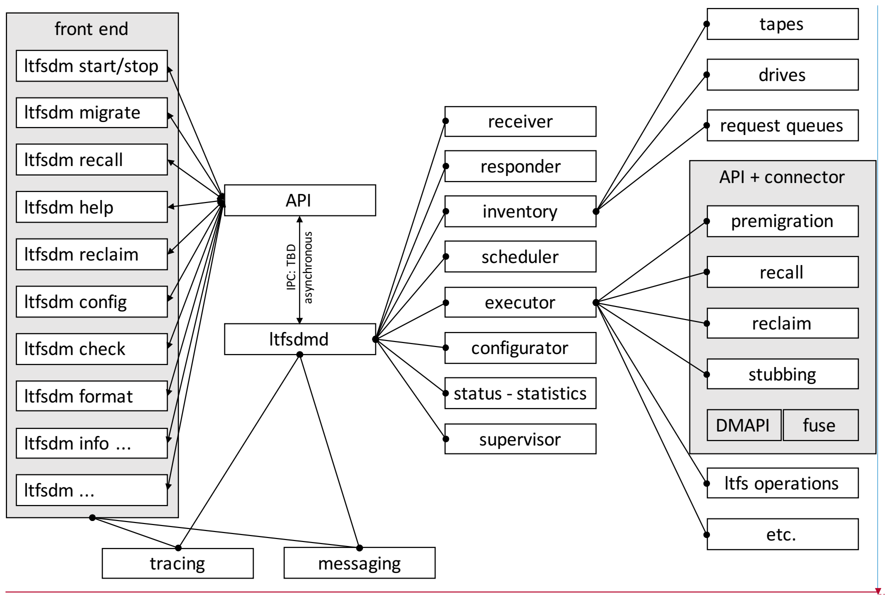
Within this figure the Open LTFS software consists of the following:



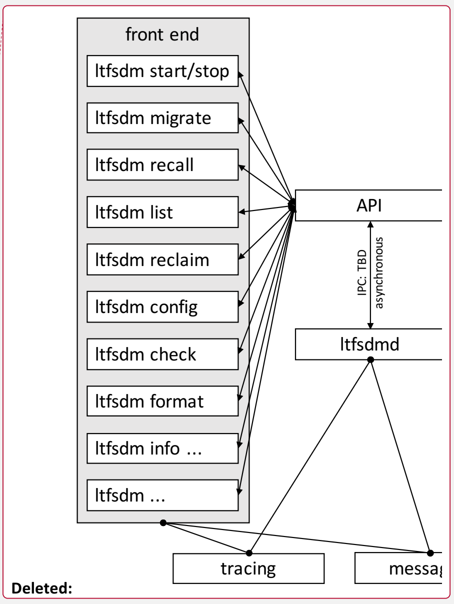
Open LTFS is not necessarily required to run within a SWIFT framework. It can be used in a non-clustering environment or additional software need to clustering capabilities like SWIFT is doing.

2. Overview

Open LTFS consists on a daemon as the Open LTFS service background process (`ltfsdmd`) and a set front end command to manage the software (`ltfsdm`) used with specific sub-commands:



Deleted:



Within the following sub sections all these components are discussed.

2.1. Front end commands

There exists a single primary command `ltfsdm` (ltfs data management) to have a single interface for the user. Specific functions to manage the software are implemented by sub-commands. These sub-commands are the following:

`start/stop`	to start or stop the Open LTFS service in background.
`status`	to see the status of the Open LTFS service.
`migrate`	to migrate file system objects from the local file system to tape.
`recall`	to recall file system objects back from tape to local disk.
`help`	to provide a summary of all available commands.
`reclaim`	to reclaim space on one or a set of tapes.
`config`	to initially set or change the configuration.
`check`	to check the consistency of a tape.
`format`	to format a tape.
`info …`	to show status information for various components (tape, drives, etc.).

An API exists to implement these commands which performs the communication between the front end and the back end service. This API also can be used by other applications to implement further functionality.

2.1.1. ltfsdm start/stop

```
ltfsdm start
ltfsdm stop
```

These commands start and stop the ltfsdm service (ltfsdmd). Only one ltfsdm service can run at a time.

2.1.2. ltfsdm status

2.1.3. ltfsdm migrate

```
ltfsdm migrate -h
ltfsdm migrate [-w] [-p] [-r <request number>] <file name> …
ltfsdm migrate [-w] [-p] [-r <request number>] -f <file list>
ltfsdm migrate [-w] [-p] [-r <request number>] -d <directory name>
```

The migrate command is used to migrate one or more files into premigrated or migrated state. The file names can be provided as parameters of the command or in a file list. It can be chosen if the command returns immediately and provide a request number that can be used with the `ltfsdm info request` command to see if the request is finished or the command will be blocked. A request number can be specified to add additional file names to a migration request previously started. It is also possible to recursively migrate files by specifying a directory.

Deleted: `list`

Deleted: list the migration status of one or a set of file system objects.

Formatted: Font:Andale Mono

Formatted: Heading 1, Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.27 cm + Indent at: 2.16 cm, Don't keep with next

Formatted: Code, Don't keep with next

Formatted: Font:Andale Mono

Formatted: Heading 1, Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.27 cm + Indent at: 2.16 cm, Don't keep with next

Formatted: Font:Andale Mono

Formatted: Heading 1, Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.27 cm + Indent at: 2.16 cm, Don't keep with next

Formatted: Don't keep with next

Formatted: Font:Andale Mono

Options:

-h	information about usage is provided
-p	migrate the all files into premigrated state – otherwise files get into migrated state
-w	the command blocks until the request is fully processed
-r	request number of a previously started migration request
-f	the file list that contains file names of files to be migrated
-d	process recursively all files below the specified directory

2.1.4. ltfsdm recall

```
ltfsdm recall -h
ltfsdm recall [-w] [-p] [-r <request number>] <file name> …
ltfsdm recall [-w] [-p] [-r <request number>] -f <file list>
ltfsdm recall [-w] [-p] [-r <request number>] -d <directory name>
```

The recall command is used to recall one or more files into premigrated or resident state. The file names can be provided as parameters of the command or in a file list. It can be chosen if the command returns immediately and provide a request number that can be used with the `ltfsdm info request` command to see if the request is finished or the command will be blocked. A request number can be specified to add additional file names to a recall request previously started. It is also possible to recursively recall files by specifying a directory.

Options:

-h	information about usage is provided
-p	recall the all files into premigrated state – otherwise files get into resident state
-w	the command blocks until the request is fully processed
-r	request number of a previously started recall request
-f	the file list that contains file names of files to be recall
-d	process recursively all files below the specified directory

2.1.5. ltfsdm help

```
ltfsdm help
```

The help command provides a summary of all available commands. This summary also is shown when not specifying any command.

2.1.6. ltfsdm reclaim

Formatted Table

Formatted: Don't keep with next

Formatted: Font:Andale Mono

Formatted: Heading 1, Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.27 cm + Indent at: 2.16 cm, Don't keep with next

Formatted: Font:Andale Mono

Formatted: Heading 1, Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.27 cm + Indent at: 2.16 cm, Don't keep with next

Formatted: Font:Andale Mono

Formatted: Code, Don't keep with next

Formatted: Font:Andale Mono

Formatted: Heading 1, Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.27 cm + Indent at: 2.16 cm, Don't keep with next

2.1.7. ltfsdm config

2.1.8. ltfsdm check

2.1.9. ltfsdm format

2.1.10. ltfsdm info …

2.1.10.1. ltfsdm info requests

```
ltfsdm info requests [-h] [-w] [-r <request number>]
```

If a migration, recall, or other request has been started in background (not using the "-w" option) the info request command can be used to query its status. If the request number is specified information about that particular request is provided. If no request number is specified all current requests in progress are listed.

Options:

-h	information about usage is provided
-w	the command blocks until the request is fully processed
-r	request number to show information for

2.1.10.2. ltfsdm info files

```
ltfsdm info files -h
ltfsdm info files <file name> …
ltfsdm info files -f <file list>
ltfsdm info files –d <directory name>
```

The list command is used to show the migration state of one or more files. The file names can be provided as parameters of the command or in a file list.

Options:

-h	information about usage is provided
-f	the file list that contains file names of files to be listed
-d	provide recursively information for all files below the specified directory

Formatted: Font:Andale Mono

Formatted: Heading 1, Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.27 cm + Indent at: 2.16 cm, Don't keep with next

Formatted: Font:Andale Mono

Formatted: Heading 1, Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.27 cm + Indent at: 2.16 cm, Don't keep with next

Formatted: Font:Andale Mono

Formatted: Heading 1, Outline numbered + Level: 3 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.27 cm + Indent at: 2.16 cm, Don't keep with next

Formatted: Code: list

Formatted: Heading 1, Outline numbered + Level: 4 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.9 cm + Indent at: 3.05 cm, Don't keep with next

Formatted: Code, Don't keep with next

Deleted:

Formatted: Indent: First line: 0.63 cm, Don't keep with next

Formatted: Don't keep with next

Formatted: Code: list

Formatted: Heading 1, Outline numbered + Level: 4 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 1.9 cm + Indent at: 3.05 cm, Don't keep with next

Formatted: Indent: First line: 0.63 cm, Don't keep with next

Formatted: Don't keep with next

Formatted: Don't keep with next

Formatted: Indent: First line: 0.63 cm, Don't keep with next

2.2. Tracing and messaging

The same tracing and messaging facilities are used for the front end as well as for the back end service. Messages are defined in a single file while trace information is not. Trace statements are defined at its particular position within the code. All messages have a specific identifier that consists of the following parts:

```
    OLTFS[S|C]NNNN[I|W|E]
```

where

- 'S' or 'C' is used based on if the message is written by the OpenLTFS backend process or by the client.
- NNNN is a four digit number.
- 'I', 'W', and 'E' are used depending if the message is informational, a warning, or an error message.

For some of the messages – like usage information – it is not appropriate to print out the identifier. The message identifier is following by a line number enclosed in round brackets.

A message can look like the following example:

```
OLTFSC0005E(50): wrong command 'asd' specified.
```

The C style macros MSG_OUT(<identifier>, arguments) for the messages showing the message identifier and MSG_INFO((<identifier>, arguments) for messages skipping the identifier within the output are used.

For tracing the following macro is used:

```
TRACE(<level>,<variable to inspect>);
```

Trace level are not defined yet.

E.g. an output of the arguments of the main routine

```
TRACE(0, argv[1]);
TRACE(0, argc);
```

can look like the following:

```
Wed Sep 14 15:41:26 2016:       ltfsdm.cc(41): argv[1](sd)
Wed Sep 14 15:41:26 2016:       ltfsdm.cc(42): argc(3)
```

Formatted: Code, Indent: First line: 1.27 cm, Don't keep with next

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

Deleted: .

Formatted: Code, Don't keep with next

Formatted: Code, Don't keep with next

Formatted: Highlight

Formatted: Code, Don't keep with next

Formatted: Code, Don't keep with next containing time stamp, file name, line number in round brackets, variable name, and variable value in round brackets. Only the value of standard data types can be inspected.

To write trace and message output to corresponding log files a facility should be used that is most common on Linux and its distributions. It looks like on RHEL 7 and on SLES 12 the rsyslog service is available for logging purposes.

2.3. Communication between front end and back end service

The communication between the front end command and the back end service is implemented within a library to make it available to other application by an API.

The communication is asynchronous (can be a subject of change).

Deleted: *TBD: communication method*

Within Spectrum Archive SUN RPC is used as the communication method without any access and security characteristics. The communication method used for Open LTFS stratifies that only authorized clients get access to the back end service. The communication is encrypted.

Comment [SS2]: Authentication/Authorization is useful security feature for this use case, but not sure if encryption is useful/needed.

The communication between the front end and the back end only happens within a single system. There is no communication over the network. UNIX domain socket communication is chosen since it is a standard way performing IPC on UNIX like operating systems.

2.4. Data serialization

Formatted: Heading 1, Don't keep with next

UNIX domain socket communication does not specify the way data is serialized for the data transfer. To satisfy that the data sent out by the client is in the same format the server can read it another framework is required. There had been two possibilities: to write a framework or to use some third party software. Open LTFS makes uses Googles Protocol Buffers. Protocol Buffers are publicly available since 2008 and are used for *nearly all inter-machine communication at Google* (see https://en.wikipedia.org/wiki/Protocol_Buffers).

Formatted: Font:Italic

The data serialization is defined by so called protocol files. The content is written in a special interface description language that has similarities to the C language. A compiler can – beside other languages – can generate C++ files out of a protocol file.

A sample definition for the information that is sent from the front end to the back end for migration and which includes

- a key,
- a token,
- the target state (migrated or premigrated),
- and the file names of the target files

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

can look like the following:

Formatted: Indent: First line: 0.63 cm

```
message LTFSDmMigRequest {
    required uint64 key = 1;
    required uint64 token = 2;

    enum State {
        PREMIGRATED = 0;
        MIGRATED = 1;
    }
    required State state = 3;

    message FileName {
        required string filename = 1;
    }

    repeated FileName filenames = 4;
}
```

Formatted: Code: list, Font:(Default) +Theme Body (Calibri), 11 pt, Font color: Auto

2.5. Back end service

The Open LTFS service performes eventually the tasks that are initiated by front end commands. The service is started by issuing the

```
ltfsdm start
```

command and stopped using the

```
ltfsdm stop
```

command. The Open LTFS service consists of the following components

receiver	The receiver listens for messages sent from the front end. It moves further processing to a different thread to be able to receiver further messages.
responder	The responder is responsible to send back an answer on a front end request.
inventory	The inventory keeps information about hardware (tapes and drives) and also the request queues.
scheduler	The scheduler is responsible to assign scheduler requests to resources (tapes and drives). It is event based: e.g. drives becomes free, new item added to the first position of a request queue.
executor	The executor executes a job queue for a specific drive and tape. It is initiated by the scheduler.
configurator	The configurator is responsible for the initial configuration and configurations changes.
status - statistics	The status and statistics part provides information about the configuration, the resources and the progress of requests/jobs.
supervisor	Not sure if this is necessary: if tasks are blocked the supervisor is informing the user that “something” does not proceed.

These components are executed by one or several separate threads each.

2.6. Receiver and Responder

2.7. Inventory

The inventory component keeps information about elements and status of

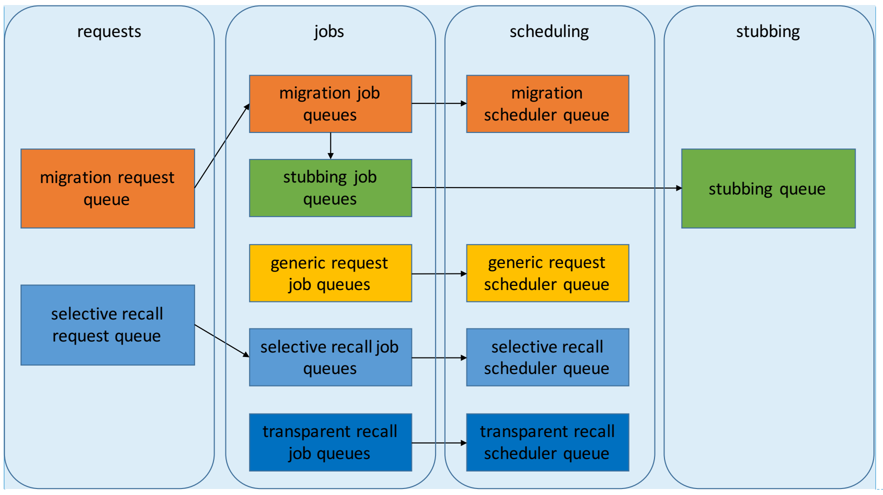
- drives,
- tapes,
- and request, job, and scheduler queues

Deleted: eventually performed

Comment [SS3]: Maybe describe receiver/responder as one component, it will also be one thread. I suppose there will be no separate IPC back-calls from responder to frontend components as that would be complicated.
-> responder is function of ltfsdm that processes query requests from frontend

Maybe add paragraph that explicits what is synchronous (one IPC call) and what is asynchronous (e.g. to migrate files submit request and get req accepted, then later query status).

2.7.1. Request Queues, Job Queues, and Scheduler Queues



requests
jobs
scheduling
stubbing
migration job queues
migration scheduler queue
migration request queue
stubbing job queues
stubbing queue
generic request job queues
generic request scheduler queue
selective recall request queue
selective recall job queues
selective recall scheduler queue
transparent recall job queues
transparent recall scheduler queue

Comment [SS4]: To be in sync with the rest of the text:
migration job queues -> premigration job queues
migration scheduler queue -> premigration scheduler queue

Migration and selective recall operations usually are performed on a predefined set of file system objects. Examples are the migration (or selective recall) of all file within a file system, the migration (or selective recall) of files with a specific pattern of their names, or the migration (or selective recall) of all files within a certain directory. If the number of objects is large the corresponding migration or recall operation can take a lot of time. Users might want to get information about the progress of such an operation. A so called migration or selective recall request refers to all jobs of at least a single migration or selective recall command. A job in this case is related to the migration and recall of a single file. Migration (resp. selective recall) requests are listed in the migration (resp. selective recall) request queue. The jobs of one migration (resp. selective recall) request are listed in one or multiple premigration or stubbing (resp. selective recall) job queues, one per tape involved. Purpose and overview of additional queues related to scheduling and stubbing is covered later in this section. Detailed use of all the queues is covered in later sections. A user can see the progress of one or more

Deleted: contains

migration or selective recall commands by checking the status of the corresponding requests. There are two lists, one for the migration requests:

mig req #	req. name	col	only premig
①	2016-07-25 14:23:10	2	no
②	'exported directories'	1	no

migration request queue

and another for the selective recall requests:

sel rec req #	req. name
①	2016-07-21 13:11:42

selective recall request queue

Migration and selective recall requests are initiated by using the `lfsdm migrate` or `ltfsdm recall` commands. Each of the commands is providing the corresponding consecutive migration or selective recall request number within their outputs. To combine several of these commands within one request it is possible to specify a request number as a commands parameter. If e.g. a migration commands is planned to be issued several times and the same migration request number should be used the first of the series of commands is started without specifying a request number. A request number gets generated in this case. For any subsequent commands the request number that has been shown within the first commands output can be reused and specified as a parameter. Doing so all that series of command belong to the same migration request.

A name can be assigned to a request for better identification. If there is no name specified, the time stamp the request had been issued is shown.

For migration, a collocation number can be used to determine the number of different tapes used for a migration request.

The request queues do not provide any statistical information like the number of files in resident state, number of files in premigrated state, number of files in migrated state, or number of jobs that failed. This information is kept within the corresponding job queues. A query on the request status collects and provides that status of all related job queues.

A job queue is a list of jobs belonging to the same migration or selective recall request or to the same tape for other operations. If e.g. a selective recall command is issued on a set of files for each file a job is created and added to the same job queue. Job queues are defined for different job types:

premigration, stubbing, generic jobs (e.g. format a tape, check a tape), selective and transparent recall.

A job is either associated with a single file in the case of premigration, stubbing, and recall or associated with an operation in the case of a generic job. An operation can be e.g. formatting or checking a tape.

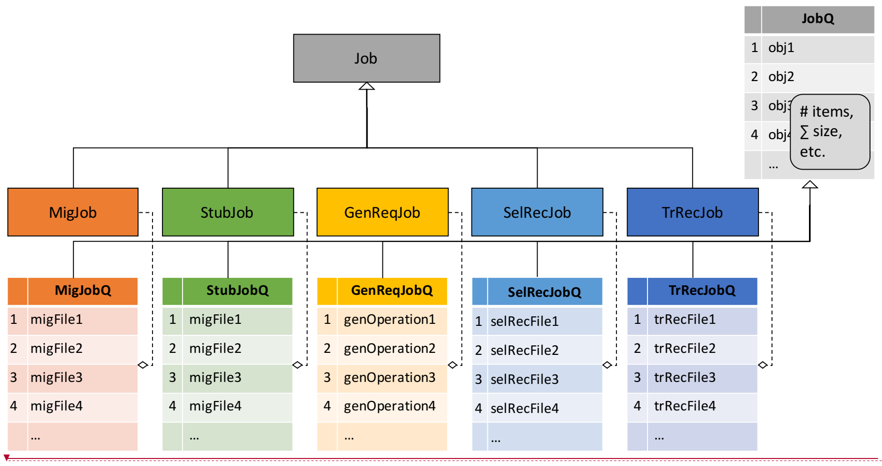
Formatted: Font:Andale Mono

Formatted: Font:Andale Mono

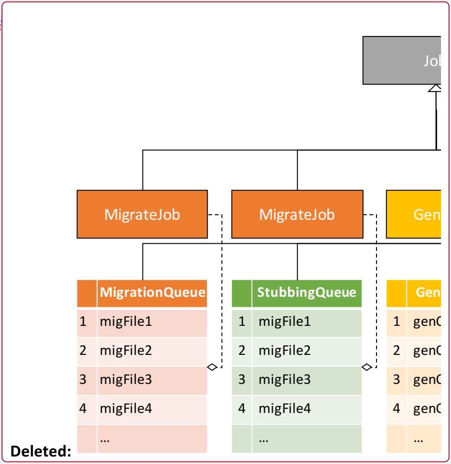
Comment [SS5]: We might want to additionally allow specifying request number as input OR UUID as input, so user or script does not have to care which one is first (requests may also come from a remote application e.g. MCStore through Swift). If this is allowed, the input should be generated as (documented that must be generated as UUID, https://en.wikipedia.org/wiki/Universally_unique_identifier)

Deleted: A consecutive number is assigned to each of the requests.

The relation between jobs and job queues can be seen in the following:



Deleted:



In Spectrum Archive there is a scalability issue because there only exists a single job queue. To find particular jobs to schedule this single job queue needs to be traversed. In the case this single job queue becomes large (e.g. a big number of files are target of migration) the processing of the queue can take a lot of time. Therefore, the idea here is to split one single queue into multiple queues:

- The migration operation is split into two phases. The premigration processing is handled by a premigration job queue. After a file changed to the premigration state the corresponding job is moved to the bottom of a corresponding stubbing job queue to perform the stubbing operation. A single stubbing job queue always is related to a single premigration job queue.
- Premigration (and the corresponding stubbing) job queues are created on request. For migration a colocation option has been introduced to tell how many different tapes should be used for a single migration request. A collocation number of 1 says that only one tape should be used at a time, and only if that tape gets full another tape can be mounted and used. For a collocation number of 2 two different tapes should be target of a single migration request, i.e. two premigration job queues are created so that premigration to both tapes may happen in parallel if there are drives available. For a collocation number of 3 three premigration job queues are created. For further migration requests further migration job queues need to be created.
  Another option for migration tells if an operation should end after the premigration phase or if stubbing should occur. Some users might like to have files not in the migrated but in the premigrated state after a migration operation.
- For each selective recall request and for each tape being involved a separate selective recall job queue is created. If e.g. the files to be recalled for a selective recall request are distributed to two tapes two selective recall job queues are created. The creation of selective recall job queues is independent from the existence of other parallel selective recall requests for same tapes.
- The number of other types of queues (generic job queue and transparent recall job queue) are fixed and correspond to the number of tapes added to the system.

Comment [SS6]: At first it looks a bit strange to "consider/count" also empty job queues. But that might actually be a good thing if considering detection of resubmission of the same tape/type of generic jobs (avoids parsing and cost not much memory since number of tapes is relatively small).
Btw, in which case do/how we prevent submitting two generic jobs to the same tape? E.g. what should be CLI response if resubmitting reclaim request for the same tape?

We could e.g.:
- detect resubmission of same tape/type generic job and fail the resubmission with "already submitted";
- on a case by case rule basis allow or not new generic job for a tape for which other/different generic jobs exist (in some cases it might make more sense to reject)

If a new premigration job queue or selective recall jobs queue has been created or if a job has been added to the generic or transparent recall job queues this job queue gets assigned to one of the four so called scheduler queues (transparent recall scheduler queue, selective recall scheduler queue, generic scheduler queue, and migration scheduler queue). The scheduler is scheduling job queues and **no** single jobs. If a job queue gets scheduled it is processed from the top to the bottom. Similarly, if a new stubbing job queue has been created it is added to the so-called stubbing queue. The stubbing queue lists all the stubbing job queues and is parsed and processed before and after tape index sync by the stubbing thread(s).

Each job queue provides status information:

- the number of jobs within the queue.
- in the case the queue contains file names the total size of all files.
- if a queue is in progress the job that is currently processed. If e.g. within the premigration phase data is being copied from disk to tape the corresponding file name is provided.

The migration and selective recall request queues can identify corresponding migration and recall job queues by the migration and recall request numbers.

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

Formatted: Indent: First line: 0.63 cm

Comment [SS7]: I was thinking we also use reference(s) ((pointer(s)) from a request in a request queue to its job queue(s). Also a scheduler queue entry should refer (point) to a job queue to be able to update its status/statistic by the process processing that job queue. Maybe we want to write that here.

2.7.2. Examples

For each of the queues mentioned before I here give some examples. In all job queues jobs get processed from top to the bottom after a job queue has been scheduled. New jobs always are added to the bottom of a queue.

2.7.2.1. Premigration Job Queues:

mig req #	req. name	col	only premig
①	2016-07-25 14:23:10	2	no
②	’exported directories’	1	no
migration request queue			

	①	PMQ-COLQ1
1	file aa.	
2	file aa.	
3	file aa.	
4	file aa.	
	,,,	

	①	PMQ-COLQ2
1	file ab.	
2	file ab.	
3	file ab.	
4	file ab.	
	…	

	②	PMQ-COLQ1
1	file ca.	
2	file ca.	
3	file ca.	
4	file ca.	
	…	

Migration requests are performed on a predefined set of file system objects. Within this example there are two migration requests (“①2016-07-25 14:23:10” and “② ’exported directories’”). For migration request ① the collocation option is set to 2. Migration jobs in this case are distributed to two premigration job queues to later distribute them to different tapes. For migration request ② only one premigration job queue has been created since the collocation option is set to 1. Each job corresponds to a file to migrate.

Deleted: distributes

2.7.2.2. Stubbing Job queue:

	①	PMQ-COLQ1
1	file aa.	
2	file aa.	
3	file aa.	
4	file aa.	
	,,,	

	①	T1	SQ-COQ1-2
1	file aa.		
2	file aa.		
3	file aa.		
4	file aa.		
	,,,		

	①	PMQ-COLQ2
1	file ab.	
2	file ab.	
3	file ab.	
4	file ab.	
	…	

	①	T2	SQ-COQ2-2
1	file ab.		
2	file ab.		
3	file ab.		
4	file ab.		
	…		

	②	PMQ-COLQ1
1	file ca.	
2	file ca.	
3	file ca.	
4	file ca.	
	…	

	②	T3	SQ-COQ1-2
1	file ca.		
2	file ca.		
3	file ca.		
4	file ca.		
	…		

stubbing queue	tape	fin
①-SQ-COQ1-1	T1	*
①-SQ-COQ2-1	T2	*
②-SQ-COQ1-1	T3	*
①-SQ-COQ1-2	T1	
①-SQ-COQ2-2	T2	
②-SQ-COQ1-2	T3	

	①	T1	SQ-COQ1-1
1	file aaa		
2	file aab		
3	file aac		
4	file aad		
	,,,		

	①	T2	SQ-COQ2-1
1	file aba		
2	file abb		
3	file abc		
4	file abd		
	…		

	②	T3	SQ-COQ1-1
1	file caa		
2	file cab		
3	file cac		
4	file cad		
	…		

The migration process is split into two phases. The first one copies data to tape (premigration). Thereafter the data needs to be removed from local storage (stubbing). The reason why to do that in two steps is that the tape index has to be synchronized in between. The index synchronization can be time consuming especially if there are a lot of files already on a tape. Therefore, the synchronization should happen not too often to not impact the performance and tape storage utilization (other operations on that tape are blocked during that time and multiple versions of index are stored on tape using storage space that cannot be reused without tape reclaim). Spectrum Archive is using one single job queue. This queue can become huge if the index synchronization happens infrequently. This has impact on the performance since the jobs to stub a file stay in the queue until these are completely processed and therefore this queue can become large while the traversing of this single queue can take much time. The only way to resolve this issue is to move out the jobs for stubbing to additional stubbing job queues.

Deleted: since

These stubbing job queues are created on request. After a file has been premigrated the corresponding job is moved to a stubbing job queue. If there does not exist a corresponding stubbing job queue or if it is in finished/closed state (see “fin” column within the stubbing queue of the example) a new stubbing job queue is created. Therefore, for each premigration job queue there exists exactly one stubbing job queue where migration jobs can be moved to.

Before the tape index synchronization happens all corresponding stubbing job queues are marked finished/closed to lock them. In this state is not possible to add any further jobs to these queues. After the tape index synchronization is finished: for files listed within these finished/closed stubbing job queues the stubbing operation is performed.

Deleted: and thereafter corresponding jobs are removed from these queues.

There exists an additional stubbing queue that provides information about all stubbing job queues that exist.

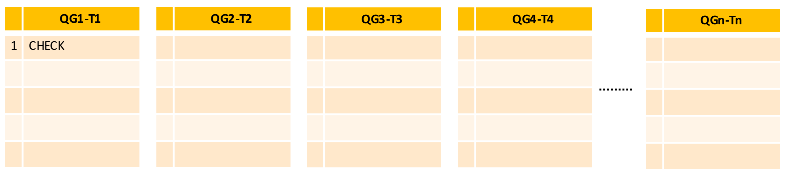
Corresponding jobs for files that have been completely processed can be either removed from the job queues (files to be premigrated removed from the premigration job queue or files to be migrated removed from the stubbing job queue) or these jobs can stay until the migration request is completely processed. It will be beneficial for a user if completed job continue to stay in the job queues since queries on the migration requests can provide more and better information about individual jobs. With an increasing number of files being processed by a migration request the memory consumption also is increasing. This fact might require to remove jobs from the queues and only to provide statistics.

Formatted: Highlight

Deleted: probably will

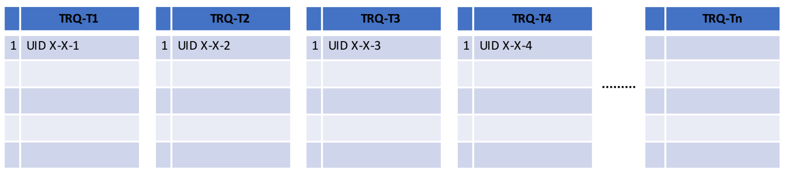
Formatted: Highlight

2.7.2.3. Generic Job Queues:



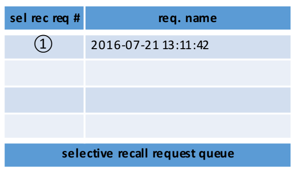
A generic request can be an operation to check, to format, or to reclaim space on a tape. A generic request always is related to a specific tape. There are as many generic job queues as tapes are assigned to an Open LTFS instance. In this example there exists one generic job to check tape T1.

2.7.2.4. Transparent Recall Job Queues:

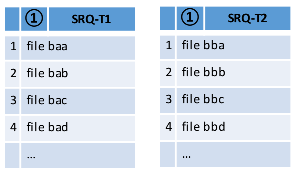


Similar like for generic requests the number of transparent recall job queues corresponds to the number of known tapes. According the DMAPI event system the only available information for transparent recalls is the UID (file system id, i-node generation number, i-node number). Therefore, the transparent recall job queues contain these UIDs instead of file or path names. In this example there are four transparent recall requests – each to a different tape.

2.7.2.5. Selective Recall Job Queues



sel rec req #	req. name
①	2016-07-21 13:11:42
selective recall request queue	

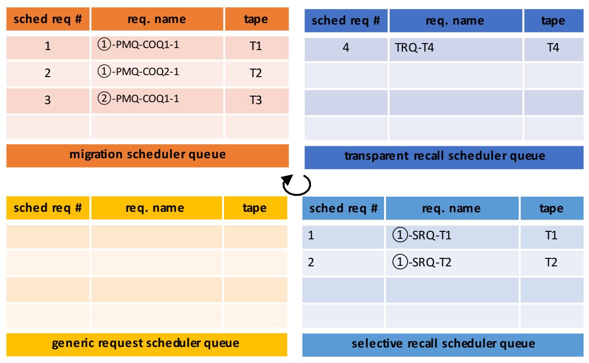


	①	SRQ-T1
1	file baa	
2	file bab	
3	file bac	
4	file bad	
	...	

	①	SRQ-T2
1	file bba	
2	file bbb	
3	file bbc	
4	file bbd	
	...	

Selective recall requests are performed on a predefined set of file system objects. In this example there exists one such request: ①2016-07-21 13:11:42 has been issued on a set of files with a pattern b[ab][abcd..]. The file names starting with “ba” are located on tape T1 – those starting with bb are located on tape T2. Therfore two selective recall job queues have been created: one for each tape.

2.7.2.6. Scheduler Queues:



sched req #	req. name	tape
1	①-PMQ-COQ1-1	T1
2	①-PMQ-COQ2-1	T2
3	②-PMQ-COQ1-1	T3

migration scheduler queue

sched req #	req. name	tape
4	TRQ-T4	T4

transparent recall scheduler queue

sched req #	req. name	tape

generic request scheduler queue

sched req #	req. name	tape
1	①-SRQ-T1	T1
2	①-SRQ-T2	T2

selective recall scheduler queue

To simplify the processing compared to Spectrum Archive the scheduler picks up job queues (job queues have been already mentioned before) and let them being processed. It does not look at particular jobs within the job queues.

If there already exist several non-empty job queues a decision is still required which of the job queues should be scheduled first. For that reason, intermediate queues have been introduced: the so called scheduler queues.

If a new job is added to any of the previously mentioned job queues and this job queue before was empty this job queue is added to one of the four scheduler queues:

- transparent recall scheduler queue
- selective recall scheduler queue
- generic request scheduler queue
- migration scheduler queue

Therefore, these scheduler queues become some kind of “queues of queues”. The entities of these scheduler queues are job queues.

In the example above there is one transparent recall job queue added to the transparent recall scheduler queue, one selective recall job queue added to the selective recall scheduler queue, and three migration job queues to the migration scheduler queue. No job queues were added to the generic scheduler queue.

If a drive is free the scheduler looks for job queues within the scheduler queues in the same order like listed above, i.e. thetransparent recalls have the highest priority (see also ⟲).

Deleted:

For each of the entities of the scheduler queues a scheduler request number (sched req #) is added. In the case a scheduler request (job queue) gets scheduled it is moved out the corresponding scheduler queue to the target drive.

If a scheduler request (job queue) with a higher priority needs to take the same resource that a lower priority scheduler request is actually using this lower priority request gets moved back into its scheduler queue. In this case it will be added according its scheduler request number (not necessarily at the bottom) to the corresponding scheduler queue.

An example:

There are three migration queues and one of the migration queues is in progress on drive 3:

sched req #	req. name	tape
3	②-PMQ-COQ1-1	T3

DRIVE 3

sched req #	req. name	tape
1	①-PMQ-COQ1-1	T1
2	①-PMQ-COQ2-1	T2

migration scheduler queue

sched req #	req. name	tape

transparent recall scheduler queue

During the time migration queue ②-PMQ-COQ1-1 is in progress on drive 3 a transparent recall queue appears for the same tape T3 that actually is used:

sched req #	req. name	tape
3	②-PMQ-COQ1-1	T3

DRIVE 3

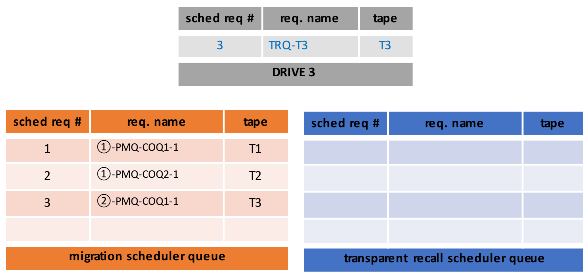
sched req #	req. name	tape
1	①-PMQ-COQ1-1	T1
2	①-PMQ-COQ2-1	T2

migration scheduler queue

sched req #	req. name	tape
3	TRQ-T3	T3

transparent recall scheduler queue

Since migration and transparent recall need the same resources and transparent recall has a higher priority the migration processing is interrupted and its queue is moved back to the migration scheduler queue at a position according its scheduler request number and the transparent recall queue gets scheduled:



sched req #	req. name	tape
3	TRQ-T3	T3

DRIVE 3

sched req #	req. name	tape
1	①-PMQ-COQ1-1	T1
2	①-PMQ-COQ2-1	T2
3	②-PMQ-COQ1-1	T3

migration scheduler queue

sched req #	req. name	tape

transparent recall scheduler queue

It still is being discussed if the scheduler request number should be global or specific for each scheduler request queue. The latter probably makes more sense.

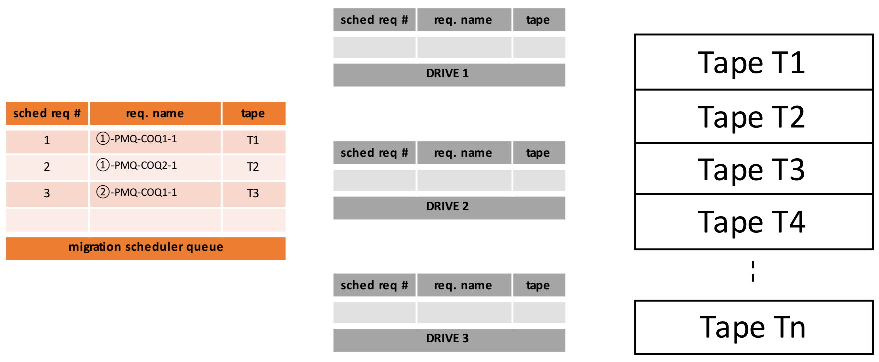
For migration queues initially there is no tape specified. A tape gets assigned to a migration queue as soon as it is added to the migration scheduler queue.

2.8. The Scheduler

The scheduler

- is assigning queues from the scheduler queues to drives,
- is mounting the tapes,
- and is starting the corresponding operation.

If e.g. there are three migration queues within the migration scheduler queue, three drives and corresponding tapes available



sched req #	req. name	tape
1	①-PMQ-COQ1-1	T1
2	①-PMQ-COQ2-1	T2
3	②-PMQ-COQ1-1	T3

migration scheduler queue

sched req #	req. name	tape

DRIVE 1

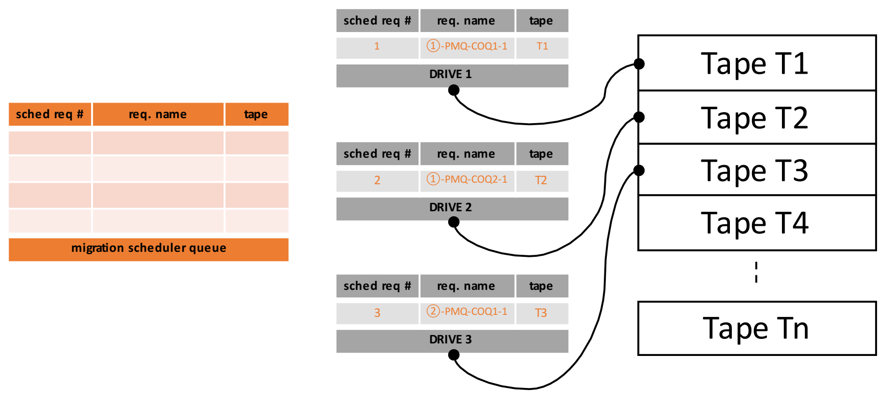
sched req #	req. name	tape

DRIVE 2

sched req #	req. name	tape

DRIVE 3

the migration queues are moved to each of the drives and tapes are mounted:



These migration queues do not appear anymore within the migration scheduler queue.

There can be two events that can let the scheduler starts acting:

I. A scheduler request (queue) has been added to the first position of a scheduler queue. It may either be newly added to the scheduler queue or it has been moved from the second position to the first. If this happens the scheduler starts looking for appropriate resources:

    1. If the corresponding tape is in use and the queue currently processed on that tape has equal or higher priority the queue cannot be scheduled.
    2. If all available drives are in use and the queues currently being processed have equal or higher priority the queue also cannot be scheduled.
    3. If contrarily a corresponding other queue has lower priority its processing gets stopped and it is moved back to its scheduler queue (at a position according its scheduler request number) and the corresponding drive becomes free.
    4. If a drive is free but a different (than the required) tape is just mounted the tape gets unmounted.
    5. If a drive is empty, the required tape gets mounted.
    6. If a drive has mounted the tape that it is needed the queue is moved to that drive and processing starts.

II. If a drive becomes free the scheduler is looking within the scheduler queues for an appropriate queue to process. Since there exist four scheduler queues it is performed in the following order based on priority:

    - transparent recall scheduler queue
    - selective recall scheduler queue
    - generic request scheduler queue
    - migration scheduler queue

    If there is a queue to be scheduled steps 4 to 6 from previous list apply in this case.

It needs to be satisfied that these two operations (I. and II.) do not run concurrently. A locking mechanism needs to be established.

2.9. Executor

Beside the assignment of queues to a drive and beside tape mounts the scheduling also includes to execute specific operations. For some of the operations external commands need to be started. A design target is to minimize the use of external commands. If possible an API should be used which gives better (and a better defined) control over the operation to be executed.

2.9.1. LTFS operations

Open LTFS is based on LTFS LE (LTFS Library Edition). Spectrum Archive uses LTFS LE+ a clustered version of LTFS LE. For Open LTFS clustering functionality is not required. The primary use case for Open LTFS is SWIFT that already provides clustering capabilities.

Some functionality of Open LTFS require operations on LTFS LE. There are operations invoked by the user like:

- to format a tape or
- to check a tape
- etc.

that are passed to the LTFS LE layer. There are other functions used internally that also require activity on the LTFS LE side like mounting, unmounting a tape, or writing the index. Formatting a tape also is internally necessary: at the end of a reclamation operation.

To determine the tape and drive inventory LTFS LE functionality also is required.

LTFS LE provides four ways of interaction:

- Formatting and checking of a tape can only be performed by an **external command**.
- The LTFS LE inventory can be obtained by using the so called **Out Of Band protocol** to talk to the LTFS LE daemon. To mount and unmount a tape the Out Of Band protocol also is used.
- For writing the tape index a specific **Extended Attribute** of the tape directory within LTFS LE is necessary to set.
- The use of the **POSIX API** for file system operations.

LTFS LE provides much more functionality compared to that what here has been specified so far to be used in Open LTFS. Additional functionality might be required.

2.9.2. API + connector

Until now there is no final decision which technology to use to implement the HSM (hierarchical storage management) functionality. As for now there are two methods available:

- the DMAPI (Data Storage Management (XDSM) API) interface for the XFS file system type.
- a FUSE file system layer developed by the German company BDT Media Automation GmbH. This software is already available according an open source license.

To be able to support different technologies HSM functionality is encapsulated by an API. This API provides the following:

- premigration of a file
- recall of a file
- reclamation of a tape
- stubbing of a pre-migrated file
- information about the migration state of a file

Since this design is not complete the API might be extended.

A further question that came up was on which level this API should be provided. The current proposal is on a quite high level (premigration of a file, recall of a file, …). There are also other possibilities to do have the API on a lower level like to provide a function for reading from disk that cover the different two technologies: i.e. cover the dm_read_invis() DMAPI call and its corresponding FUSE part. Providing an API on a lower level would probably lead to more common code. E.g. if the call reading from disk is covered by an API the remaining code like performing this read calls within a loop and writing the data to tape could be performed commonly. If the whole premigration code is done via API some parts of the code have to be performed in the same way for the two version of the premigration code. A decision on the right level to use depends very much on the similarity of the two technologies. If e.g. the whole copy process from disk to tape is very different an API at a low level cannot be used. Since our knowledge regarding the current FUSE BDT solution is very limited we will start on some high level function like listed above.

Beside reclamation, formatting and checking tape the other functions are on file level. The level of reclamation needs to be investigated (e.g. tape vs. file).

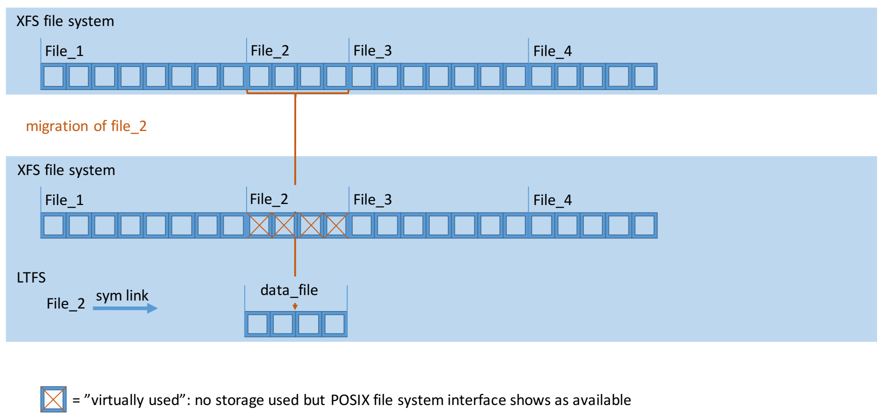
Comment [SS8]: Tape space can be reused only if entire tape is reclaimed than formatted. So it seems this should always be on tape level, but we might want to allow interrupting/resuming a tape reclamation after each file or after each X GB reclaimed, in order to serve recalls that might be requested from the tape being reclaimed. Also provide some statistic during reclamation (#files, space copied so far).

2.9.3. DMAPI

The DMAPI of for the XFS file system type (as well as for others) provides an interface to implement HSM software.

The following chart shows how migration works regarding DMAPI implementation:

migration – XFS DMAPI



During a migration data is copied to tape and thereafter the space of file data is reclaimed. All these operations happen below the virtual file system layer. The POSIX file system interfaces still show the same file size after migration. If a migrated file is accessed Open LTFS is notified by the DMAPI event system and corresponding read, write, or truncate calls within a corresponding user space application that initiated the recall are blocked until the data is back on disk.

Comment [SS9]: We should mention streaming or partial recall as possible future work, if those are generally supported by XFS dmapi.

There are two components of the DMAPI that need a proper cleanup since those are persistent after DMAPI program termination without doing such a cleanup.

Deleted: according

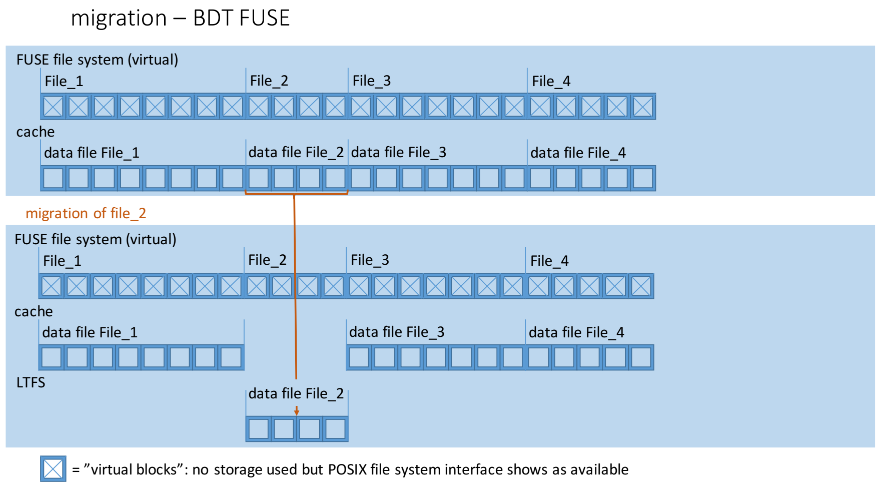
- If DMAPI sessions do not get destroyed before termination of a corresponding process these will remain until reboot of the system. DMAPI sessions claim system resources. Over the time the system can run out of resources.
- DMAPI locks (unlike POSIX locks which automatically got removed if a corresponding process terminate) are persistent even if a corresponding process ends without removing this lock. This can lead to inaccessible files until remount of the file system these files are located.

There does not exist any function within DMAPI that provides support for these two topics. Open LTFS has to implement such cleanup facilities.

2.9.4. FUSE

The BDT FUSE solution implements a virtual file system that aggregates a disk file system as a cache and LTFS SE. By using a virtual file system an appropriate level can be chosen about how much information is shown regarding the underlying storage (disk or tape). Data movement from disk to tape can be performed invisible to the user.

The following chart shows how migration works regarding BDT FUSE implementation:



If data is written to the FUSE file system, it first is stored within the cache file system. During a migration data is moved from the cache file system to LTFS. A transparent recall can be initiated by read, write, or truncate (not sure about truncate) call within a user space application. Those calls are blocked until the data is moved back to the cache (Not sure about the latter since it is also possible to read directly from tape without moving back data to disk).

For a BDT provided FUSE implementation of Open LTFS the BDT code has to be partly changed. The BDT solution is using LTFS SE (LTFS Single Drive Edition) and implements its own library manager. Unlike LTFS LE it does not support the most generic tape library SCSI interface and therefore it does not support all tape libraries e.g. it does not support IBM TS4500 tape library. By changing to LTFS LE the BDT provided library manager is not needed anymore.

2.9.5. DMAPI or FUSE

Both solutions XFS DMAPI and BDT FUSE have advantages and disadvantages:

XFS DMAPI:

- + Mature code that has been originally implemented for the IRIX operating system many years ago.
- + Users can re-use their existing file system if it is a XFS one (needs to be verified). XFS is a commonly used file system type.
- + Experience is already available according the Spectrum Scale DMAPI for Spectrum Archive.

BDT FUSE:

- + More and better control compared to the fixed DMAPI interface. Additional functionality can be implemented on demand within the FUSE virtual file system whereas the DMAPI function calls are fixed.
- - Additional work required to fully understand the source code to adapt it to our requirements.

Our current position is to work on a XFS DMAPI implementation.

2.10. Configurator

There are two different components to be configured:

- LTFS LE: There are specifics to be configured e.g. which library to use or which drives to use for LTFS LE. LTFS LE provides configuration for such kind of settings and Open LTFS is operating on an already configured LTFS LE system.
- Open LTFS: LTFS LE is not designed to use in a clustering environment. It needs to be satisfied that a single tape is only used on one single node in a clustering environment. There might be further specifics subject of an Open LTFS configuration.

On each node there exists a configuration file with a list of tapes and drives available for that node. Like usually on Linux this configuration file is located within the `/etc/OpenLTFS` directory.

A question still exists how to satisfy that a single tape does not appear in the configuration file of different nodes. There are the following possibilities:

Comment [SS10]: Another option is that user must configure one (logical) tape library to be used by the node. OpenLTFS queries LTFS LE for drives/nodes used by the library. Smaller probability of misconfiguration. Further 'trick' would be to require logical tape library to be named after the node name by default, and allow user to change the config mode to specifying arbitrary lib name or to specifying list of tapes/drives.
Design is that OpenLTFS not be aware of other nodes that might be connected to the same library, and it is responsibility of user to not assign same drives/tapes to different nodes (if tape library is shared).

- The user has to take care during the configuration of Open LTFS. A disadvantage of this is that in the case the user has wrongly configured the tape distribution Open LTFS will show up severe issues during operation.
- Open LTFS provides functionality to satisfy that this will not happen. This would require to implement clustering capabilities with Open LTFS the we would like to avoid. To keep the Open LTFS design simple clustering capabilities should be only provided by other software like SWIFT.
- This other software – like SWIFT – will take case that a single tape is only used on a single node. There would be an interaction required between Open LTFS and this software. A design idea of Open LTFS is to run fully independently. An interaction with e.g. SWIFT would be opposed to that principle.

2.11. Status and Statistics

The status and statistics component should provide information about the status and the statistics regarding the following:

- the tape drives: `ltfsdm info drives`.
- the migration state of a file: `ltfsdm info files`.
- the jobs being processed: `ltfsdm info jobs`. It needs to discussed if this command is really useful (e.g. if a user migrates millions of files). At least - like within the current version of Spectrum Archive - it should show the information in an appropriate time frame. Another command might me more useful:
- the requests being processed: `ltfsdm info requests`. This command provides information about requests added to the request queues.
- the queues that exist and which are not empty: `ltfsdm info queues`. This command provides some overview about upcoming work. If there are no non-empty queues non of the queues is listed.
- the tape library: `ltfsdm info library`.
- the tapes: `ltfsdm info tapes`. Only tapes assigned to the node where this information is requested are shown.
- the status of the Open LTFS service: `ltfsdm status`.

For migration and selective recall requests it is not useful to show every particular file to be processed within the output if the number of files gets huge. If all file within a sub-tree of a file system should be migrated and the number of files is about several of millions a user is preliminary interested in numbers to reflect the progress instead of all file names within the output. The output of the `info request` commands contains the following information:

- request id
- description (or timestamp the initial commands has been issued)
- number of files in resident state
- number of files in premigrated state
- number of files in migrated state
- number of files that failed the operation
- object currently in progress (if any)

This information is shown for migration (also in the case premigration state as target: number of file in migration state remains 0 in this case) and also for selective recall (in the case premigration as target the number of files in resident state remains 0; in the case of resident as target number of files in premigrated state remains 0).

The `info jobs` command can be applied to any job queue. This information provided is the following:

- object name
- state (not in progress, in progress, etc tbd)
- throughput (if in progress)

Deleted: or to tape drives

Formatted: Font:Andale Mono

Formatted: Indent: First line: 0 cm

Formatted: Font:Andale Mono

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

Formatted: Indent: First line: 0 cm

Formatted: Font:Andale Mono

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

Formatted: Highlight

Formatted: Highlight

For other operations like reclamation there also might be a need for intermediate states.

For the `info files` command the migration state is shown:

- migrated
- premigrated
- resident

There are also transient states:

- resident -> premigrated
- premigrated -> migrated
- migrated -> premigrated
- migrated -> resident
- premigrated -> resident

These transient states are also reflected within the DMAPI attributes (or BDT FUSE equivalent). If e.g. a list of files is submitted for migration the whole list is sent to the Open LTFS service. One of the initial tasks that are performed by the back end is to change to a corresponding transient state.

Since the DMAPI attributes are persistent there needs to be set up a cleanup procedure (e.g if the Open LTFS back end process terminates unexpectedly - the attribute will remain). POSIX locks would be beneficial regarding cleanup but probably not possible to use in case the requests and corresponding job queues get large.

2.12. Supervisor

The operations started by the executor are supervised. If there is an operation being blocked some information needs to be provided to the user. For operations where there is no external command used this is possible. For external commands (e.g. some LTFS LE operations) there is no control from the Open LTFS service. This needs to be investigated.

Deleted: If a user initiates a lot of jobs a possibility needs to be provided to see the progress. For most for the operations the following information is sufficient: ... [1]

Deleted: an

Formatted: Font:Andale Mono

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

Formatted: Indent: First line: 0.63 cm

Formatted: Highlight

Comment [SS11]: We said one option is to store transient state along with the process id of ltfsdmd and to consider it valid only if the process is alive (maybe also use a timestamp to avoid risk of restarting ltfsdmd and getting same process id). It is not clear if we want to provide this function, because info is reliably available through ltfsdmd (CLI or API).

Most likely for Swift it is very useful to also be able to read current status (reliable one such as migrate/resident/premigrated) w/o having to go through ltfsdmd. A Swift config file may have an entry if this info is expected to find in normal EA or in DMAPI EA. (intercept and transparent read status EA even if stored in DMAPI, in OpenLTFS or in SwiftHLM).

Deleted: For migration and recall information about throughput is also provided. All this statistic information is provided with the corresponding `ltfsdm info` commands.

Page 29: [1] Deleted **Martin Petermann** **8/3/16 1:28:00 PM**

If a user initiates a lot of jobs a possibility needs to be provided to see the progress. For most for the operations the following information is sufficient:

the number of jobs not yet processed
the number of jobs being processed
the number of jobs finished successful
the number of jobs that failed

Since the migration operation is split into two phases: first a file gets pre-migrated and after the tape index is synchronized that file gets stubbed. Since there is an intermediate phase where the file resides as well on disk and on tape the following information also is available for migration:

the number of jobs in pre-migrated state.