Open LTFS design

1. Introduction 2

2. Overview 5

2.1. migration states 5

2.2. Front end commands 7

2.2.1. ltfsdm start/stop 7

2.2.2. ltfsdm add 7

2.2.3. ltfsdm status 7

2.2.4. ltfsdm migrate 8

2.2.5. ltfsdm recall 8

2.2.6. ltfsdm help 9

2.2.7. ltfsdm reclaim 9

2.2.8. ltfsdm config 9

2.2.9. ltfsdm check 9

2.2.10. ltfsdm format 9

2.2.11. ltfsdm info … 9

2.2.11.1. ltfsdm info requests 9

2.2.11.2. ltfsdm info files 10

2.2.11.3. ltfsdm info fs 10

2.3. Tracing and messaging 10

2.3.1. Messaging 10

2.3.2. Tracing 11

2.4. Communication between front end and back end service 12

2.5. Data serialization 12

2.6. Back end service 13

2.7. Receiver 14

2.8. MessageParser 14

2.9. SQLite database for non-persistent data 14

2.9.1. JOB\_QUEUE 15

2.9.2. REQUEST\_QUEUE 16

2.10. Inventory 17

2.11. Processing Overview 18

2.11.1. ① 18

2.11.2. ② 19

2.11.3. ③ 19

2.11.4. ④ 19

2.12. The Scheduler 20

2.13. The name space on tape 21

2.14. LTFS operations 21

2.14.1. API + connector 22

2.14.2. DMAPI 23

2.14.3. FUSE 24

2.14.4. DMAPI or FUSE 25

2.15. Configurator 25

2.16. Status and Statistics 26

2.17. Supervisor 27

# Introduction

IBM currently provides three ILM solutions on Linux to migrate data from disk to tape: Spectrum Protect HSM, HACMP?, and Spectrum Archive Enterprise Edition. These 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.

A further requirement was not to have a dependency on TSM HSM. This gives more flexibility on the design and platform coverage.

Since the requirement on this software are similar to those for Spectrum Archive Enterprise Edition in the following design it is also explained why to go a different way within some parts of the design.

To implement a ILM 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). We could implement a similar solution with a FUSE layer but based on LTFS LE (LTFS Library Edition).

There is a third technology that is worth being evaluated:

* Linux provides an event system that provides information about changes within a file system called inotify. As far as I’m aware of virus scanner are using this event system.

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.

# 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:



Within the following sub sections all these components are discussed.

# migration states

There exist three types of migration states:

* A file is resident: file data only exist on the disk file system.
* A file is in premigrated state: the same file data exist on the disk file system as well on tape.
* A file is in migrated state: the file data only exist on tape.

There are different operations that lead to a change of the migration state. The following picture gives an overview about these operations:



The migration operation is split into two phases. In a first phase the data is copied to tape (premigration). A file is in premigrated state after this first step. Since a new file on tape has been created in this phase the LTFS index must be written to tape (indicated by the sync box within the chart). Without writing the index the file information is not persistently available on tape. This needs to happen before the date is removed from the disk within the following stubbing phase. Otherwise there is a risk of losing the data.

It is possible to just premigrate a file or to stub a premigrated file which results is a migrated file. A file in premigrated state has the beneficial behavior that it is still possible to read but in the case space needs to be freed space can be reclaimed very fast because of the quick stubbing operation.

Recall is the reverse operation where it is possible to convert a file to premigrated state or to resident state.

If a migrated file is read it is always converted to premigrated state. Writing to a migrated file results in a recall operation to resident state.

# 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  add | to start or stop the Open LTFS service in background.  to add a file system for Open LTFS management. |
| 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.

# 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.

# ltfsdm add

ltfsdm add <mount point>

Add Open LTFS management to a file system specified by the mount point.

# ltfsdm status

ltfsdm status

Provides status information of the ltfsdm service. In the case the ltfsdm service is operating its process if is shown.

# ltfsdm migrate

ltfsdm migrate –h

ltfsdm migrate [-w] [-p] [-c <colocation factor>]

[-n <request number>] [-R <number of replicas:1..3>]

<file name> …

ltfsdm migrate [-w] [-p] [-c <colocation factor>]

[-n <request number>] [-R <number of replicas:1..3>]

-f <file list>

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. By specifying the colocation factor, it can be determined how many tapes can be used in parallel. It is possible to provide up to three replicas on different tapes.

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 |
| -n | request number of a previously started migration request |
| -c | the colocation factor determines the number of tapes used in parallel |
| -f | the file list that contains file names of files to be migrated |
| -R | number of replicas being created |

# ltfsdm recall

ltfsdm recall –h

ltfsdm recall [-w] [-r] [-n <request number>] <file name> …

ltfsdm recall [-w] [-r] [-n <request number>] -f <file list>

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.

Options:

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

# ltfsdm help

ltfsdm help

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

# ltfsdm reclaim

# ltfsdm config

# ltfsdm check

# ltfsdm format

# ltfsdm info …

# ltfsdm info requests

ltfsdm info requests -h

ltfsdm info requests

ltfsdm info requests [-w] [-n <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 |
| -n | request number to show information for |

# ltfsdm info files

ltfsdm info files -h

ltfsdm info files <file name> …

ltfsdm info files -f <file list>

The info files 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 |

# ltfsdm info fs

ltfsdm info fs

The info fs command is showing the file systems managed by Open LTFS represented by its mount points.

# Tracing and messaging

# 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[X|S|C|D]NNNN[I|W|E]

where

* [X|S|C|D]is used based on if the message is common (‘X’), the message is written by the OpenLTFS backend process (‘S’), by the client (‘C’), or the message is related to the connector (‘D’).
* 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.

All messages are specified in a single file messages.cfg located in the root directory of the source code. The syntax of a message definitions is similar to the format specifier of the C printf function:

LTFSDMC0005E "Wrong command '%s' specified.\n"

Internally the Boost Format library is used to provide a better type safety than the corresponding standard C function.

# Tracing

For tracing the following macro is used:

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

The following four trace levels are defined:

Trace::error

Trace::little

Trace::medium

Trace::much

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

TRACE(Trace::error, errno);

TRACE(Trace::little, argc);

can look like the following:

2017-01-04T12:16:38.275657:[026753:026809]:--------Migration.cc(0454):errno(2)

2017-01-16T17:18:49.012693:[010412:010412]:-----------ltfsdm.cc(0052):argc(3)

containing time stamp, process id and thread id in square brackets, file name, line number in parentheses, 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.

# Communication between front end and back end service

The communication is asynchronous.

Within Spectrum Archive Enterprise Edition 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 between the front end and the back end only happens within a single node. 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.

# Data serialization

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 is using 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)>.

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 request number,
* the client pid,
* the target state (migrated or premigrated),
* the number of replicas,
* amd the colocation factor.

can look like the following:

message LTFSDmMigRequest {

required uint64 key = 1;

required int64 reqNumber = 2;

required uint64 pid = 3;

enum State {

PREMIGRATED = 0;

MIGRATED = 1;

}

required State state = 4;

required int64 numReplica = 5;

required int64 colFactor = 6;

}

# Back end service

The Open LTFS service eventually performs 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

|  |  |
| --- | --- |
| Server | The Server is starting further back end threads and is waiting for their termination. Currently the following threads are started:   * the Scheduler thread * the Receiver thread * the TransRecall (transparent recall) thread |
| 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. After a new message is received a new MessageParser thread is started immediately that evaluates the message. |
| MessageParser | The MessageParser evaluates a message received by the Receiver and performs an appropriate action like adding jobs to the queues. |
| JOB\_QUEUE | The JOB\_QUEUE is a SQLite table. Its entries correspond to the files (filename, uid) to be migrated or recalled or identifiers for other operations. |
| REQUEST\_QUEUE | The REQUEST\_QUEUE is a SQLite table. Its entries correspond to single requests or operations, e.g. a migration command for 1000 files. |
| inventory | The inventory keeps information about hardware (tapes and drives). |
| Migration | * Adds migration jobs to the JOB\_QUEUE. * Creates one or more requests within the REQUEST\_QUEUE. * Processes the premigration and stubbing operations. |
| SelRecall | * Adds selective recall jobs to the JOB\_QUEUE. * Creates one or more requests within the REQUEST\_QUEUE. * Processes the selective recall operations. |
| TransRecall | * Waits for recall events. * Adds selective recall jobs to the JOB\_QUEUE. * Creates one or more requests within the REQUEST\_QUEUE. * Processes the transparent recall operations. |
| Scheduler | The Scheduler gets invoked is a resource (tape) gets free to use or new request has been created. If a new request can be addressed to a free resource a corresponding operation is scheduled. |
| 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. |
| Connector | The Connector is a library providing an API to encapsulate different data management technologies (DMAPI, FUSE). The reason to provide the same interfaces is to make it easy to exchange these technologies. Doing so we could start development without doing a determination. |

command. The Open LTFS service consists of the following components:

# Receiver

The Receiver listens on a socket for new messages/commands from the front end. After a message is received the corresponding connection is transferred to a new MessageParser thread to continue listening. Up to 40 (Const::MAX\_RECEIVER\_THREADS) messages can be processed at a time: up to 40 MessageParser threads can be started in parallel before blocking further parsing. The implementation of a limit is to avoid overloading the back end with front end requests. The number seems a reasonable value. It can be and should be changed if required.

# MessageParser

The MessageParser first determines the message type. Currently there the following messages are evaluated:

* requestNumber: Independent on the request issued by the front end a request number needs to be generated first. The reason for this number is to associate jobs (e.g. a single file to be migrated within the JOB\_QUEUE) to requests (e.g. a command to migrate 1000 files). Corresponding jobs (same request number, etc.) are processed together all at once. Providing a request number a front end command can query the back end according a specific request number. If it is required to run an operation fully asynchronously (e.g. the recall command should return immediately after being issued) the progress of the request can be queried by using this request number.
* stopMessage: to terminate the back end
* migrationMessage: to migrate one or more files
* selRecallMessage: to recall one or more files
* statusMessage: is used to query the status of the back end. If the back end is operating as expected the process id of the back end process is provided back to the front end.
* addMessage: to add data management to a file system

# SQLite database for non-persistent data

Contrarily to direct access media like hard disk access to tape storage needs to be organized. Even LTFS provides a POSIX like file system interface for tape read requests to files should be performed in the order these files are located on tape. If tapes are part of a tape library and data is required from different tapes the accesses needs to be organized in a way to limit the number of tape mounts.

An ILM solution like Open LTFS requires to queue read and write accesses to tapes to perform them in an optimal order. For Spectrum Archive Enterprise Edition, the Boost Multi-Index Container library was used. A difficulty with that library was that it is not safe to be used from multiple threads. Access functions had to be implemented with a locking mechanism. With the time, more complex functions were added to these access functions which should not happen. That was the reason for looking for an alternative that is thread safe.

SQLite is a database system that can be embedded into the final application (it’s not a client-server database system). Therefore, it can be also integrated into Open LTFS. The data base also can be stored into memory which is a requirement for non-persistent queues in respect to performance.

Two queues exist in Open LTFS:

* A job queue (SQLite table JOB\_QUEUE) that stores information about individual files to be migrated or recalled. Other operations beside migration and recall operations probably also will add entries to the job queue.
* A request queue (SQLite table REQUEST\_QUEUE) that stores information about a whole request. If the migration command is started to migrate 1000 files (e.g. ltfsdm migrate file.1 … file.1000) one entry will be added to the request queue and thousand entries added to the job queue. The Scheduler later-on is inspecting the request queue and schedules all jobs related to a request at once.

# JOB\_QUEUE

The JOB\_QUEUE table contains the following columns:

|  |  |  |  |
| --- | --- | --- | --- |
| column name | data type | possible values | description |
| OPERATION | INT NOT NULL | DataBase::TRARECALL DataBase::SELRECALL DataBase::MIGRATION | The currently possible operations are transparent recall, selective recall, and migration. |
| FILE\_NAME | CHAR(4096) |  | file name |
| REQ\_NUM | INT NOT NULL | non-negative integer | The request number of the corresponding request. |
| TARGET\_STATE | INT NOT NULL | defined in the protocol buffer description: e.g. for migration:  enum State {  PREMIGRATED = 0;  MIGRATED = 1;  } | Files can be migrated to premigrated state or to migrated state. Files can be recalled to premigrated state or to resident state. |
| REPL\_NUM | INT | integer | Number of the replica associated to this job. Up to three replicas can be created on different tapes. |
| COLOC\_GRP | INT | integer | Colocation group to spread migration requests among different tapes. |
| FILE\_SIZE | INT NOT NULL | non-negative integer | file size |
| FS\_ID | BIGINT NOT NULL | non-negative 64-bit integer | file system id |
| I\_GEN | INT NOT NULL | non-negative integer | file inode generation number |
| I\_NUM | BIGINT NOT NULL | non-negative 64-bit integer | file inode number |
| MTIME\_SEC | BIGINT NOT NULL | non-negative 64-bit integer | file modification time: seconds |
| MTIME\_NSEC | BIGINT NOT NULL | non-negative 64-bit integer | file modification time: nano seconds |
| LAST\_UPD | INT NOT NULL | non-negative integer | last update of this record |
| TAPE\_ID | CHAR(9) | nine-character string | tape id |
| FILE\_STATE | INT NOT NULL | FsObj::RESIDENT,  FsObj::PREMIGRATED,  FsObj::MIGRATED,  FsObj::FAILED | Current migration state of the file. For migration, the state can change in the following way two times:   1. FsObj::RESIDENT -> FsObj::PREMIGRATED 2. FsObj::PREMIGRATED -> FsObj::MIGRATED |
| START\_BLOCK | INT | non-negative integer | Starting block on tape (not applicable for migration. |

Not all columns are used for every operation. For transparent recalls, e.g. a file name is unknown.

The following constraints are set according uniqueness to fail in case of duplicate entries:

* CONSTRAINT JOB\_QUEUE\_UNIQUE\_FILE\_NAME UNIQUE (FILE\_NAME, REPL\_NUM)  
  The same file name and replication number only should appear once.
* CONSTRAINT JOB\_QUEUE\_UNIQUE\_UID UNIQUE (FS\_ID, I\_GEN, I\_NUM, REPL\_NUM)  
  The same file system id, inode generation number, inode number and replication number only should appear once.

# REQUEST\_QUEUE

The REQUEST\_QUEUE table contains the following columns:

|  |  |  |  |
| --- | --- | --- | --- |
| column name | data type | possible values | description |
| OPERATION | INT NOT NULL | DataBase::TRARECALL DataBase::SELRECALL DataBase::MIGRATION | The currently possible operations are transparent recall, selective recall, and migration. |
| REQ\_NUM | INT NOT NULL | non-negative integer | request number of this request |
| TARGET\_STATE | INT | defined in the protocol buffer description: e.g. for migration:  enum State {  PREMIGRATED = 0;  MIGRATED = 1;  } | Files can be migrated to premigrated state or to migrated state. Files can be recalled to premigrated state or to resident state. |
| NUM\_REPL | ??? | should be integer | Number of replicas: up to three replicas can be created on different tapes. |
| REPL\_NUM | INT | integer | number of this replica: up to three replicas can be created on different tapes. |
| COLOC\_GRP | INT | integer | Colocation group to spread migration requests among different tapes. |
| TAPE\_ID | CHAR(9) | nine-character string | tape id |
| TIME\_ADDED | INT NOT NULL | non-negative integer | The time the request has been added to the queue. |
| STATE | INT NOT NULL | DataBase::REQ\_NEW,  DataBase::REQ\_INPROGRESS,  DataBase::REQ\_COMPLETED | state of the request |

Not all columns are used for every operation. E.g. replication number is only used for migration.

The following constraint are set according uniqueness to fail in case of duplicate entries:

CONSTRAINT REQUEST\_QUEUE\_UNIQUE UNIQUE(REQ\_NUM, REPL\_NUM, COLOC\_GRP, TAPE\_ID)

An entry with the same request number, replication number, colocation group, and tape id sould only appear once.

# Inventory

The inventory component keeps information about elements and status of

* drives and
* tapes

# Processing Overview



# ①

This processing overview describes how migration and selective recall requests are processed from the front end to the back end.

The front end commands accept file names for migration and selective recall as a parameter list or as a file list. If it is provided by a file list, the file names are separated by new lines.

A request is transferred to the back end in three different steps:

* At first a request number is requested from the back end. This number is used to identify the request throughout the complete processing. It is planned to re-use a request number to add further jobs to a previously created request. Another plan is that the calling application provides a unique number. In both cases, there is no need for this step.
* Within a second step request specific information is sent to the back end. For migration it is the following:
  + a key that has been randomly generated during the start of the back end. This key is contained in a file Const::KEY\_FILE: /var/run/ltfsdm/OpenLTFS.key and only visible to the root user.
  + the previously received request number.
  + the process id of the client process.
  + the target state of the migration process.
  + the number of replicas to be generated.
  + the colocation factor.

For selective recall no information about replicas and no colocation factor is transferred to back end. The target states also differ: while for migration the target state can be premigrated or resident, for selective recall it can be premigrated or resident.

* Thereafter the file information is transferred. There are several messages sent to the back end if the number of files exceed 100000 (Const:: MAX\_OBJECTS\_SEND). The file names are transferred without any further information. An empty string denotes the end of the file list.

# ②

For each request a separate socket connection is used. If the Receiver on the back end side receives a new message it immediately starts a MessageParser thread to evaluate the messages. The evaluation is part of the MessageParser::run method.

First the jobs are added to the JOB\_QUEUE table by [Migration|SelRecall]::addJob methods.

For the migration operation for each replica, there is created a corresponding job within this table. E.g. two replicas should be created for a single file two entries are created within the table. The TAPE\_ID and START\_BLOCK columns are unused for migration.

For selective and transparent recall, there is not set REPL\_NUM and COLOC\_GRP columns since those are only used by migration processing. In case of a transparent recall DMAPI does not provide a file name. Therefore the FILE\_NAME column is not set.

After all jobs are created within the JOB\_QUEUE table the a request is added to the REQUEST\_QUEUE table by the [Migration|SelRecall]::addRequest methods. For selective and transparent recall there are no NUM\_REPL, REPL\_NUM , and COLOC\_GRP values set.

# ③

The Scheduler is informed by a condition that there is a new request (Scheduler::run method) and traverses the REQUEST\_QUEUE table by doing a SELECT query on the new requests:

SELECT OPERATION, REQ\_NUM, TARGET\_STATE, NUM\_REPL, REPL\_NUM, COLOC\_GRP, TAPE\_ID FROM REQUEST\_QUEUE WHERE STATE=DataBase::REQ\_NEW

ORDER BY OPERATION,REQ\_NUM;

The query provides the table for the new requests ordered by the operation type and the request number. From top to the bottom the requests are checked for a free resource (drive/tape). The first matching result not necessarily needs to be the request responsible for the current condition.

# ④

If the there is a matching free resource available for a request a new thread is created to execute the operation by calling the corresponding [Migration|SelRecall]::execRequest methods.

The data transfer happens according SELECT queries on the JOB\_QUEUE table.

# The Scheduler

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

1. 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.
2. 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.

# The name space on tape

# 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.

# 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).

# 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:



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.

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.

* 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.

# 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.

# 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.

# 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:

* 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.

# 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)

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.

# 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.