LTFS Data Management

0.4.12-master.2018-07-05T21:50:55

Generated by Doxygen 1.8.5

Thu Jul 5 2018 22:52:15

Contents

Chapter 1

Commands

LTFS Data Management consists of a client interface and a set of background processes that perform the processing. Beside the special case of transparent recall where requests are driven by file i/o all other operations are initiated by the client interface.

The following client interface commands exist:

```
commands:
      ltfsdm help              - gives an overview
      ltfsdm start             - start the LTFS Data Management service in background
      ltfsdm stop              - stop the LTFS Data Management service
      ltfsdm add               - adds LTFS Data Management management to a file system
      ltfsdm status            - provides information if the back end has been started
      ltfsdm migrate           - migrate file system objects from the local file system to tap
      ltfsdm recall            - recall file system objects back from tape to local disk
      ltfsdm retrieve          - synchronizes the inventory with the information provided by S
      ltfsdm version           - provides the version number of LTFS Data Management
info sub commands:
      ltfsdm info requests     - retrieve information about all or a specific LTFS Data Manage
      ltfsdm info jobs         - retrieve information about all or a specific LTFS Data Manage
      ltfsdm info files        - retrieve information about the migration state of file syster
      ltfsdm info fs           - lists the file systems managed by LTFS Data Management
      ltfsdm info drives       - lists the drives known to LTFS Data Management
      ltfsdm info tapes        - lists the cartridges known to LTFS Data Management
      ltfsdm info pools        - lists all defined tape storage pools and their sizes
pool sub commands:
      ltfsdm pool create       - create a tape storage pool
      ltfsdm pool delete       - delete a tape storage pool
      ltfsdm pool add          - add a cartridge to a tape storage pool
      ltfsdm pool remove       - removes a cartridge from a tape storage pool
```

To start LTFS Data Management the ltfsdm start is to be used:

```
[root@visp ~]# ltfsdm start
LTFSDMC0099I(0073): Starting the LTFS Data Management backend service.
LTFSDMX0029I(0062): LTFS Data Management version: 0.0.624-master.2017-11-09T10.57.51
LTFSDMC0100I(0097): Connecting.
LTFSDMC0097I(0141): The LTFS Data Management server process has been started with pid  27905.
```

1.1 ltfsdm help

The ltfsdm help command lists all available client interface commands.

parameters	description
-	-

Example:

```
[root@visp ~]# ltfsdm help
commands:
           ltfsdm help              - show this help message
           ltfsdm start             - start the LTFS Data Management service in background
           ltfsdm stop              - stop the LTFS Data Management service
           ltfsdm add               - adds LTFS Data Management management to a file system
           ltfsdm status            - provides information if the back end has been started
           ltfsdm migrate           - migrate file system objects from the local file system to tape
           ltfsdm recall            - recall file system objects back from tape to local disk
           ltfsdm retrieve          - synchronizes the inventory with the information
                                      provided by Spectrum Archive LE
           ltfsdm version           - provides the version number of LTFS Data Management
info sub commands:
           ltfsdm info requests     - retrieve information about all or a specific LTFS Data Management reques
           ltfsdm info jobs         - retrieve information about all or a specific LTFS Data Management jobs
           ltfsdm info files        - retrieve information about the migration state of file system objects
           ltfsdm info fs           - lists the file systems managed by LTFS Data Management
           ltfsdm info drives       - lists the drives known to LTFS Data Management
           ltfsdm info tapes        - lists the cartridges known to LTFS Data Management
           ltfsdm info pools        - lists all defined tape storage pools and their sizes
pool sub commands:
           ltfsdm pool create       - create a tape storage pool
           ltfsdm pool delete       - delete a tape storage pool
           ltfsdm pool add          - add a cartridge to a tape storage pool
           ltfsdm pool remove       - removes a cartridge from a tape storage pool
```

The corresponding class is HelpCommand.

1.2 ltfsdm start

The ltfsdm start command starts the LTFS Data Management service.

`usage:  ltfsdm start`

parameters	description
-	-

Example:

```
[root@visp ~]# ltfsdm start
LTFSDMC0099I(0073): Starting the LTFS Data Management backend service.
LTFSDMX0029I(0062): LTFS Data Management version: 0.0.624-master.2017-11-09T10.57.51
LTFSDMC0100I(0097): Connecting.
LTFSDMC0097I(0141): The LTFS Data Management server process has been started with pid  13378.
```

The corresponding class is StartCommand.

1.3 ltfsdm stop

The ltfsdm stop command stops the LTFS Data Management service.

`usage:  ltfsdm stop [-x]`

parameters	description
-x	force the stop of LTFS Data Management even a managed file system is in use

Example:

```
[root@visp ~]# ltfsdm stop
The LTFS Data Management backend is terminating.
Waiting for the termination of the LTFS Data Management server...........
[root@visp ~]#
```

The corresponding class is StopCommand.

1.4 ltfsdm add

The ltfsdm add command is used to add file system management with LTFS Data Management to a particular file system.

`usage: ltfsdm add <mount point>`

parameters	description
<mount point>	path to the mount point of the file system to be managed

Example:

```
[root@visp ~]# ltfsdm add /mnt/xfs
```

The corresponding class is AddCommand.

1.5 ltfsdm status

The ltfsdm status command provides the status of the LTFS Data Management service.

`usage: ltfsdm status`

parameters	description
-	-

Example:

```
[root@visp ~]# ltfsdm status
LTFSDMC0032I(0068): The LTFS Data Management server process is operating with pid  13378.
```

The corresponding class is StatusCommand.

1.6 ltfsdm migrate

The ltfsdm migrate command is used to migrate one or more files to tape.

`usage:`

`ltfsdm migrate -h`

`ltfsdm migrate [-p] [-P <pool list: 'pool1,pool2,pool3'>] [-n <request number>] <file name> ...`

`ltfsdm migrate [-p] [-P <pool list: 'pool1,pool2,pool3'>] [-n <request number>] -f <file list>`

parameters	description
-p	to premigrate files, without specifying this option files get migrated

-P <pool list: 'pool1,pool2,pool3'>	a file can be migrated up to three different tape storage pools in parallel, at least one tape storage pool needs to be specified
-n <request number>	attach to an ongoing migration request to see its progress
<file name>	a set of file names of files to be migrated
-f <file list>	a file name containing a list of files to be migrated

Example:

```
[root@visp sdir]# find dir.* -type f |ltfsdm migrate -P pool1 -f -
--- sending completed within 385 seconds ---
                resident  transferred  premigrated     migrated       failed
[00:06:44]        989202        10798            0            0            0
[00:06:54]        980801        19199            0            0            0
[00:07:06]        972900        27100            0            0            0
[00:07:16]        949856        50144            0            0            0
[00:07:26]        908483        91517            0            0            0
[00:07:36]        867137       132863            0            0            0
[00:07:46]        825946       174054            0            0            0
[00:07:56]        784595       215405            0            0            0
[00:08:06]        743378       256622            0            0            0
[00:08:16]        702042       297958            0            0            0
[00:08:26]        660813       339187            0            0            0
[00:08:36]        619431       380569            0            0            0
[00:08:46]        586392       413608            0            0            0
[00:08:56]        545061       454939            0            0            0
[00:09:06]        503781       496219            0            0            0
[00:09:16]        462584       537416            0            0            0
[00:09:26]        421563       578437            0            0            0
[00:09:36]        380156       619844            0            0            0
[00:09:46]        342861       657139            0            0            0
[00:09:56]        321660       678340            0            0            0
[00:10:06]        286385       713615            0            0            0
[00:10:16]        245078       754922            0            0            0
[00:10:26]        203848       796152            0            0            0
[00:10:36]        181600       818400            0            0            0
[00:10:46]        140372       859628            0            0            0
[00:10:56]         99105       900895            0            0            0
[00:11:06]         58311       941689            0            0            0
[00:11:16]         17043       982957            0            0            0
[00:12:21]             0      1000000            0            0            0
[00:12:35]             0       941128            0        58872            0
[00:12:45]             0       876747            0       123253            0
[00:12:55]             0       812104            0       187896            0
[00:13:05]             0       747845            0       252155            0
[00:13:15]             0       684347            0       315653            0
[00:13:25]             0       619911            0       380089            0
[00:13:35]             0       555271            0       444729            0
[00:13:45]             0       491031            0       508969            0
[00:13:55]             0       426315            0       573685            0
[00:14:05]             0       362974            0       637026            0
[00:14:15]             0       299364            0       700636            0
[00:14:25]             0       234782            0       765218            0
[00:14:35]             0       170123            0       829877            0
[00:14:45]             0       105773            0       894227            0
[00:14:55]             0        41364            0       958636            0
[00:15:09]             0            0            0      1000000            0
[00:15:22]             0            0            0      1000000            0
```

The corresponding class is MigrateCommand.

1.7 ltfsdm recall

The ltfsdm recall command is used to selectively recall one or more files from tape.

`usage:`

`ltfsdm recall -h`

```
ltfsdm recall [-r] [-n <request number>] <file name> ...
ltfsdm recall [-r] [-n <request number>] -f <file list>
```

parameters	description
-r	to recall files to resident state, without specifying this option files get recalled to premigrated state
-n <request number>	attach to an ongoing recall request to see its progress
<file name>	a set of file names of files to be recalled
-f <file list>	a file name containing a list of files to be recalled

Example:

```
[root@visp sdir]# time -p find dir.* -type f |ltfsdm recall -f -
--- sending completed within 470 seconds ---
              resident  transferred  premigrated     migrated       failed
[00:08:07]           0            0        19089       980911            0
[00:08:17]           0            0        45460       954540            0
[00:08:27]           0            0        71614       928386            0
[00:08:37]           0            0        97978       902022            0
[00:08:47]           0            0       124256       875744            0
[00:08:57]           0            0       150820       849180            0
[00:09:07]           0            0       177126       822874            0
[00:09:17]           0            0       203667       796333            0
[00:09:27]           0            0       230053       769947            0
[00:09:37]           0            0       256636       743364            0
[00:09:47]           0            0       282795       717205            0
[00:09:57]           0            0       309180       690820            0
[00:10:07]           0            0       335801       664199            0
[00:10:17]           0            0       362352       637648            0
[00:10:27]           0            0       388731       611269            0
[00:10:37]           0            0       414920       585080            0
[00:10:47]           0            0       441328       558672            0
[00:10:57]           0            0       467469       532531            0
[00:11:07]           0            0       494076       505924            0
[00:11:17]           0            0       520329       479671            0
[00:11:27]           0            0       546638       453362            0
[00:11:37]           0            0       572859       427141            0
[00:11:47]           0            0       599138       400862            0
[00:11:57]           0            0       625164       374836            0
[00:12:07]           0            0       651295       348705            0
[00:12:17]           0            0       677979       322021            0
[00:12:27]           0            0       704257       295743            0
[00:12:37]           0            0       730872       269128            0
[00:12:47]           0            0       757410       242590            0
[00:12:57]           0            0       784288       215712            0
[00:13:07]           0            0       810809       189191            0
[00:13:17]           0            0       837339       162661            0
[00:13:27]           0            0       863603       136397            0
[00:13:37]           0            0       889898       110102            0
[00:13:47]           0            0       916118        83882            0
[00:13:57]           0            0       942182        57818            0
[00:14:07]           0            0       968286        31714            0
[00:14:17]           0            0       994684         5316            0
[00:14:27]           0            0      1000000            0            0
[00:14:40]           0            0      1000000            0            0
```

The corresponding class is RecallCommand.

1.8 ltfsdm retrieve

The ltfsdm retrieve command synchronizes the status of drives and cartridges with Spectrum Archive LE.

```
usage:  ltfsdm retrieve
```

parameters	description
-	-

Example:

```
[root@visp ~]# ltfsdm retrieve
[root@visp ~]#
```

The corresponding class is RetrieveCommand.

1.9 ltfsdm version

The ltfsdm version command provide the version information of the LTFS Data Management software.

`usage: ltfsdm version`

parameters	description
-	-

Example:

```
[root@visp ~]# ltfsdm version
LTFS Data Management version: 0.0.624-master.2017-11-09T10.57.51
```

The corresponding class is VersionCommand.

1.10 ltfsdm info requests

The ltfsdm info request command lists all LTFS Data Management requests and their corresponding status.

`usage:`

`ltfsdm info requests -h`

`ltfsdm info requests`

`ltfsdm info requests [-n <request number>]`

parameters	description
-n <request number>	request number for a specific request to see the information

Example:

```
[root@visp ~]# ltfsdm info requests -n 28
operation          state              request number     tape pool          tape id            targe
migration          in progress        28                 pool1              D01301L5           in pr
```

The corresponding class is InfoRequestsCommand.

1.11 ltfsdm info jobs

The ltfsdm info jobs command lists all jobs that are currently processed by the backend.

`usage:`

`ltfsdm info jobs -h`

`ltfsdm info jobs`

`ltfsdm info jobs [-n <request number>]`

parameters	description
-n <request number>	restrict the jobs to be displayed to a certain request

Example:

```
[root@visp ~]# ltfsdm info jobs -n 2
operation          state              request number     tape pool          tape id            size
migration          transferring       2                  pool1              DV1462L6           10737
migration          transferring       2                  pool1              DV1462L6           10737
migration          transferring       2                  pool1              DV1462L6           10737
migration          transferring       2                  pool1              DV1462L6           10737
migration          transferring       2                  pool1              DV1462L6           10737
migration          transferring       2                  pool1              DV1462L6           10737
migration          transferring       2                  pool1              DV1462L6           10737
migration          transferring       2                  pool1              DV1462L6           10737
migration          transferring       2                  pool1              DV1462L6           10737
migration          transferring       2                  pool1              DV1462L6           10737
```

The corresponding class is InfoJobsCommand.

1.12 ltfsdm info files

The ltfsdm info files command provides information about the migration status of one or more files.

```
usage:
ltfsdm info files -h
ltfsdm info files [-v] <file name> ...
ltfsdm info files [-v] -f <file list>
```

parameters	description
-v	verbose output to show the attributes stored at (pre)migrated files
<file name> ...	a set of file names to get the migration status
-f <file list>	the name of a file containing file names to get the migration status

Example:

```
[root@visp ~]# ls /mnt/lxfs/bigfile* |ltfsdm info files -f -
state            size              blocks           tape id   file name
m          47049088000                  8          D01301L5   /mnt/lxfs/bigfile
r          47049088000           91892752                 -   /mnt/lxfs/bigfile.1
r          47049088000           91892752                 -   /mnt/lxfs/bigfile.1.cpy
r          47049088000           91892752                 -   /mnt/lxfs/bigfile.cp
```

The migration states are:

state	description
m	migrated
p	premigrated
r	resident

The corresponding class is InfoFilesCommand.

1.13 ltfsdm info fs

The ltfsdm info fs command lists all file systems that are managed with LTFS Data Management:

```
usage:
ltfsdm info fs -h
```

`ltfsdm info fs`

Example:

```
[root@visp ~]# ltfsdm info fs
device              mount point         file system type    mount options
/dev/sdc1           /mnt/lxfs           xfs                 rw,relatime,attr2,inode64,noquota
```

The columns have the following meaning:

column	meaning
device	device name of the file system managed with LTFS Data Management
mount point	mount point where the Fuse overlay file system is mounted
file system type	file system type of the file system managed with LTFS Data Management
mount options	mount options of the file system managed with LTFS Data Management

The corresponding class is InfoFsCommand.

1.14 ltfsdm info drives

The ltfsdm info drives command lists all available tape drives.

`usage:`

`ltfsdm info drives -h`

`ltfsdm info drives`

parameters	description
-	-

Example:

```
[root@visp ~]# ltfsdm info drives
id            device name    slot          status        usage
9068051229    /dev/IBMtape0  256           Available     free
1013000505    /dev/IBMtape1  259           Available     free
```

The corresponding class is InfoDrivesCommand.

1.15 ltfsdm info tapes

The ltfsdm info tapes command lists all available cartridges and corresponding space information.

`usage:`

`ltfsdm info tapes -h`

`ltfsdm info tapes`

parameters	description
-	-

Example:

```
[root@visp ~]# ltfsdm info tapes
id              slot            total           remaining       reclaimable     in progress     status
                                capacity (GiB)  capacity (GiB)  estimated (GiB) (GiB)
DV1462L6        4112            0               0               0               0               not mounted ye
```

```
DV1463L6        4096            0               0               0               0               not mounted ye
DV1464L6        257             2242            2222            1               0               writable
DV1465L6        4102            0               0               0               0               not mounted ye
DV1466L6        4115            0               0               0               0               not mounted ye
DV1467L6        4110            2242            2072            178             0               writable
DV1468L6        256             2242            2238            2               0               writable
```

The corresponding class is InfoTapesCommand.

1.16 ltfsdm info pools

The ltfsdm info pools command provides information about tape storage pools and the number of assigned tapes.

`usage:`

`ltfsdm info pools -h`

`ltfsdm info pools`

parameters	description
-	-

Example:

```
pool name     links         total cap.    rem. cap.     unref. cap.   #tapes
pool1         no            2296532       2296529       0             1
pool2         yes           2296532       2296529       0             1
pool3         no            2296532       2296529       0             1
```

The corresponding class is InfoPoolsCommand.

1.17 ltfsdm pool create

The ltfsdm pool create command creates a tape storage pool.

`usage:`

`ltfsdm pool create -h`

`ltfsdm pool create [-l] -P <pool name>`

parameters	description
-l	create symbolic links on tapes added to that pool during migration
-P <pool name>	pool name of the tape storage pool to be created

Example:

```
[root@visp ~]# ltfsdm pool create -P newpool
Pool "newpool" successfully created.
```

The corresponding class is PoolCreateCommand.

1.18 ltfsdm pool delete

The ltfsdm pool delete command deletes a tape storage pool.

`usage:`

`ltfsdm pool delete -h`

`ltfsdm pool delete -P <pool name>`

parameters	description
-P <pool name>	pool name of the tape storage pool to be deleted

Example:

```
[root@visp ~]# ltfsdm pool delete -P newpool
Pool "newpool" successfully deleted.
```

The corresponding class is PoolDeleteCommand.

1.19 ltfsdm pool add

The ltfsdm pool add command adds a cartridge to a tape storage pool.

`usage:`

`ltfsdm pool add -h`

`ltfsdm pool add [-F|-C] -P <pool name> -t <tape id> [-t <tape id> [...]]`

parameters	description
-F	format a cartridge before add to a tape storage pool
-C	check a cartridge before add to a tape storage pool
-P <pool name>	pool name to which a cartridge should be added
-t <tape id>	id of a cartridge to be added

Example:

```
[root@visp ~]# ltfsdm pool add -P 'large pool' -t DV1478L6
Tape DV1478L6 successfully added to pool "large pool".
```

The corresponding class is PoolAddCommand.

1.20 ltfsdm pool remove

The ltfsdm pool remove command removes a cartridge from a tape storage pool.

`usage:`

`ltfsdm pool remove -h`

`ltfsdm pool remove -P <pool name> -t <tape id> [-t <tape id> [...]]`

parameters	description
-P <pool name>	pool name to which a cartridge should be removed
-t <tape id>	id of a cartridge to be removed

Example:

```
[root@visp ~]# ltfsdm pool remove -P 'large pool' -t DV1478L6
Tape DV1478L6 successfully removed from pool "large pool".
```

The corresponding class is PoolRemoveCommand.

Chapter 2

Design

Directories

In the following the LTFS Data Management design is described on a low level. It describes how the processing is done from the client side to start the LTFS Data Management server and to migrate and recall files.

LTFS Data Management is a client-server application where the server (sometimes also referred as backend) is started initially and eventually performs all the operations and the client - as an interface to LTFS Data Management - that initiates these operations. A user only has to work with the client part of the application. The code is structured within the following sub-directories:

path	description
src/common	code that is used within multiple parts (client, server, connector, common)
src/messages	Messaging
src/communication	code for the communication between: client $\longleftrightarrow$ server,server $\longleftrightarrow$ Fuse overlay file system (transparent recalls)
src/client	Client Code
src/connector	code for the connector interface, see Connector for more information
src/server	Server Code

The common code consists of the following:

path	description
src/common/Configuration.h	code to maintain the configuration information (storage pools, file systems)
src/common/Const.h	internal constants of the code consolidated here
src/common/errors.h	error values used within the code consolidated here
src/common/FileSystems.h	file system information retrieval and mount operations
src/common/LTFSDMException.h	the LTFS Data Management exception class
src/common/Message.h	the LTFS Data Management messaging facility
src/common/Trace.h	the LTFS Data Management tracing facility
src/common/util.h	utility functions

There are two files within the main directory that are used to generate c++ code:

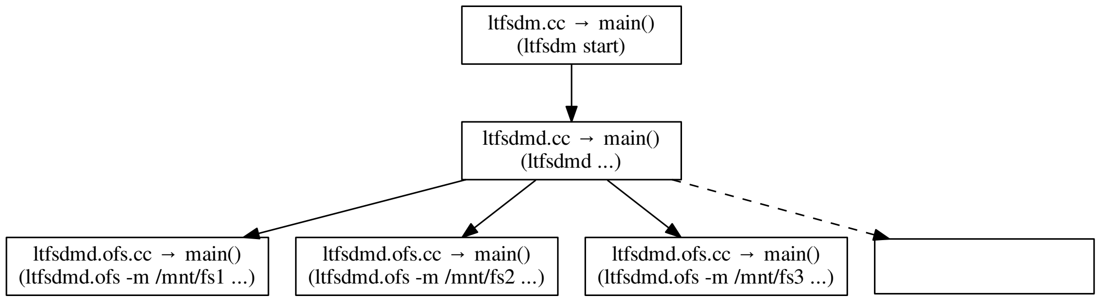
path	description
src/ltfsdm.proto	protocol buffers definition file for the communication
src/messages.cfg	the text based definition file for the messages

Processes

The following main() function entry points exist:

path	description
src/client/ltfsdm.cc	client entry point
src/server/ltfsdmd.cc	server entry point
src/connector/fuse/ltfsdmd.ofs.cc	Fuse overlay file system entry point
src/messages/msgcompiler.cc	message compiler entry point

There are four executables created that correspond to the list above. The message compiler only is used during the build. The other three executables are used during normal operation:



The number of processes of the Fuse overlay file systems corresponds to the number of file systems managed with LTFS Data Management. E.g. if only one file system is managed there will be only one Fuse overlay file system process.

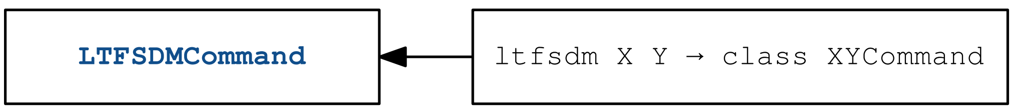
The following is an example that shows the processes that are running in the background if one file system (here **/mnt/lxfs**) is managed:

```
[root\@visp ~]# ps -p $(pidof ltfsdmd) $(pidof ltfsdmd.ofs)
 PID TTY      STAT   TIME COMMAND
 32246 ?        Ssl  674:51 /root/LTFSDM/bin/ltfsdmd
 32263 ?        Sl   918:32 /root/LTFSDM/bin/ltfsdmd.ofs -m /mnt/lxfs -f /dev/sdc1 -S 1510663933 -N 4975136
```

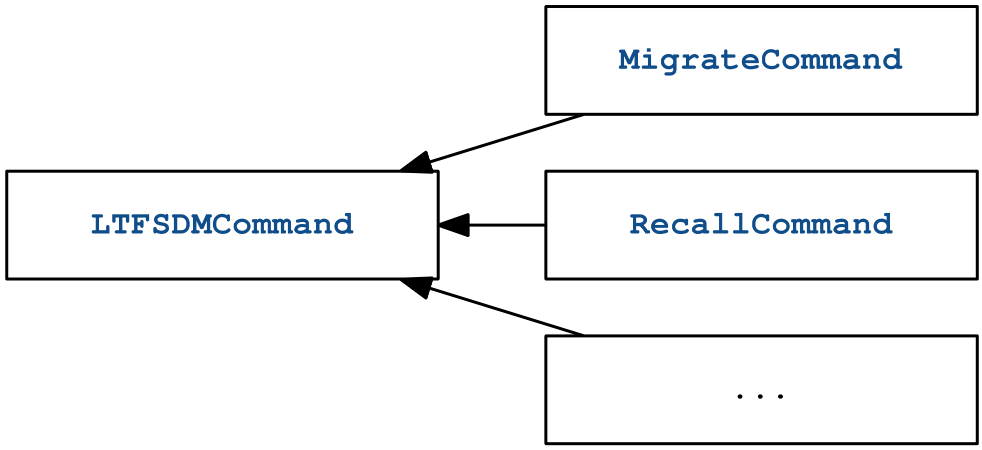
2.1 Client Code

Class Hierarchy

For each of the commands there exists a separate class that is derived from the LTFSDMCommand class. The class name of a command consists of the string "Command" that is appended to the command name.



E.g. for the ltfsdm migrate and for the ltfsdm recall commands there exist MigrateCommand and RecallCommand classes:



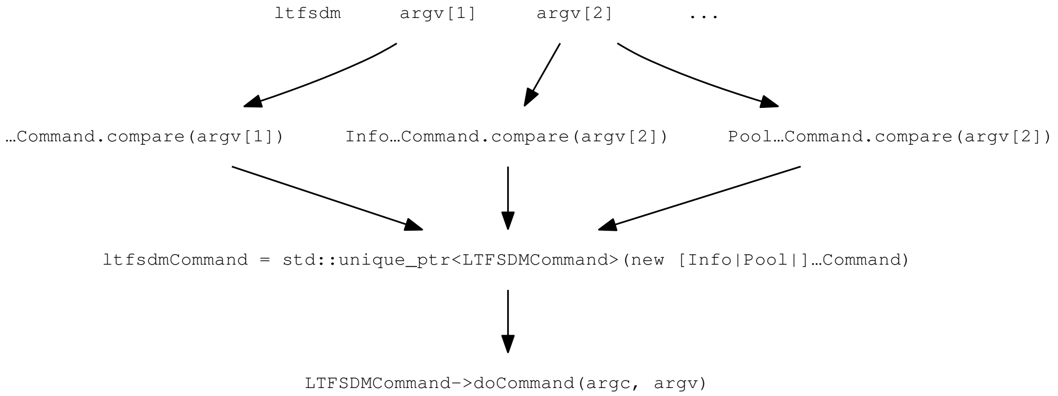
Any new command should follow this rule of creating a corresponding class name.

The actual processing of a command happens within virtual the LTFSDMCommand::doCommand method. Any command needs to implement this method even there is no need to talk to the backend.

Command evaluation

LTFS Data Management is started, stopped, and operated using the ltfsdm executable.

For all commands the first and for the info and pool commands also the second argument of the ltfsdm executable is evaluated.



Options

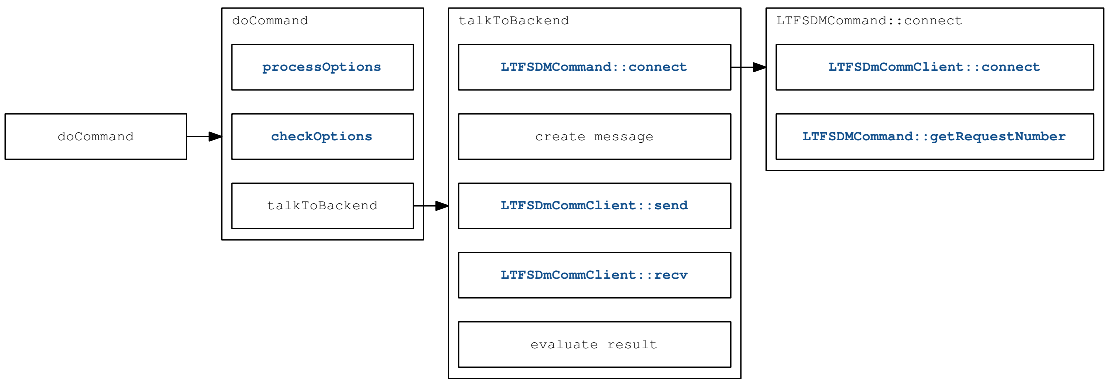
Some commands do an additional option processing. Each option has a particular meaning even it is used by different commands. The reason for doing so is to make it easier for users to switch between the commands. The option processing is performed within the single method LTFSDMCommand::processOptions. The following is a list of all options:

option	meaning
-h	show the usage
-p	perform a premigration instead to fully migrate (no stubbing)
-r	recall to resident state instead to premigrated state
-n <request number>	the request number
-f <file name>	the name of a file that contains a list of file names
-m <mount point>	the mount point of a file system to be managed
-P <pool list>	a list of up to three tape storage pools (separated by commas)
-t <tape id>	the id of a cartridge
-x	indicates a forced operation
-F	format a cartridge when added to a tape storage pool
-C	check a cartridge when added to a tape storage pool

The LTFSDMCommand::checkOptions method checks if the number of arguments is correct and the request number is not set.

Command processing

In general the command processing within the client code is performed in the following way:



Not all items that are listed in the following are performed for all commands. The ltfsdm info files command e.g does not communicate to the backend since it is not necessary to do so for the file state evaluation. The following list gives a description of code components of the LTFS Data Management client:

item	description	same implementation for all commands
doCommand	performs all required operations for a certain command	no

LTFSDMCommand::process-Options	option processing	yes
LTFSDMCommand::checkOptions	checks the number of arguments	yes
talkToBackend	performs the communication with the backend	no
LTFSDMCommand::connect	connects to the backend and retrieves the request number	yes
create message	assembles a Protocol Buffer message to send to the backend	no
LTFSDmCommClient::send	sends a Protocol Buffer message	yes
LTFSDmCommClient::recv	receives a Protocol Buffer message	yes
evaluate result	checks the information that has been sent from the backend	no
LTFSDmCommClient::connect	connects to the backend	yes
LTFSDMCommand::getRequest-Number	retrieves the request number	yes

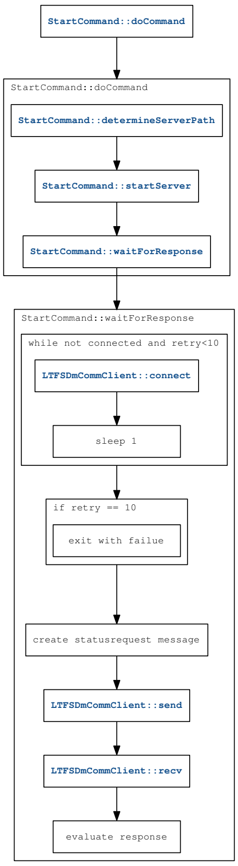
For the following four commands the processing is described in more detail:

- start
- stop
- migrate
- recall

2.1.1 start processing

The following is a summary of the start command processing:

```
StartCommand::doCommand
   StartCommand::determineServerPath
   StartCommand::startServer
   StartCommand::waitForResponse
       while not connected and retry<10
           LTFSDmCommClient::connect
           sleep 1
       if retry == 10
           exit with failue
       create statusrequest message
       LTFSDmCommClient::send
       LTFSDmCommClient::recv
       evaluate response
```



The start commands starts the LTFS Data Management server. To do so its path name needs to be detected. This is performed by the StartCommand::determineServerPath method. Since the client and the server path are the same it is just necessary to read the link to the executable of the current client process via procfs.

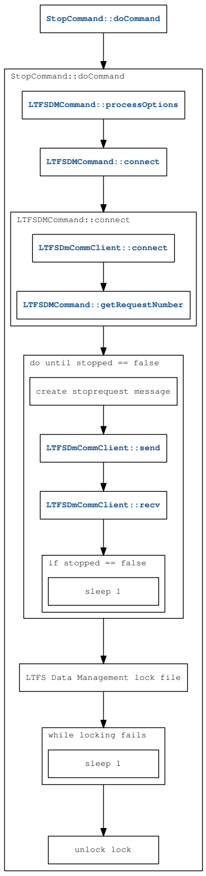
The backend is started within the StartCommand::startServer method. It is started via popen system call.

After the backend is started the status information is requested within the StartCommand::waitForResponse method. A connection is retried 10 times before giving up and reporting a failure.

2.1.2 stop processing

The following is a summary of the stop command processing:

```
StopCommand::doCommand
    LTFSDMCommand::processOptions
    LTFSDMCommand::connect
        LTFSDmCommClient::connect
        LTFSDMCommand::getRequestNumber
    do
        create stoprequest message
        LTFSDmCommClient::send
        LTFSDmCommClient::recv
        if stopped == false
            sleep 1
    until stopped == false
    LTFS Data Management lock
    while locking fails
        sleep 1
    unlock lock
```



When processing the stop command at first a stoprequest message is sent to the to the backend. If this fails it is repeated as long the backend does not respond with success message. Thereafter the server lock is tried to acquire to see that the server process is finally gone. The backend holds a lock all the time it is operating.

2.1.3 migrate processing

The following is a summary of the migrate command processing:

```
MigrateCommand::doCommand
    LTFSDMCommand::processOptions
    LTFSDMCommand::checkOptions
    if filename arguments
        create stream containing file names
    LTFSDMCommand::isValidRegularFile
    MigrateCommand::talkToBackend
        create migrequest message
        LTFSDmCommClient::send
        LTFSDmCommClient::recv
        evaluate response
        LTFSDMCommand::sendObjects
            while filenames to send
                create sendobjecs message
                LTFSDmCommClient::send
                LTFSDmCommClient::recv
                evaluate response
        LTFSDMCommand::queryResults
            do
                create reqstatusrequest message
                LTFSDmCommClient::send
                LTFSDmCommClient::recv
                print progress
            until not done
```

After the option processing is done it is evaluated in which way the file names are provided to the command. There are three different possibilities:

- the file names are provided as arguments to the command
- a file list is provided containing the file names
- a "-" is provided as file list name and the file names are provided by stdin

If a file list is provided it is checked if it is a valid regular file.

The further processing is performed by communicating with the backend. The three steps that are performed are the following:

- general migration information is sent to the backend
- the file names are send to the backend
- the progress or results are queried

2.1.4 recall processing

The following is a summary of the recall command processing:

```
RecallCommand::doCommand
    LTFSDMCommand::processOptions
    LTFSDMCommand::checkOptions
    if filename arguments
        create stream containing file names
    LTFSDMCommand::isValidRegularFile
    RecallCommand::talkToBackend
        create selrecrequest message
        LTFSDmCommClient::send
        LTFSDmCommClient::recv
        evaluate response
        LTFSDMCommand::sendObjects
            while filenames to send
                create sendobjecs message
                LTFSDmCommClient::send
                LTFSDmCommClient::recv
                evaluate response
        LTFSDMCommand::queryResults
            do
                create reqstatusrequest message
                LTFSDmCommClient::send
                LTFSDmCommClient::recv
                print progress
            until not done
```

After the option processing is done it is evaluated in which way the file names are provided to the command. There are three different possibilities:

- the file names are provided as arguments to the command
- a file list is provided containing the file names
- a "-" is provided as file list name and the file names are provided by stdin

If a file list is provided it is checked if it is a valid regular file.

The further processing is performed by communicating with the backend. The three steps that are performed are the following

- general selective recall information is sent to the backend
- the file names are send to the backend
- the progress or results are queried

2.2 Connector

Connector

LTFS Data Management provides three possibilities to move data between disk and tape:

operation	explanation
migration	moves data from disk to tape
selective recall	selectively moves data from tape to disk
transparent recall	transparently moved data from tape to disk

Selective recall moves data on request. I.e. a file is known as migrated and a user wants to recall it back to disk. Transparent recall does the same but on data access. I.e. a user does not need to be aware that data is on tape. If data is e.g. read the read operation is blocked for the time data is transferred back to disk. For selective recall applications need to be aware that a file is migrated and need to initiate the recall operation. For transparent recall applications do not need to be aware of the migration state. Read, write, and truncate system calls are blocked as long as it take to move data back to disk. In this case applications do not have to be rewritten to deal with migrated data but should be tolerant against delays since it can take time to perform the data transfer.

Three different technologies for data management

To implement an application that provides a transparent recall feature additional functionality provided by the operating system is necessary to intercept and block i/o system calls. Migration and selective recall do not have any additional requirements on the operating system and therefore are simpler and more flexible to implement. Three different technologies have been discussed or implemented for that purpose:

technology/API	description
DMAPI	Data Management API (`https://en.wikipedia.org/wiki/DMAPI`), only available for the XFS file system type
Fanotify	File system event notification system (`http://man7.org/linux/man-pages/man7/fanotify.7.html`), file system type independent, no mandatory locking
Fuse overlay file system	File system in user space (`https://github.com/libfuse/libfuse`), file system type independent, no direct file system access

The Fanotify API looked very promising since it is file system type independent and also provides direct file system access. This API seems to have two limitations that still persist:

- Only open (FAN_OPEN_PERM) and read (FAN_ACCESS_PERM) calls are able to intercept. It is possible to deal with that limitation by using the assumption that a file opened for write will be always written. If an application opens a file for writing it would be always recalled even there never followed a read or write call.
- It does not to seem possible to block file access if the data management application that implements the Fanotify API has not been started. Events only seem to be generated if the data management application is in operation. Especially this limitation does not allow to use this technology.

For DMAPI there has been implemented a reference implementation. One limitation is that it is only available for teh XFS file system type. Since only the SLES distributions fully provide this API it is not considered to completely support that.

Fuse is an API to implement a file system in user space. By using this third approach it is possible to write an overlay file system on top on the original file system that should be managed. Read, write, or truncate system call can be intercepted within the overlay file system to let data be transferred back frpm tape to disk. Other system calls are forwarded to the original file system. See Fuse connector for detailed information. Also this third possibility has disadvantages:

- The original file system never should be directly accessed. All access should happen through the Fuse overlay file system.
- Migrated files appear as empty file within the original file system. There is no practical way to make it impossible for a user to mount the original file system and therefore there is still a possibility to work with that empty files within the original file system.

- Currently only POSIX functionality is provided by the Fuse overlay file system. Any file system type specific functionality is not available.

The Fuse approach is the only that is completely developed for LTFS Data Management.

The Connector

The Connector is an API that should cover the very different technologies. Currently it is implemented for the Fuse overlay approach and the DMAPI reference implementation. If there will be a better or more suitable API

- additionally to the three approaches discussed here - for doing data management it should be possible to integrate that by using the connector.

The attributes

If a file is premigrated or migrated it is necessary persistently to store some information about the migration state and where to find the corresponding data on tape. It is possible to use a database for this purpose with the disadvantage to perform costly lookups. A more appropriate way is to store this information together with the files. Extended attributes provide a possibility to store these specifications outside the file data. For extended attributes there also exists a POSIX compliant interface for the file system types that are supported with LTFS Data Management.

The two attributes that exist are the following:

The migration target attribute

The migration target attribute the provides information where the data is on tape:

```
struct mig_target_attr_t
{
    unsigned long typeId;
    bool added;
    int copies;
    struct
    {
        char tapeId[Const::tapeIdLength + 1];
        long startBlock;
    } tapeInfo[Const::maxReplica];
};
```

The purpose of the components is the following:

component	purpose
typeId	To verify the that data type has not changed. After a code change it can happen that the attribute becomes invalid.
added	Workaround only used for the dmapi connector.
copies	The number of tapes the data has been copied.
tapeInfo	The tape ID and the starting block number of all tapes the data has been copied to.

The migration state attribute

The migration state attribute provides the information about the migration state of a file including some of the original stat data:

```
struct mig_state_attr_t
{
    unsigned long typeId;
    enum state_num
    {
```

```
            RESIDENT = 0,
            IN_MIGRATION = 1,
            PREMIGRATED = 2,
            STUBBING = 3,
            MIGRATED = 4,
            IN_RECALL = 5
        } state;
        unsigned long size;
        struct timespec atime;
        struct timespec mtime;
        struct timespec changed;
    };
```

The purpose of the components is the following:

component	purpose
typeId	To verify the that data type has not changed. After a code change it can happen that the attribute becomes invalid.
state	The migration state.
size	The file size before migration. To keep it and show it after (pre)migration.
atime	The atime before migration. To keep it and show it after (pre)migration.
mtime	The mtime before migration. To keep it and show it after (pre)migration.
changed	The time stamp when this attribute has been last updated.

This attribute is not used for the dmapi connector since the migration state is determined by so call dmapi managed regions. The size and the time stamps are not necessary to store in the dmapi case.

2.2.1 Fuse connector

Setup of LTFS Data Management for a file system.

Fuse Connector

The Fuse overlay file system

The Fuse Connector provides an overlay file system on top of an original file system that is managed by LTFS Data Management. File system access must only be performed by the Fuse overlay file system.

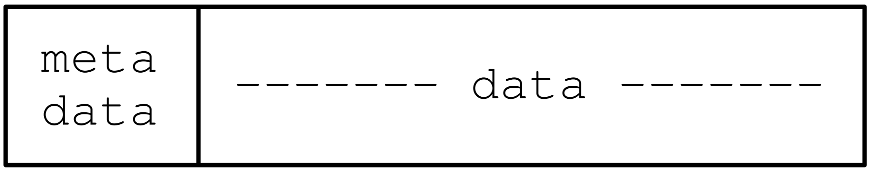
The Fuse overlay file system has the following two basic tasks:

- It provides some kind of control over the original file system. Most of the file system accesses are bypassed to the original. Read, write, and truncate system calls are suspended on migrated files to copy data from tape to disk.
- If a file is migrated, within the original file system this file is truncated to zero size. The Fuse overlay file system still shows this file in its original size. This is necessary for applications to perform i/o operations in the same way as data is locally available (resident or premigrated state). Some LTFS Data Management attributes are hidden because these are only internally used. The access and modification time stamps should be kept.

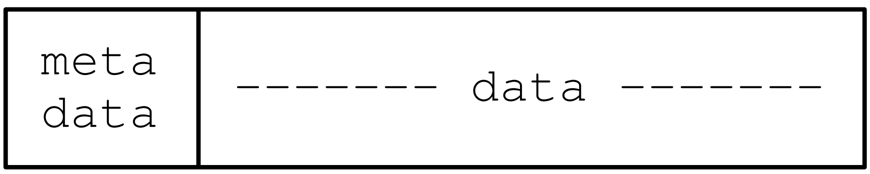
The following table shows how how a file appears in different migration states (resident, premigrated, migrated) within the two file systems. The Fuse overlay file system is virtual in a sense that there is no storage directly managed and all data and meta data are stored "somewhere else". Nevertheless it is possible to access data and meta data from the original file system.

In the following the it is shown how files appear (since the Fuse overlay file system only is virtual) for the different migration stages:

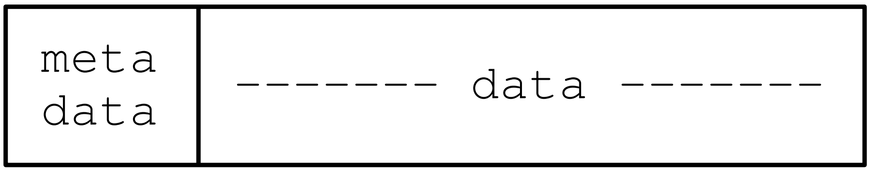
- resident/premigrated
  - Fuse overlay file system



  - original file system



- migrated
  - Fuse overlay file system



– original file system

Within the Fuse overlay file system the stat information of a migrated file looks as follows:

```
  File: ‘file.1’
  Size: 6335488     Blocks: 8          IO Block: 4096   regular file
Device: 42h/66d Inode: 3312825     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2018-01-12 14:57:16.780451059 +0100
Modify: 2018-01-12 14:57:06.096451529 +0100
Change: 2018-01-12 15:24:22.584379530 +0100
 Birth: -
```

For the file the following is the stat information within the original file system:

```
  File: 'file.1'
  Size: 0           Blocks: 8          IO Block: 4096   regular empty file
Device: 831h/2097d  Inode: 3312825     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2018-01-12 15:23:40.800381368 +0100
Modify: 2018-01-12 15:24:22.584379530 +0100
Change: 2018-01-12 15:24:22.584379530 +0100
 Birth: -
```

For each file system that is managed a Fuse overlay file system is created. Each of these file systems is operating within a different process. If there are three file systems (/dev/loop0, /dev/loop1, and /dev/loop2) which are managed with LTFS Data Management it looks like the following:

- mount information before managing:

```
vex:~ # cat /proc/mounts |grep loop
/dev/loop0 /mnt/loop0 ext4 rw,relatime,data=ordered 0 0
/dev/loop1 /mnt/loop1 ext4 rw,relatime,data=ordered 0 0
/dev/loop2 /mnt/loop2 ext4 rw,relatime,data=ordered 0 0
```

- now managing with LTFS Data Management:

```
vex:~ # ltfsdm add /mnt/loop0
vex:~ # ltfsdm add /mnt/loop1
vex:~ # ltfsdm add /mnt/loop2
```

- process and mount information thereafter:

```
vex:~ # cat /proc/mounts |grep loop
LTFSDM:/dev/loop0 /mnt/loop0 fuse rw,nosuid,nodev,relatime,user_id=0,group_id=0,default_permissions,allow
LTFSDM:/dev/loop1 /mnt/loop1 fuse rw,nosuid,nodev,relatime,user_id=0,group_id=0,default_permissions,allow
LTFSDM:/dev/loop2 /mnt/loop2 fuse rw,nosuid,nodev,relatime,user_id=0,group_id=0,default_permissions,allow

vex:~ # ps -fp $(pidof ltfsdmd) $(pidof ltfsdmd.ofs)
UID          PID    PPID  C STIME TTY      STAT   TIME CMD
root     3987134       1  0 14:13 ?        Ssl    0:00 /root/LTFS-Data-Management/bin/ltfsdmd
root     3987592 3987591  0 14:18 ?        Sl     0:00 /root/LTFS-Data-Management/bin/ltfsdmd.ofs -m /mnt
root     3987610 3987609  0 14:18 ?        Sl     0:00 /root/LTFS-Data-Management/bin/ltfsdmd.ofs -m /mnt
root     3987625 3987624  0 14:18 ?        Sl     0:00 /root/LTFS-Data-Management/bin/ltfsdmd.ofs -m /mnt
```

Each of the ltfsdmd.ofs processes represent a different Fuse overlay file system.

Startup

File systems are managed by LTFS Data Management in two cases:

- If a file system is added using the ltfsdm add command.
- If a file system has been added previously: during the startup phase of LTFS Data management done automatically.

The following two methods are involved when managing a file system:

```
FsObj::manageFs
    -> FuseFS::init
```

The FuseFS::init method works as documented:

> Setup of LTFS Data Management for a file system.
> The setup happens according the following steps:

1. Unmount the original file system.
2. Perform a so called fake mount (see `mount -f` command) by only specifying the mount point to see if it can be mounted automatically. Any file system managed by LTFS Data Management should not be mounted beside the LTFS Data Management service (especially not automatically during system startup).
3. Start of the Fuse overlay file system. The Fuse overlay file system is mounted at the original mount point.
4. Wait for the Fuse overlay file system to be in operation and open a file descriptor for the ioctl communication.
5. Mount the original file system within the cache mount point Const::LTFSDM_CACHE_MP.
6. Open the file descriptor FuseFS::rootFd on its root: i.e. <original mount point>/Const::LTFSDM_CACHE_MP.
7. Via ioctl tell the Fuse process to continue. It was blocked before since it can only be fully operational if access to the orginal file system is available.
8. Perform a detached unmount of the original file system.

After the last step there is no general access possible to the original file system. However the LTFS Data Management service is able to access the file system via FuseFS::rootFd file descriptor. The procedure listed here i.a. is to guarantee that the original file system is not in use anymore when doing the management. If one of the steps fails the original file system will not be managed.

Parameters

starttime	start time of the LTFS Data Management service

Code overview

code part	description
class Connector	Class for external usage to manage recall events.
class FsObj	Class for external usage to represent a file system object.
file ltfsdmd.ofs.cc	File that provides the main entry point for the Fuse overlay file system.
class FuseFS	Fuse overlay file system implementation.
namespace FuseConnector	Global variables available within the Fuse Connector.

Connector

This class is providing the recall event system. Most prominent methods are

- Connector::getEvents to get a recall event
- Connector::respondRecallEvent to respond a recall event

Further methods initialize and stop the recall event system.

FsObj

The FsObj class provides an interface to manage and work with file system objects. This includes:

- to manage a file system

  FsObj::isFsManaged

  FsObj::manageFs

- to provide stat information of a file system object

  FsObj::stat

- to provide file file uid, see fuid_t

  FsObj::getfuid

- to provide the tape id of migrated and premigrated files

  FsObj::getTapeId

- to lock file system objects

  FsObj::lock

  FsObj::try_lock

  FsObj::unlock

- to read from and to write to files

  FsObj::read

  FsObj::write

- to work with file attributes

  FsObj::addAttribute

  FsObj::remAttribute

  FsObj::getAttribute

- to perform file state changes or to prepare them

  FsObj::preparePremigration

  FsObj::finishPremigration

  FsObj::prepareRecall

  FsObj::finishRecall

  FsObj::prepareStubbing

  FsObj::stub

- to get the migration state of a file system object

  FsObj::getMigState

ltfsdmd.ofs.cc

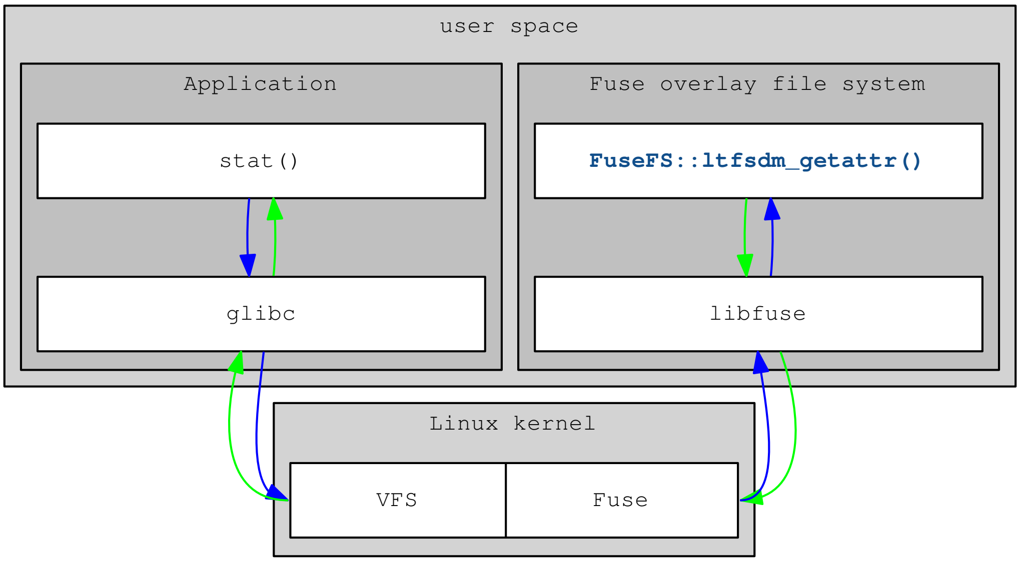
For every file system that is managed with LTFS Data Management an Fuse overlay file system is created as a separate process. This file provides main entry point that also acts as an interface to the LTFS Data Management backend. Within the main entry point the following things are done:

- An option processing is performed to receive information that is provided by the backend.
- Messaging and Tracing is setup.
- The 128bit file system uuid is determined.
- The Fuse options are set.
- The Fuse shared information is set.

FuseFS

Call back functions have to be implemented to create a Fuse file system. These call back functions are similar and correspond to POSIX calls.

In the following graph it is shown how a stat call within a user space application is processed: the stat call is performed in the application, the request is transferred via glibc system library, the kernel vfs layer, the kernel fuse layer, the libfuse system library to the FuseFS::ltfsdm_getattr method which is the corresponding call back implemented for the Fuse overlay file system (blue color). The the response is shown in the reverse direction (green color):



The following call back functions have been implemented:

```
static int ltfsdm_getattr(const char *path, struct stat *statbuf);
static int ltfsdm_access(const char *path, int mask);
static int ltfsdm_readlink(const char *path, char *buffer, size_t size);
static int ltfsdm_opendir(const char *path, struct fuse_file_info *finfo);
static int ltfsdm_readdir(const char *path, void *buf,
        fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *finfo);
static int ltfsdm_releasedir(const char *path,
        struct fuse_file_info *finfo);
static int ltfsdm_mknod(const char *path, mode_t mode, dev_t rdev);
static int ltfsdm_mkdir(const char *path, mode_t mode);
static int ltfsdm_unlink(const char *path);
static int ltfsdm_rmdir(const char *path);
static int ltfsdm_symlink(const char *target, const char *linkpath);
static int ltfsdm_rename(const char *oldpath, const char *newpath);
static int ltfsdm_link(const char *oldpath, const char *newpath);
static int ltfsdm_chmod(const char *path, mode_t mode);
static int ltfsdm_chown(const char *path, uid_t uid, gid_t gid);
static int ltfsdm_truncate(const char *path, off_t size);
static int ltfsdm_utimens(const char *path, const struct timespec times[2]);
static int ltfsdm_open(const char *path, struct fuse_file_info *finfo);
static int ltfsdm_ftruncate(const char *path, off_t size,
        struct fuse_file_info *finfo);
// read not used
static int ltfsdm_read(const char *path, char *buffer, size_t size,
        off_t offset, struct fuse_file_info *finfo);
static int ltfsdm_read_buf(const char *path, struct fuse_bufvec **bufferp,
        size_t size, off_t offset, struct fuse_file_info *finfo);
// write not used
static int ltfsdm_write(const char *path, const char *buf, size_t size,
        off_t offset, struct fuse_file_info *finfo);
static int ltfsdm_write_buf(const char *path, struct fuse_bufvec *buf,
```

```
        off_t offset, struct fuse_file_info *finfo);
static int ltfsdm_statfs(const char *path, struct statvfs *stbuf);
static int ltfsdm_release(const char *path, struct fuse_file_info *finfo);
static int ltfsdm_flush(const char *path, struct fuse_file_info *finfo);
static int ltfsdm_fsync(const char *path, int isdatasync,
        struct fuse_file_info *finfo);
static int ltfsdm_fallocate(const char *path, int mode, off_t offset,
        off_t length, struct fuse_file_info *finfo);
static int ltfsdm_setxattr(const char *path, const char *name,
        const char *value, size_t size, int flags);
static int ltfsdm_getxattr(const char *path, const char *name, char *value,
        size_t size);
static int ltfsdm_listxattr(const char *path, char *list, size_t size);
static int ltfsdm_removexattr(const char *path, const char *name);
static int ltfsdm_ioctl(const char *path, int cmd, void *arg,
        struct fuse_file_info *fi, unsigned int flags, void *data);
static void *ltfsdm_init(struct fuse_conn_info *conn);
static void ltfsdm_destroy(void *ptr);
```

Communication between LTFS Data Management backend and the Fuse overlay file system

Communication between the LTFS Data Management backend and the Fuse overlay file system is performed in two different ways:

- using the file system API
- using a local socket

The later one only is used to communicate transparent recall events and the correspondig responses. For the socket communication Google Protocol Buffers are used in the same way as the client talks to the backend.

File system API communication

In general the POSIX file operation calls are forwarded to the underlying original file system. E.g. if an attribute of a file within the Fuse overlay file system is changed, the attribute change actually is done at the corresponding file within the underlying original file system. However it is possible to transfer information that is not related to specific files, file content, or file attributes within the original file system that is managed. E.g. the overlay file system provides information of a virtual file that does not exist in the original file system. For normal files read or write calls are transferred to the original file within the original file system. The virtual file just is used for communication. The same can be done for virtual attributes that does not exist on files within the original file system. Using virtual attributes is a common way of communicating.

One virtual attribute is also used by LTFS Data Management:

- Const::LTFSDM_EA_FSINFO

This virtual attribute provides information part of FuseFS::FuseHandle.

A more appropriate way to communicate with the file system is the implementation of the ioctl call. This call is generally used to communicate with the kernel - also outside Fuse. The ioctl calls that are implemented here are the following:

```
enum
{
    LTFSDM_FINFO = _IOR('l', 0, FuseFS::FuseHandle),    // provides the mount point,
   the relative path
                                                        // of a file, and the lock path
    LTFSDM_PREMOUNT = _IO('l', 1),                      // synchronization when adding a file system
    LTFSDM_POSTMOUNT = _IO('l', 2),                     // set the root file descriptor when adding
                                                        // management to a file system
    LTFSDM_STOP = _IO('l', 3),                          // not used
    LTFSDM_LOCK = _IOWR('l', 4, FuseFS::FuseHandle),    // not used
    LTFSDM_TRYLOCK = _IOWR('l', 5, FuseFS::FuseHandle), // not used
    LTFSDM_UNLOCK = _IOW('l', 6, FuseFS::FuseHandle),   // not used
};
```

Local socket communication

For transparent recalls the Fuse overlay file system needs to inform the backend about recall events. The backend needs to provide the response if it finished successfully or not. For that purpose local socket communication is used since it already has been used for the communication between clients and the backend.

The following Protocol Buffers messages are used for that purpose:

- LTFSDmProtocol::LTFSDmTransRecRequest
- LTFSDmProtocol::LTFSDmTransRecResp

Mandatory file locking

LTFS Data Management requires mandatory file locking. If a file data is just being transferred to tape applications should not be able to write to that file.

To provides this functionality one possibility would be to to use the standard advisory locking and lock certain portions of that file to indicate that it is locked mandatorily. A problem with this approach is that such a lock can interfere with a similar lock performed by an application on the same file.

The approach in LTFS Data Management is to use a virtual files for the purpose of locking. These files are not accessible from user space applications and have the following format:

Const::LTFSDM_LOCK_DIR/<inode number>.[m|f]

where 'm' and 'f' indicate different levels of locking.

2.3 Server Code

The backend

The ltfsdmd command

The backend usually is started by the ltfsdm start command. However it is possible to start the backend directly by using the

```
ltfsdmd [-f] [-m] [-d <debug level>]
```

command.

The option arguments have the following meaning:

option	meaning
-f	Start the backend in foreground. Messages will be printed out to stdout.
-m	Store the SQLite database in memory. By default it is stored in "/var/run" which usually is memory mapped.
-d	Use a different trace level. See tracing for details of trace levels.

Server components

Main task of the backend is the processing of migration and recall requests and to perform the resource management of cartridges and tape drives. To optimally use the resources for migration and recall requests, requests need to be queued and scheduled when a required resource is ready to be used. Information needs to be stored temporarily for the queuing. Two SQLite tables are used for that purpose. A description of the two tables can be found at SQLite tables.

The following is a high level sequence how such requests get processed:

- a request and corresponding jobs are added to the internal queues
- the scheduler is looking for a free drive and tape resource
- if there exists a corresponding free resource the request gets scheduled
- the jobs can be processed on that tape

The term request and job here is used in the following way:

- A request is an operation started by a single command. If a user likes to migrate a lot of files by a single command there will be a single migration request. The scheduler only is considering requests to be scheduled.
- A job is a single operation that is performed on a cartridge. For migration: a job is related to a file to be migrated. If there is a migration request to migrate 1000 files to a single tape 1000 jobs will be created.

Some requests as well as some jobs can and should happen concurrently. E.g. there are two drives available and some files get recalled form a single tape another tape can be used in parallel e.g. for migration. If a request is being processed on a tape it should be possible to add further requests and jobs to the internal queues in parallel.

Threads

For concurrency threads are created. Some of them run all the time - some others are started on request. For the latter case thread pools are available where threads automatically terminate if not longer used.

In the following example threads are listed after starting the backend:

```
  Id   Target Id         Frame
  12   Thread 0x7f8f62c7a700 (LWP 16635) "ltfsdmd" 0x00007f8f65889a9b in recv () from /lib64/libpthread.so.0
  11   Thread 0x7f8f62479700 (LWP 16640) "Scheduler" 0x00007f8f65886945 in pthread_cond_wait@@GLIBC_2.3.2 () f
  10   Thread 0x7f8f61c78700 (LWP 16641) "w:Scheduler" 0x00007f8f65883f57 in pthread_join () from /lib64/libpt
  9    Thread 0x7f8f61477700 (LWP 16642) "SigHandler" 0x00007f8f6588a371 in sigwait () from /lib64/libpthread.
  8    Thread 0x7f8f60c76700 (LWP 16643) "w:SigHandler" 0x00007f8f65883f57 in pthread_join () from /lib64/libp
  7    Thread 0x7f8f4bfff700 (LWP 16644) "Receiver" 0x00007f8f6588998d in accept () from /lib64/libpthread.so.
  6    Thread 0x7f8f4b7fe700 (LWP 16645) "w:Receiver" 0x00007f8f65883f57 in pthread_join () from /lib64/libpth
  5    Thread 0x7f8f4affd700 (LWP 16646) "RecallD" 0x00007f8f6588998d in accept () from /lib64/libpthread.so.0
  4    Thread 0x7f8f4a7fc700 (LWP 16647) "w:RecallD" 0x00007f8f65883f57 in pthread_join () from /lib64/libpthr
  3    Thread 0x7f8f49ffb700 (LWP 16648) "ltfsdmd.ofs" 0x00007f8f640f07fd in read () from /lib64/libc.so.6
  2    Thread 0x7f8f497fa700 (LWP 16662) "ltfsdmd.ofs" 0x00007f8f640f07fd in read () from /lib64/libc.so.6
* 1    Thread 0x7f8f660e08c0 (LWP 16633) "ltfsdmd" 0x00007f8f65883f57 in pthread_join () from /lib64/libpthrea
```

These threads have the following purpose

Id	function being executed	description
12	-	communication with LTFS LE
11	Scheduler::run	schedules requests based on free resources
10	SubServer::waitThread	waits for Scheduler thread termination
9	Server::signalHandler	cares about signals
8	SubServer::waitThread	waits for signal handler thread termination

7	Receiver::run	listens for client messages
6	SubServer::waitThread	waits for Receiver thread termination
5	TransRecall::run	listens for transparent recall requests
4	SubServer::waitThread	waits for TransRecall thread termination
3	FuseFS::execute	started the Fuse connector process for a single file system
2	FuseFS::execute	started the Fuse connector process for another file system
1	ltfsdmd.cc:main()	main thread

In this example two file systems are managed by LTFS Data Management. Therefore two Fuse threads exist (for starting the Fuse overlay file system processes). Thread pools are not visible after initial start since threads within a thread pool are created on request.

Furthermore the scheduler is creating additional threads for each request being scheduled:

function	description
Migration::execRequest	schedules a migration request
SelRecall::execRequest	schedules a selective recall request
TransRecall::execRequest	schedules a transparent recall request

For each of these threads there will be an additional waiter thread.

The following thread pools are available:

operation	object	function being executed	description
message parsing	Receiver::run -> wqm	MessageParser::run	After the Receiver gets a new message this message is further processed by a new thread from this thread pool.
premigration	LTFSDMDrive::wqp	Migration::preMigrate	For premigration there is one thread pool per drive since only a single request can be executed on a certain drive at a time.
stubbing	Server::wqs	Migration::stub	There exist one thread pool for all stubbing operations (even from different requests).
transparent recall	TransRecall::run -> wqr	TransRecall::addJob	For adding transparent recall requests and jobs.

Overall this leads to the following picture:

```
main()
ltfsdmd.run
    communication with LTFS LE (1 thread)
    LTFSDMDrive::wqp(number of thread pools equal number of drives)
    FuseFS::execute (threads equal number of files systems)
    Server::wqs (1 thread pool)
    Scheduler::run (1 thread)
        Migration::execRequest (number of thread less or equal number of drives)
        SelRecall::execRequest (number of thread less or equal number of drives)
        TransRecall::execRequest (number of thread less or equal number of drives)
    Server::signalHandler (1 thread)
    Receiver::run (1 thread)
        Receiver::run -> wqm (1 thread pool)
    TransRecall::run (1 thread)
```

```
        TransRecall::run -> wqr (1 thread pool)
```

To create threads there are two facilities created:

- Threads normally created by the SubServer class. After the thread function finishes the thread is terminated. For each thread an additional waiter thread is created. See SubServer.
- For a better performance and less overhead a ThreadPool class exist. Threads will stay for another 10 seconds before termination after the thread function terminates. Within these 10 seconds it is possible to reuse them. See ThreadPool.

Backend Processing

The following sections provides an overview about the backend processing of client requests:

- Receiver and client message parsing
- Scheduler
- Migration
- Selective Recall

Furthermore for transparent recalling the following section is available:

- Transparent Recall

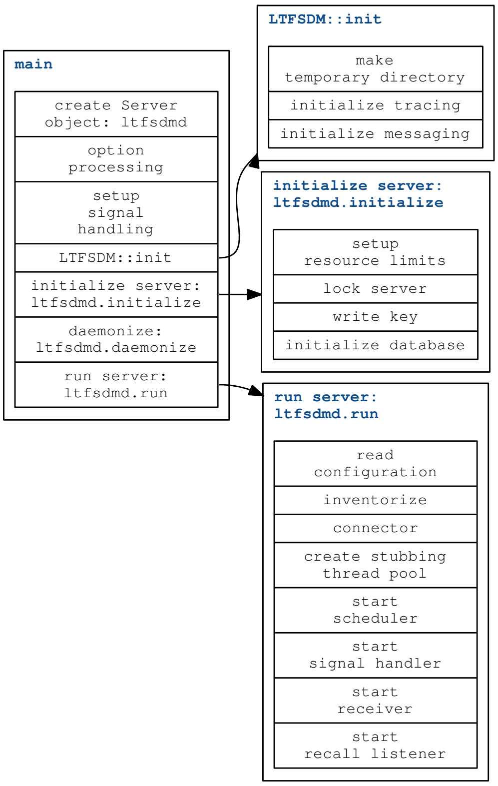
The startup sequence

During the startup initialization is done and threads are started for further processing.

The configuration is read and the information about drives and cartridges are received from LTFS LE. The configuration provides information about the managed file systems and the tape storage pools. When the connector object is created these file systems will be managed. For the Fuse connector an overlay file system will be created for each managed file system. The creation of the drive and cartridge inventory in the following is called inventorize. During this operation premigration thread pools are created: one pool for each drive. A thread pool for the stubbing operation is setup. Thereafter the threads for scheduling, signal handling, the receiver, and the listener for the transparent recall requests are started.

The following gives an overview:

```
main
    create Server object: ltfsdmd
    option processing
    setup signal handling
    LTFSDM::init
        make temporary directory
        initialize tracing
        initialize messaging
    initialize server: ltfsdmd.initialize
        setup system limits
        lock server
        write key file
        initialize database
    ltfsdmd.daemonize
    ltfsdmd.run
        read configuration
        inventorize
        connector
        create stubbing thread pool
        start scheduler
        start signal handler
        start receiver
        start recall listener
```



For each of the these items in the following there is a more detailed desciption. Most corresponding code is part of the ltfsdmd.cc and Server.cc files.

<table>
<tr><th>item</th><th>description</th></tr>
<tr><td>main</td><td>main entry point</td></tr>
<tr><td>create Server object: ltfsdmd</td><td>see:

```
Server ltfsdmd;
```

</td></tr>
<tr><td>option processing</td><td>process options for the ltfsdmd command:

```
while ((opt = getopt(argc, argv, "fmd:")) != -1) {
    switch (opt) {
        case 'f':
            detach = false;
            break;
        case 'm':
            dbUseMemory = true;
            break;
        case 'd':
            try {
                tl = (Trace::traceLevel) std::stoi(optarg);
            } catch (const std::exception& e) {
                TRACE(Trace::error, e.what());
                tl = Trace::error;
            }
            break;
        default:
            std::cerr << ltfsdm_messages[LTFSDMC0013E] << std::end
            err = static_cast<int>(Error::GENERAL_ERROR);
            goto end;
    }
}
```

</td></tr>
<tr><td>setup signal handling</td><td>setup the signals for signal handling:

```
sigemptyset(&set);
sigaddset(&set, SIGQUIT);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGTERM);
sigaddset(&set, SIGPIPE);
sigaddset(&set, SIGUSR1);
pthread_sigmask(SIG_BLOCK, &set, NULL);
```

</td></tr>
<tr><td>LTFSDM::init</td><td>initialize common items that are not server specific:

```
void LTFSDM::init(std::string ident)

{
    mkTmpDir();
    messageObject.init(ident);
    traceObject.init(ident);
}
```

</td></tr>
<tr><td>LTFSDM::init -> make temporary directory</td><td>create temporary directory /var/run/ltfsdm, see: mkTmpDir()</td></tr>
<tr><td>LTFSDM::init -> initialize tracing</td><td>see: Message::init</td></tr>
<tr><td>LTFSDM::init -> initialize messaging</td><td>see: Trace::init</td></tr>
<tr><td>initialize server: ltfsdmd.initialize</td><td>see: Server::initialize</td></tr>
<tr><td>initialize server: ltfsdmd.initialize -> setup system limits</td><td>set the application specific resource limits:

```
if (setrlimit(RLIMIT_NOFILE, &Const::NOFILE_LIMIT) == -1) {
    MSG(LTFSDMS0046E);
    THROW(Error::GENERAL_ERROR, errno);
}

if (setrlimit(RLIMIT_NPROC, &Const::NPROC_LIMIT) == -1) {
    MSG(LTFSDMS0046E);
    THROW(Error::GENERAL_ERROR, errno);
}
```

</td></tr>
</table>

initialize server: ltfsdmd.initialize -> lock server	see: Server::lockServer()
initialize server: ltfsdmd.initialize -> write key file	see: Server::writeKey()

initialize server: ltfsdmd.initialize -> initialize database	initialize the internal SQLite database:
	`try {`
	`DB.cleanup();`
	`DB.open(dbUseMemory);`
	`DB.createTables();`
	`} catch (const std::exception& e) {`
	`TRACE(Trace::error, e.what());`
	`MSG(LTFSDMS0014E);`
	`THROW(Error::GENERAL_ERROR);`
	`}`

daemonize: ltfsdmd.daemonize	detaching the server process, see: Server::daemonize
run server: ltfsdmd.run	performing the remaining initialization and starting the main threads, see Server::init

run server: ltfsdmd.run -> read configuration	read the configuration file:
	`try {`
	`Server::conf.read();`
	`} catch (const std::exception& e) {`
	`MSG(LTFSDMX0038E);`
	`goto end;`
	`}`

run server: ltfsdmd.run -> inventorize	import information from LTFS about cartridges and drives: `inventory = new LTFSDMInventory();`

run server: ltfsdmd.run -> connector	create a connector object:
	`connector = std::shared_ptr<Connector>(`
	`new Connector(true, &Server::conf));`

run server: ltfsdmd.run -> create stubbing thread pool	create a thread pool for the stubbing operations:
	`Server::wqs = new ThreadPool<Migration::mig_info_t,`
	`std::shared_ptr<std::list<unsigned long>>, FsObj::file_sta`
	`&Migration::changeFileState,`
	`Const::MAX_STUBBING_THREADS,`
	`"stub1-wq");`

run server: ltfsdmd.run -> start scheduler	see: Scheduler::run
run server: ltfsdmd.run -> start signal handler	see: Server::signalHandler
run server: ltfsdmd.run -> start receiver	see: Receiver::run
run server: ltfsdmd.run -> start recall listener	thread listening for transparent recall requests, see TransRecall::run

2.3.1 SQLite tables

SQLite tables

Since client requests like migration or selective recall cannot always being processed immediately information about these requests needs to be stored temporarily. A request only can be executed if there is a free drive and tape resource available. A request is related to a single command initiated by a client. A client can e.g.:

- call the recall command which leads to one recall request.
- call the migration command which leads to up to three migration requests depending on the number of storage pools specified.

All requests are stored in the SQLite table REQUEST_QUEUE.

For each request one or more files can be processed. For each file one jobs is created (for migration up to three jobs depending on the number of tape storage pools being specified). Jobs are stored within the SQlite table JOB_QUEUE.

JOB_QUEUE

column	data type	details
OPERATION	INT	operation: see DataBase::operation
FILE_NAME	CHAR(4096)	file name
REQ_NUM	INT	for each new request incremented by one
TARGET_STATE	INT	target state for migration and recall: FsObj::state
REPL_NUM	INT	0, for migration 0,1,2 depending of the number of tape storage pools
TAPE_POOL	VARCHAR	name of the tape storage pool
FILE_SIZE	BIGINT	file size
FS_ID_H	BIGINT	higher 64 bit part of the 128 bit file system id
FS_ID_L	BIGINT	lower 64 bit part of the 128 bit file system id
I_GEN	INT	inode generation number
I_NUM	BIGINT	inode number
MTIME_SEC	BIGINT	mtime: seconds
MTIME_NSEC	BIGINT	mtime: nano seconds
LAST_UPD	INT	last update of this record (need to check if really used)
TAPE_ID	CHAR(9)	id of the cartridge that is being used
FILE_STATE	INT	file state: see FsObj::file_state
START_BLOCK	INT	starting block of the data on tape of a (pre)migrated file
CONN_INFO	BIGINT	address of connector specific information

REQUEST_QUEUE

column	data type	details
OPERATION	INT	operation: see DataBase::operation
REQ_NUM	INT	for each new request incremented by one
TARGET_STATE	INT	target state for migration and recall: FsObj::state
NUM_REPL		number of replicas, only used for migration
REPL_NUM	INT	0, for migration 0,1,2 depending of the number of tape storage pools
TAPE_POOL	VARCHAR	name of the tape storage pool
TAPE_ID	CHAR(9)	id of the cartridge that is being used

DRIVE_ID	VARCHAR	id of the drive that is being used mount or unmount requests
TIME_ADDED	INT	time the request has been added (need to check if really used)
STATE	INT	request state, see DataBase::req_state

2.3.2 SubServer

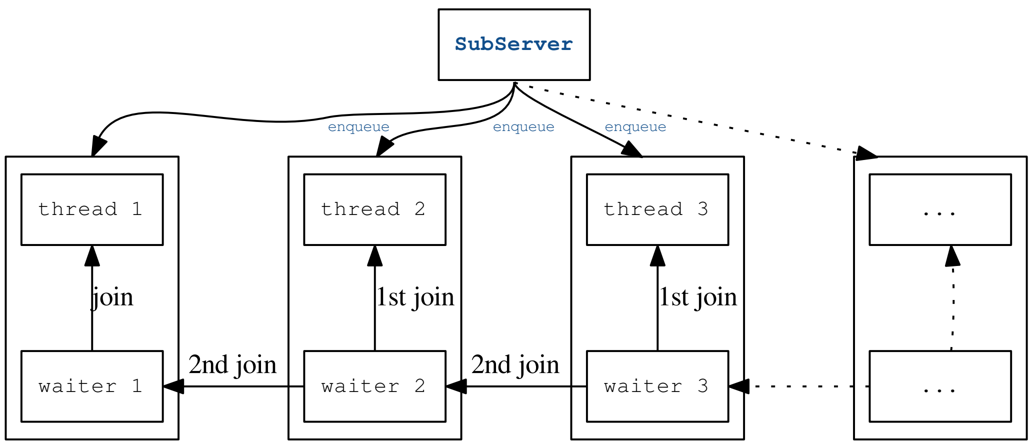
The SubServer class is designated as a facility to start threads and wait for their completion. It has the following capabilities:

- It is possible to set a maximum number of threads. Further thread creation is blocked if this limit is hit.
- A name can be specified for each thread to be created by calling the SubServer::enqueue method.
- The method SubServer::waitAllRemaining blocks until all threads are finished.

For each thread to be executed a waiter thread is created. This waiter joins joins the following threads:

- the thread that has been newly started
- if it is not the first thread it waits for the previous waiter to complete.

This additional waiter was necessary because of this upper limit of number of threads. A simpler implementation would be to have a vector of threads and join all of them at the when waiting for completion. In this case there would be no control about when a thread finished his work.



2.3.3 ThreadPool

The ThreadPool class is designated as a facility to start threads and wait for their completion. It has the following capabilities and limitations:

- Only a single function can be specified to be executed by the threads, parameters can be different.
- A single name can be specified for the threads.

- A thread can be reused up to 10 seconds of inactivity. Thereafter it will terminate.

For the constructor of the class the following parameters need to be specified:

- the function to be executed
- the maximum number of threads
- the name of the threads

A new thread can be enqueued with the ThreadPool::enqueue method. Only the function (that has been specified with the constructor) parameters and if necessary (if not Const::UNSET should be specified) a request number as a first parameter. For ThreadPool::waitCompletion method a request number can be specified to wait only for for specific request to finish (if not Const::UNSET should be specified). This only is the case for those ThreadPools that perform tasks for different request at the same time.

The ThreadPool is used by doing the following three steps:

- Setup the ThreadPool within its constructor with a name and a function to be executed.
- Execute this function in a thread by the ThreadPool::enqueue method and specifying a request number (use Const::UNSET if not required) and function parameters.
- Wait for all threads to complete by using the ThreadPool::waitCompletion method.

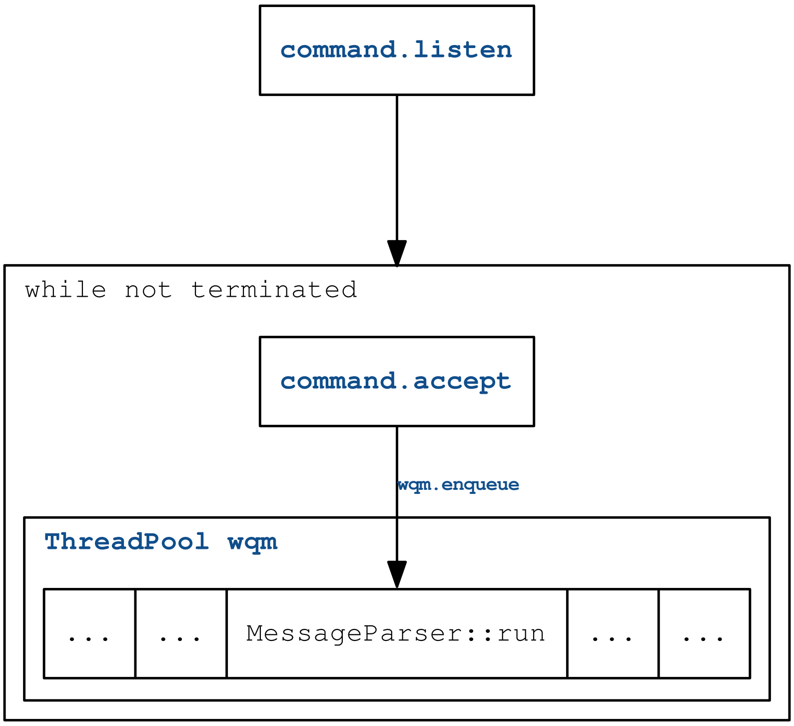
2.3.4 Receiver and client message parsing

When a client send a message to the backend there are two components that are processing such a message:

- The Receiver listens on a socket and provides the information sent to
- a MessageParser object that is evaluating the message in a separate thread.

For details about parsing client messages see Message parsing.

The Receiver is started by calling the Receiver::run method. This method is executed within a separate thread and it is listening for messages sent by a client. If a new message is received further processing is performed within another thread to keep the Receiver listening for further messages. For these threads a ThreadPool wqm is available calling MessageParser::run with message specific parameters: the key number, the LTFSDmCommServer command, and a pointer to the Connector.



2.3.4.1 Message parsing

The message parsing is invoked by Receiver calling the MessageParser::run method. Within this method it is determined which Protocol Buffer message has been received. The Protocol Buffers framework provides functions to determine which command has been issued on the client side. If the migration command has been called the following method

command.has_migrequest()

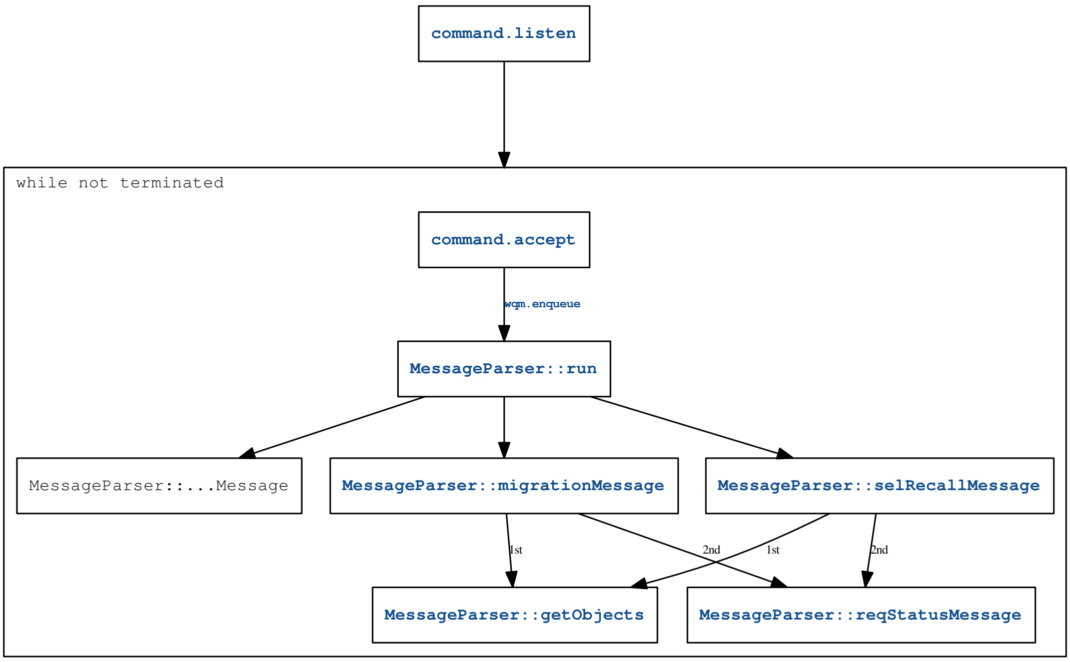
should be used to identify. This identification is part of the MessageParser::run method. Further parsing and processing is performed within command specific methods:

Message	Description
MessageParser::reqStatusMessage	query results after a migration or recall request have been sent
MessageParser::migrationMessage	migration command
MessageParser::selRecallMessage	recall command
MessageParser::requestNumber	message to get a request number
MessageParser::stopMessage	stop command
MessageParser::statusMessage	status command
MessageParser::addMessage	add command
MessageParser::infoRequestsMessage	info requests command
MessageParser::infoJobsMessage	info jobs command
MessageParser::infoDrivesMessage	info drives command

MessageParser::infoTapesMessage	info tapes command
MessageParser::poolCreateMessage	pool create command
MessageParser::poolDeleteMessage	pool delete command
MessageParser::poolAddMessage	pool add command
MessageParser::poolRemoveMessage	pool remove command
MessageParser::infoPoolsMessage	info pools command
MessageParser::retrieveMessage	retrieve command

For selective recall and migration the file names need to be transferred from the client to the backend. This is handled within the MessageParser::getObjects method. Sending the objects and querying the migration and recall status is performed over same connection like the initial migration and recall requests to be followed and not processed by the Receiver.

The following graph provides an overview of the complete client message processing:



2.3.5 Scheduler

Scheduler

The Scheduler main method Scheduler::run is started by the Server::run method and continuously running as an additional thread. For an overview about all threads that are started within the backend have a look at Server Code. Within the Server::run routine a loop is waiting on a condition of either a new request has been added or a resource became free. If there is a request that has not been scheduled so far and a corresponding resource is became this request can be scheduled. Therefore there can be two possibilities a request get scheduled:

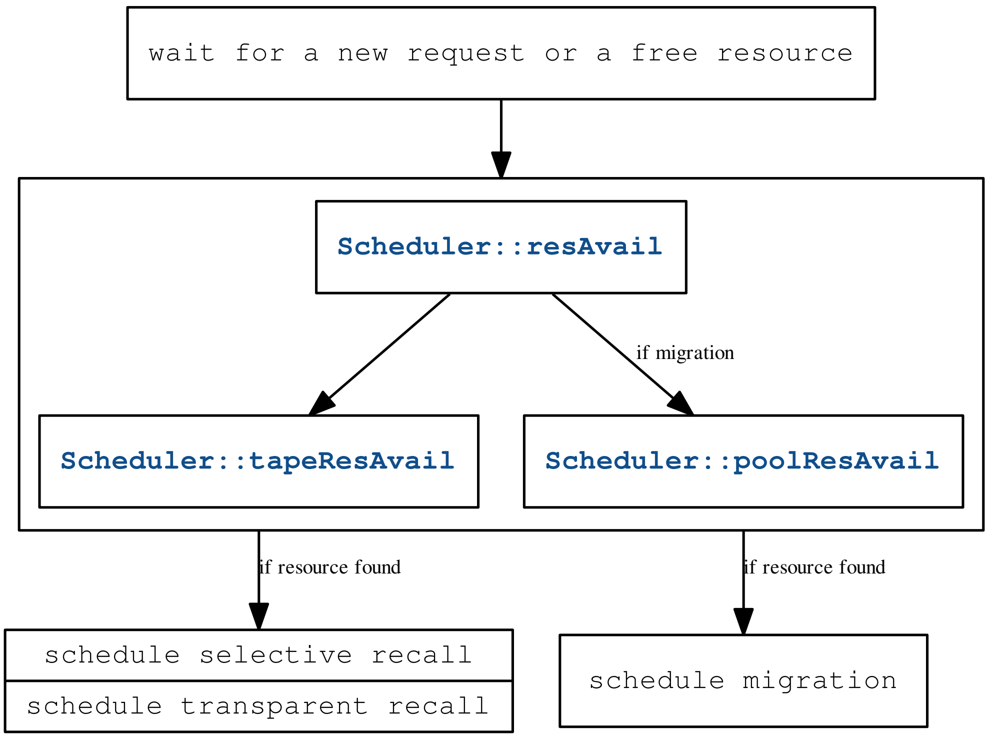
- A new request has been added an a cartridge and drive resource already is available.
- A request has been previously added but there was no free drive and cartridge resource available at that time. Now, a corresponding resource became free.

Within the outer while loop of Scheduler::runthe condition Scheduler::cond is waiting for a lock on the Scheduler::mtx mutex.

The scheduler also initiates mount and unmounts of cartridges. E.g. if there is a new request to migrate data but all available drives are empty the scheduler initiates a tape mount for a corresponding cartridge. Therefore a notification on the Scheduler::cond condition is done in the following cases:

- a new request has been added
- a request has been completed: the corresponding tape and drive resource is available thereafter
- a tape mount is completed (see LTFSDMInventory::mount): drive and cartridge can be used to schedule a request
- a tape unmount is completed (see LTFSDMInventory::unmount): drive can be used to mount a cartridge

After that Scheduler::resAvail checks if there is a resource available to schedule a request or to mount, move, or unmount cartridges (Scheduler::resAvailTapeMove). For recall, format, or check operations a specific cartridge needs to be considered (Scheduler::tapeResAvail). For migration it needs to be a cartridge from a corresponding tape storage pool where at least one file will fit on it (Scheduler::poolResAvail).



Scheduler::tapeResAvail

A tape resource is checked for availability in the following way (return statements are performed in respect to the condition):

1. If the corresponding cartridge is moving: **return false**.
2. If the corresponding cartridge is mounted (but not in use) it can be used for the current request: **return true**.
3. If there is a free (not in use) drive: **mount tape** and **return false**.

4. If there is a drive that has cartridge mounted that is not in use: **unmount tape** and **return false**.
5. Next it is checked if a operation with a lower priority can be suspended. E.g. the cartridge is used for migration recall requests have a higher priority and can led the migration request to suspend processing. If an operation already has been suspended (LTFSDMCartridge::isRequested is true): **return false.**
6. Now try to **suspend an operation**.
7. **return false**

Scheduler::poolResAvail

A tape storage pool is checked for availability in the following way (return statements are performed in respect to the condition):

1. If a cartridge of the specified tape storage pool is mounted but not in use and the remaining space is larger than the smallest file to migrate: **return true**.
2. If there is no cartridge that is not mounted there is no need to look for a cartridge from another pool to unmount: **return false**.
3. Check if there is an empty drive to mount a tape which is part of the specified pool. If this is the case: **mount tape** and **return false**.
4. Check if a for the current request there is a tape mount/unmount already in progress. If this is the case: **return false**.
5. Thereafter it is checked if there is a cartridge from another pool that is mounted but not in use. **Unmount tape** and **return false**.
6. **return false**

Schedule request

If Scheduler::resAvail is true a request can be scheduled. Depending on the operation type a new thread is created (Scheduler::subs, SubServer::enqueue) to execute:

operation type	executed method
DataBase::MIGRATION	Migration::execRequest
DataBase::SELRECALL	SelRecall::execRequest
DataBase::TRARECALL	TransRecall::execRequest

2.3.6 Migration

Migration

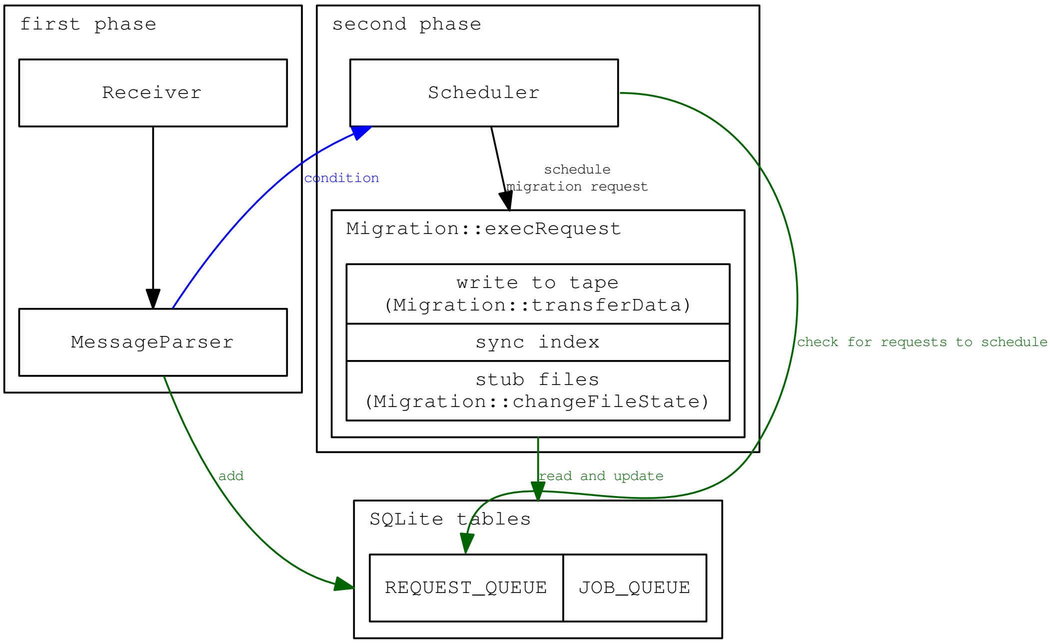
The migration processing happens within two phases:

1. The backend receives a migration message which is further processed within the MessageParser::migrationMessage method. During this processing migration jobs and migration requests are added to the internal queues.
2. The Scheduler identifies a migration request to scheduled this request.

   The migration happens within the following sequence:

   - The corresponding data is written to the selected tape.
   - The tape index is synchronized.

- The corresponding files on disk are stubbed (for migrated state as target only) and the migration state is changed.



This high level description is explained in more detail in the following subsections.

The second step will not start before the first step is completed. For the second step the required tape and drive resources need to be available: e.g. a corresponding cartridge is mounted on a tape drive. The second phase may start immediately after the first phase but it also can take a longer time depending when a required resource gets available.

1. adding jobs and requests to the internal tables

When a client sends a migration request to the backend the corresponding information is split into two parts. The first part contains information that is relevant for the whole request:

- the tape storage pools the migration is targeted to
- the target migration state (premigrated or migrated)

Thereafter the file names of the files to be migrated are sent to the backend. When receiving this information corresponding entries are added to the SQL table JOB_QUEUE. For each file name one or more jobs are created based on the number of tape storage pools being specified. After that: entries are added to the SQL table REQUEST_QUEUE. For each storage pool being specified a corresponding entry is added to that table.

The following is an example of the two tables when migrating four files to two pools:

```
sqlite> select * from JOB_QUEUE;
OPERATION   FILE_NAME              REQ_NUM     TARGET_STATE  REPL_NUM    TAPE_POOL   FILE_SIZE   FS_ID_H
----------  ---------------------  ----------  ------------  ----------  ----------  ----------  ------------
2           /mnt/lxfs/test3/file.1  6          1             0           pool_1      32768       -42298609219
2           /mnt/lxfs/test3/file.1  6          1             1           pool_2      32768       -42298609219
```

```
2          /mnt/lxfs/test3/file.2  6          1          0          pool_1     32768      -42229860921
2          /mnt/lxfs/test3/file.2  6          1          1          pool_2     32768      -42229860921
2          /mnt/lxfs/test3/file.3  6          1          0          pool_1     32768      -42229860921
2          /mnt/lxfs/test3/file.3  6          1          1          pool_2     32768      -42229860921
2          /mnt/lxfs/test3/file.4  6          1          0          pool_1     32768      -42229860921
2          /mnt/lxfs/test3/file.4  6          1          1          pool_2     32768      -42229860921
sqlite> select * from REQUEST_QUEUE;
OPERATION   REQ_NUM     TARGET_STATE  NUM_REPL    REPL_NUM    TAPE_POOL   TAPE_ID     TIME_ADDED  STATE
----------  ----------  ------------  ----------  ----------  ----------  ----------  ----------  ----------
2           6           1             2           0           pool_1                  1514983959  0
2           6           1             2           1           pool_2                  1514983959  0
```

For a description of the columns see SQLite tables.

During the scheduling later-on the cartridges will be identified as parts of the tape storage pools.

The following is an overview of this initial migration processing including corresponding links to the code parts:

MessageParser::migrationMessage:

- retrieve tape storage pool information (migreq.pools())
- retrieve target state information (premigrated or migrated state, migreq.state())
- create a Migration object
- respond back to the client with a request number
- MessageParser::getObjects: retrieving file names to migrate
  - Migration::addJob: add migration information the the SQLite table JOB_QUEUE
- Migration::addRequest: add a request to the SQLite table REQUEST_QUEUE
- MessageParser::reqStatusMessage: provide updates of the migration processing to the client

2. Scheduling migration jobs

After a migration request has been added to the REQUEST_QUEUE and there is a free tape and drive resource available to schedule this migration request the following will happen:

In Scheduler::run if a migration request is ready do be scheduled:

- update record in request queue to mark it as DataBase::REQ_INPROGRESS
- Migration::execRequest
  - if needsTape is true:
    - Migration::processFiles to transfer data to tape
      - Migration::transferData: transfer the data to tape of all files according this request
      - synchronize tape index
      - release tape for further operations since for stubbing files there is nothing written to tape
  - Migration::processFiles to change the file state
    - Migration::changeFileState: stub all corresponding files (for migrated state as target only) and perform the change of the migration state
  - update record in request queue to mark it as DataBase::REQ_COMPLETED

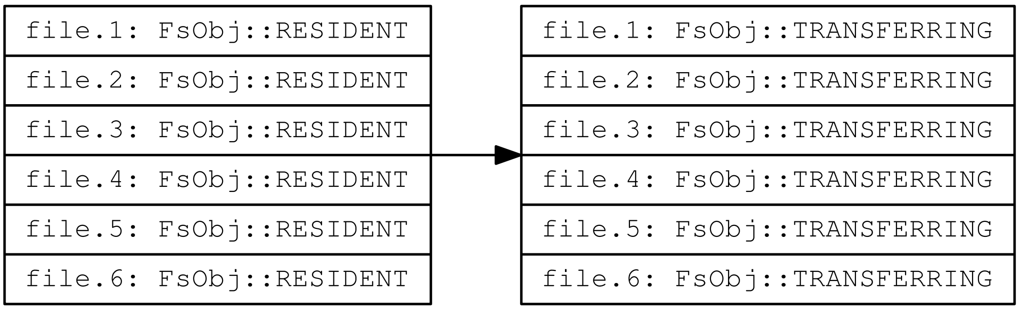
In Migration::execRequest adding an entry to the mrStatus object is necessary for the client that initiated the request to receive progress information.

Migration::processFiles

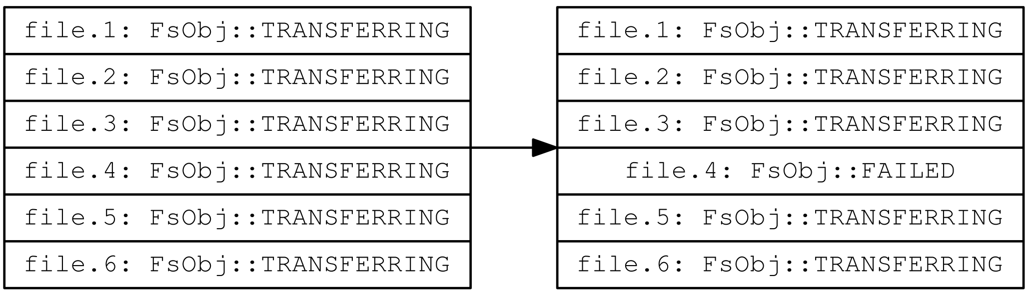
The Migration::processFiles method is called twice first to transfer the file data and a second time to stub them if necessary and to perform the change of the migration state . If all files to be processed are already premigrated there is no need to mount a cartridge. In this case the data transfer step is skipped. If the target state is LTFSDmProtocol::LTFSDmMigRequest::PREMIGRATED files are not stubbed and only the migration state is changed to premigrated.

The Migration::processFiles method in general perform the following steps:

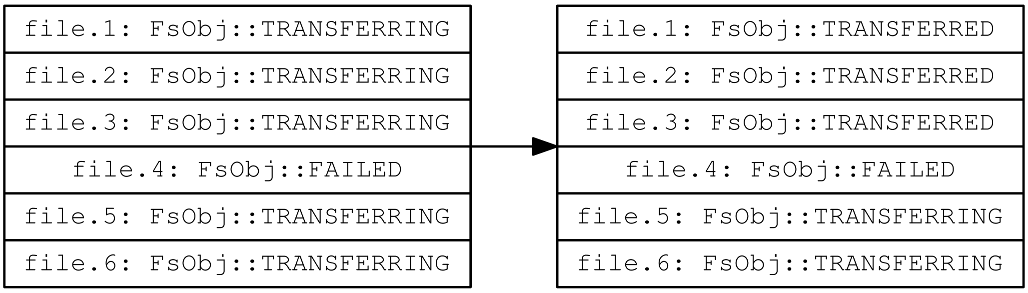
1. All corresponding jobs are changed to FsObj::TRANSFERRING or FsObj::CHANGINGFSTATE depending on the migration phase. The following example shows this change for the first phase of six files:



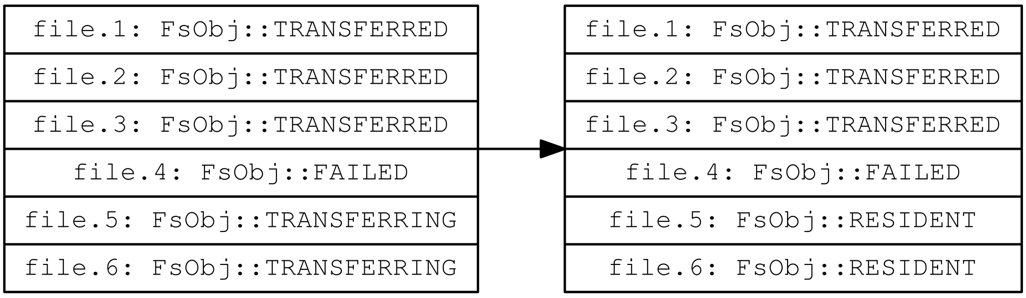
2. Process all these jobs in FsObj::TRANSFERRING or FsObj::CHANGINGFSTATE state which results in the data transfer or stubbing of all corresponding files. The following change indicates that the data transfer of file file.4 failed:



3. A list is returned containing the inode numbers of these files where the previous operation was successful. Change all corresponding jobs to FsObj::TRANSFERRED or FsObj::MIGRATED depending of the migration phase. The following changed indicates that data transfer stopped before file file.5:



4. The remaining jobs (those where no corresponding inode numbers were in the list) have not been processed and need to be changed to the original state if these were still in FsObj::TRANSFERRING or FsObj::CHANGI-NGFSTATE state. Jobs that failed in the second step already have been marked as FsObj::FAILED. A reason for remaining jobs left over from the second step could be that a request with a higher priority (e.g. recall) required the same tape resource. This change is shown below for the remaining two files of the example:



If more than one job is processed the data transfer or migration state change operations can be performed in parallel. For data transfer each file needs to be written continuously on tape and therefore the writes are serialized. For this purpose two or more ThreadPool objects exists:

- one ThreadPool object for migration state change: Server::wqs
- for each LTFSDMDrive object one ThreadPool object: LTFSDMDrive::wqp to transfer data to tape.

In the data transfer case the Migration::transferData method is executed and in case of changing the migration state it is the Migration::changeFileState method. Each of these methods operate on a single file.

The following table provides a sequence of changes of different items that are changing during the migration of a resident file:

	JOB_QUEUE FsObj::file_-state	attribute on disk: FuseFS::mig_-info::state_-num and tape id	tape index synchronized	data on disk	data on tape
1.	RESIDENT	no attributes	-	+	-
2.	TRANSFERRI-NG	no attributes	-	+	-
3.	TRANSFERRI-NG	IN_MIGRATIO-N	-	+	-
4.	TRANSFERRI-NG	IN_MIGRATIO-N	-	+	+
5.	TRANSFERRI-NG	IN_MIGRATIO-N && tape id	-	+	+
6.	TRANSFERR-ED	IN_MIGRATIO-N && tape id	-	+	+
7.	TRANSFERR-ED	IN_MIGRATIO-N && tape id	+	+	+
8.	CHANGINGFS-TATE	IN_MIGRATIO-N && tape id	+	+	+
9.	CHANGINGFS-TATE	STUBBING && tape id	+	+	+
10.	CHANGINGFS-TATE	STUBBING && tape id	+	-	+
11.	CHANGINGFS-TATE	MIGRATED && tape id	+	-	+

Each of these steps have a reason:

1. This is the initial state when a resident file is newly added to the JOB_QUEUE table.
2. The migration is split into three phases:
   (a) transfer file data to a tape
   (b) The the tape index stored on disk is written to the tape
   (c) stub the file and mark it as migrated

   In a first step all files that should be transferred to tape are marked according that operation to see which were left over in case the transfer has been suspended e.g. by a recall request on the same tape.
3. The file attribute is changing to IN_MIGRATION. This intermediate state is to perform a proper cleanup in case the back end process terminates unexpectedly. See FuseFS::recoverState for recovery of a migration state.
4. The file data is transferred to tape.
5. The corresponding tape id is added to the attributes of the file.
6. The state in the JOB_QUEUE table is changed to TRANSFERRED since that data transfer was successful.
7. The the tape index stored on disk is written to the tape (migration phase 2 of 3).
8. The last (third) migration phase is to stub the file and to mark it as migrated. It starts to change the state in the JOB_QUEUE table for all files that are targeted for this phase to CHANGINGFSTATE. Even the stubbing operation cannot be suspended it is to keep the the first and the third phase similar.

9. The file attribute is changed to STUBBING. Like previously this intermediate state is to perform a proper cleanup in case the back end process terminates unexpectedly. See FuseFS::recoverState for recovery of a migration state.
10. The file is truncated to zero.
11. The attribute is changed to MIGRATED.

Migration::transferData

For data transfer the following steps are performed:

1. In a loop the data is read from disk and written to tape.
2. The FILE_PATH attribute is set on the data file on tape.
3. A symbolic link is created by recreating the original full path on tape pointing to the corresponding data file.
4. The status object mrStatus gets updated for the output statistics.
5. The tape is added to the attribute of the data file on tape.

For data transfer each file needs to be written continuously on tape. Since the copy of data from disk to tape is performed in a loop by doing the reads and writes this loop is serialized by a std::mutex LTFSDMDrive::mtx.

Migration::changeFileState

For the change of the migration state (includes stubbing in the case that the migrated state is the target) the following steps are performed:

1. The attributes on the disk file are changed accordingly.
2. The file is truncated to zero size (only if the migrated state is the target).
3. The status object mrStatus gets updated for the output statistics.

It is required that the attributes are changed before the file is truncated. It needs to be avoided that a file is truncated before it changes to migrated state. Otherwise: in an error case it could happen that the file is truncated but still in transferred state which indicates that the data locally is available.

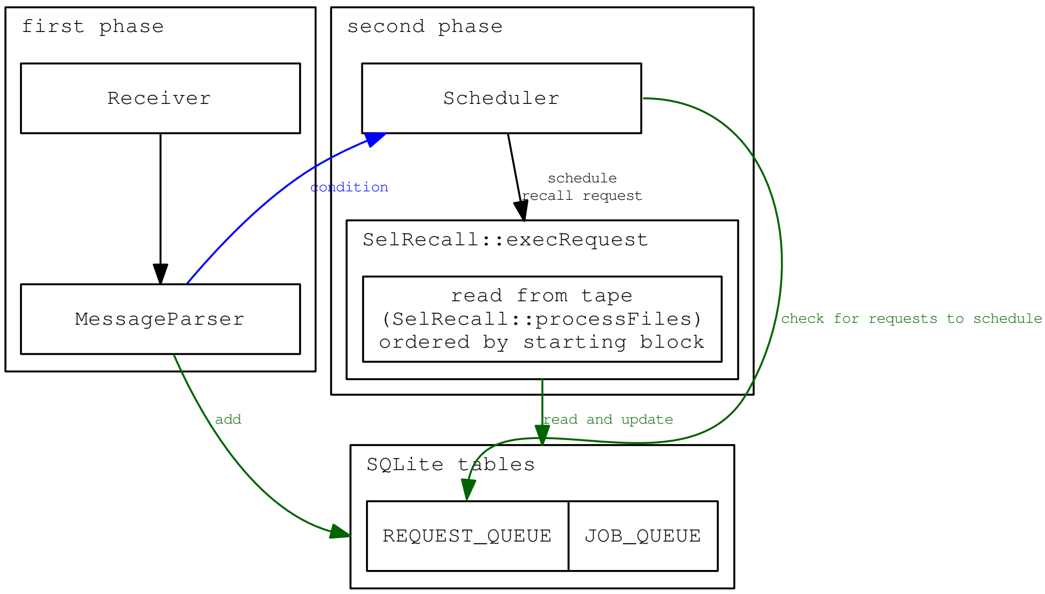
2.3.7 Selective Recall

SelRecall

The selective recall processing happens within two phases:

1. The backend receives a selective recall message which is further processed within the MessageParser::selRecallMessage method. During this processing selective recall jobs and selective recall requests are added to the internal queues.
2. The Scheduler identifies a selective recall request to get scheduled. The order of files being recalled depends on the starting block of the data files on tape:

```
const std::string SelRecall::SELECT_JOBS =
        "SELECT FILE_NAME, FILE_STATE, I_NUM FROM JOB_QUEUE WHERE REQ_NUM=%1%"
                " AND TAPE_ID='%2%'"
                " AND (FILE_STATE=%3% OR FILE_STATE=%4%)"
                " ORDER BY START_BLOCK";
```



This high level description is explained in more detail in the following subsections.

The second step will not start before the first step is completed. For the second step the required tape and drive resources need to be available: e.g. a corresponding cartridge is mounted on a tape drive. The second phase may start immediately after the first phase but it also can take a longer time depending when a required resource gets available.

1. adding jobs and requests to the internal tables

When a client sends a selective recall request to the backend the corresponding information is split into two parts. The first part contains information that is relevant for the whole request: the target recall state. A file can be recalled to resident state or to premigrated state.

Thereafter the file names of the files to be recalled are sent to the backend. When receiving this information corresponding entries are added to the SQL table JOB_QUEUE. For each file one entry is created. After that an entry is added to the SQL table REQUEST_QUEUE.

This is an example of these two tables in case of selectively recalling a few files:

```
sqlite> select * from JOB_QUEUE;
OPERATION   FILE_NAME                 REQ_NUM     TARGET_STATE  REPL_NUM    TAPE_POOL   FILE_SIZE   FS_ID_H
----------  ------------------------  ----------  ------------  ----------  ----------  ----------  ------------
1           /mnt/lxfs/test2/file.0    2           0                                     32768       -42298609219
1           /mnt/lxfs/test2/file.2    2           0                                     32768       -42298609219
1           /mnt/lxfs/test2/file.4    2           0                                     32768       -42298609219
sqlite> select * from REQUEST_QUEUE;
OPERATION   REQ_NUM     TARGET_STATE  NUM_REPL    REPL_NUM    TAPE_POOL   TAPE_ID     TIME_ADDED  STATE
----------  ----------  ------------  ----------  ----------  ----------  ----------  ----------  ----------
1           2           0                                                 D01301L5    1515426071  1
```

For a description of the columns see SQLite tables.

The following is an overview of this initial selective recall processing including corresponding links to the code parts:

`MessageParser::selRecallMessage`:

- create a SelRecall object with target state information (premigrated or resident state, recreq.state())
- respond back to the client with a request number
- MessageParser::getObjects: retrieving file names to recall
  - SelRecall::addJob: add recall information the the SQLite table JOB_QUEUE
- SelRecall::addRequest: add a request to the SQLite table REQUEST_QUEUE
- MessageParser::reqStatusMessage: provide updates to the recall processing to the client

2. Scheduling selective recall jobs

After a selective recall request has been added to the REQUEST_QUEUE and there is a free tape and drive resource available to schedule this selective recall request the following will happen:

Scheduler::run:

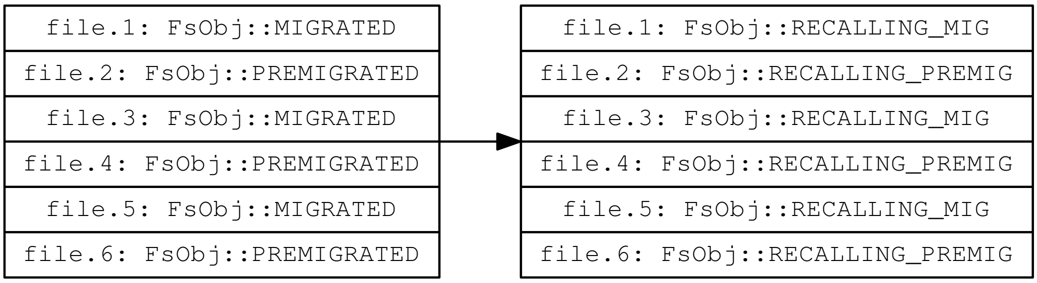
- if a selective request is ready do be scheduled:
  - update record in request queue to mark it as DataBase::REQ_INPROGRESS
  - SelRecall::execRequest
    - add status: mrStatus.add
    - call SelRecall::processFiles depending the target state SelRecall::targetState
    - if needsTape is true:
      - release tape for further operations since for stubbing files there is nothing written to tape
    - update record in request queue to mark it as DataBase::REQ_COMPLETED

In SelRecall::execRequest adding an entry to the mrStatus object is necessary for the client that initiated the request to receive progress information.

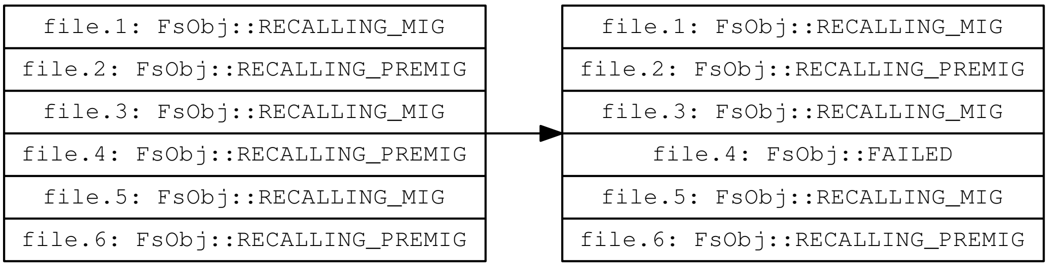
SelRecall::processFiles

The SelRecall::processFiles method is traversing the JOB_QUEUE table to process individual files for selective recall. In general the following steps are performed:

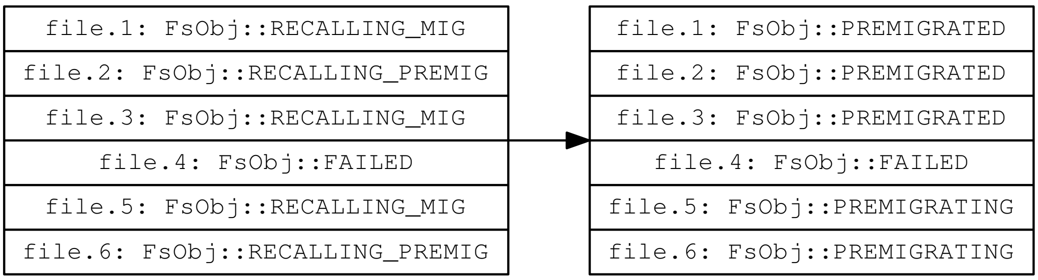
1. All corresponding jobs are changed to FsObj::RECALLING_MIG or FsObj::RECALLING_PREMIG depending if it is called for files in migrated or in premigrated state. The following example shows this change regarding six files:



2. Process all these jobs in FsObj::RECALLING_MIG or FsObj::RECALLING_PREMIG state which results in the recall of all corresponding files. For all jobs in FsObj::RECALLING_PREMIG state there will no data transfer happen. The following change indicates that the recall of file file.4 failed:

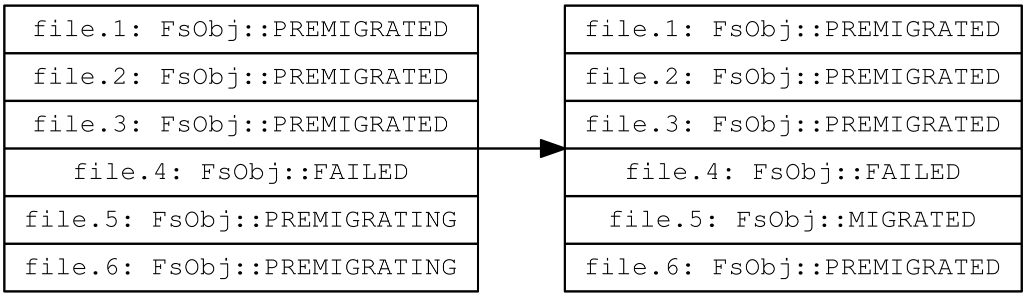


3. A list is returned containing the inode numbers of these files where the previous operation was successful. Change all corresponding jobs to FsObj::PREMIGRATED or FsObj::RESIDENT depending of the target state. The following changed indicates that recall stopped before file file.5 and target state is premigrated:



4. The remaining jobs (those where no corresponding inode numbers were in the list) have not been processed and need to be changed to the original state if these were still in FsObj::RECALLING_MIG or FsObj::RECALLING_PREMIG state. Jobs that failed in the second step already have been marked as FsObj::FAILED. A reason for remaining jobs left over from the second step could be that a request with a higher priority

(transparent recall) required the same tape resource. This change is shown below for the remaining two files of the example:



In opposite to migration recalls are not performed in parallel. For an optimal performance the data should be read serially from tape in the order of the starting block of each data file.

SelRecall::recall

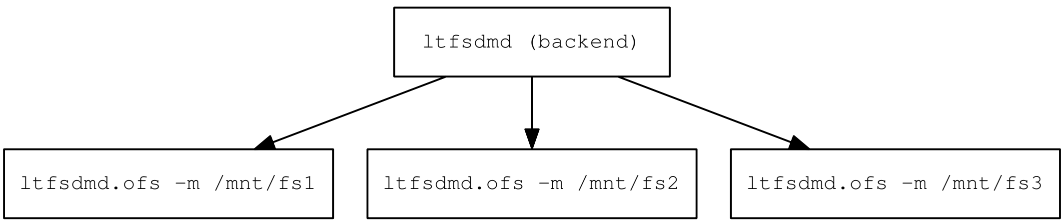
Recalling an individual file is performed according the following steps:

1. If state is FsObj::MIGRATED data is read in a loop from tape and written to disk.
2. The attributes on the disk file are updated or removed in the case of target state resident.

2.3.8 Transparent Recall

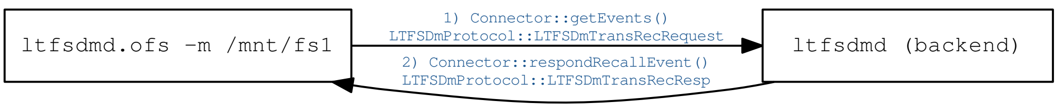
TransRecall

For each file system that is managed with LTFS Data Management a Fuse overlay file system is created. After that the original file system should not be used anymore by an user or by an application: only the Fuse overlay file system should be used instead. For each of these Fuse overlay file systems an additional process is started:



Within the Fuse processes read, write, and truncate calls on premigrated or migrated files are intercepted since there is a requirement to recall (transfer data back from tape to disk) data or to perform a file state change. The Fuse overlay file system as part of the Fuse connector is described in more detail at Fuse connector.

The data transfer and the file system change of a file are performed within the backend. Therefore there needs to be a communication between the Fuse overlay file system processes and the backend regarding files to recall. The communication method is the same like between the client and the backend: local socket communication and Google protocol buffers for message serialization:



If a recall request LTFSDmProtocol::LTFSDmTransRecRequest has been sent to the backend within the Fuse process read, write, and truncate processing is blocked until the backend responds with Connector::respondRecallEvent.

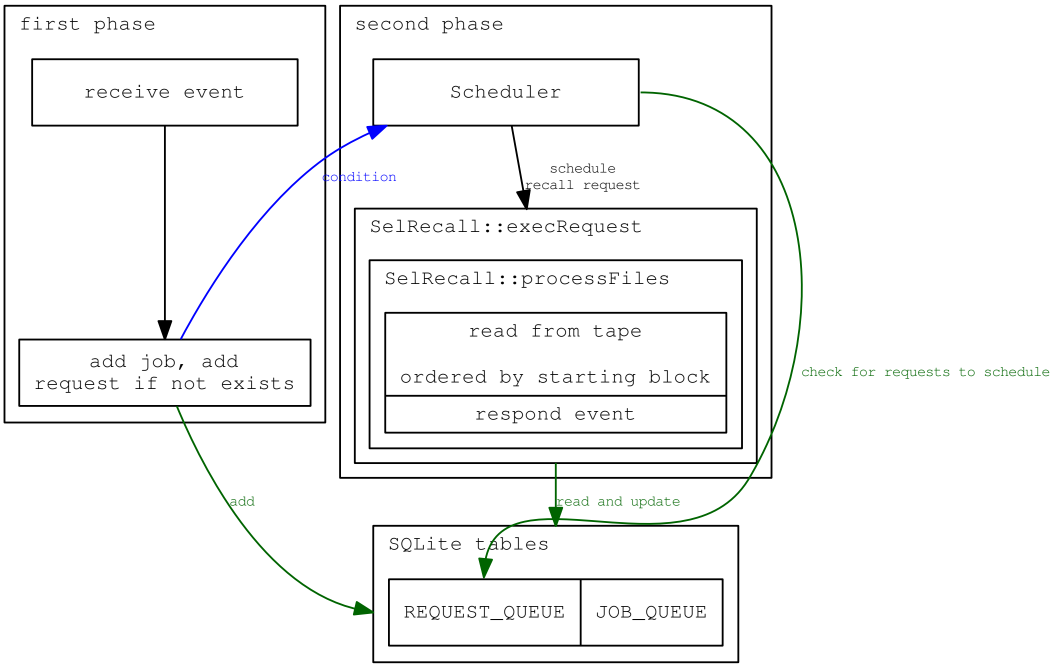
The transparent recall processing within the backend happens within two phases:

1. One backend thread ("RecallD" executing TransRecall::run) waits on a socket for recall events. Recall events are are initiated by applications that perform read, write, or truncate calls on a premigrated or migrated files. A corresponding job is created within the JOB_QUEUE table and - if it does not exist - a request is created within the REQUEST_QUEUE table.

2. The Scheduler identifies a transparent recall request to get scheduled. The order of files being recalled depends on the starting block of the data files on tape:

```
const std::string TransRecall::SELECT_JOBS =
        "SELECT FS_ID_H, FS_ID_L, I_GEN, I_NUM, FILE_NAME, FILE_STATE, TARGET_STATE, CONN_INFO  FROM
       JOB_QUEUE"
                " WHERE REQ_NUM=%1%"
                " AND (FILE_STATE=%2% OR FILE_STATE=%3%)"
                " AND TAPE_ID='%4%' ORDER BY START_BLOCK";
```

If the transparent recall job is finally processed (even it is failed) the event is responded as a Protocol Buffers message (LTFSDmProtocol::LTFSDmTransRecResp).



This high level description is explained in more detail in the following subsections.

If there are multiple recall events for files on the same tape only one request is created within the REQUEST_QUEUE table. This request is removed if there are no further outstanding transparent recalls for the same tape. If there is a new transparent recall event and if a corresponding request already exists within the REQUEST_QUEUE table this existing request is used for further processing this request/event.

The second step will not start before the first step is completed. For the second step the required tape and drive resources need to be available: e.g. a corresponding cartridge is mounted on a tape drive. The second phase may start immediately after the first phase but it also can take a longer time depending when a required resource gets available.

1. adding jobs and requests to the internal tables

One backend thread exists (see Server Code) that executes the TransRecall::run method to wait for recall events. Recall events are sent as Protocol Buffers messages (LTFSDmProtocol::LTFSDmTransRecRequest) over a socket. The information provided contains the following:

- opaque information specific to the connector
- an indicator if a file should be recall to premigrated or to resident state
- the file uid (see fuid_t)
- the file name

Thereafter the tape id for the first tape listed within the attributes is obtained. The recall will happen from that tape. There currently is no optimization if the file has been migrated to more than one tape to select between these tapes in an optimal way.

To add a corresponding job within the JOB_QUEUE table or if necessary a request within the REQUEST_QUEUE table an additional thread is used as part of the ThreadPool wqr executing the method TransRecall::addJob.

This is an example of these two tables in case of transparently recalling a few files:

```
sqlite> select * from JOB_QUEUE;
OPERATION   FILE_NAME              REQ_NUM     TARGET_STATE  REPL_NUM    TAPE_POOL   FILE_SIZE   FS_ID_H
----------  ---------------------  ----------  ------------  ----------  ----------  ----------  -----------
0           /mnt/lxfs/test2/file.4  3          1             -1                      32768       -42298609219
0           /mnt/lxfs/test2/file.2  3          1             -1                      32768       -42298609219
0           /mnt/lxfs/test2/file.0  3          1             -1                      32768       -42298609219
sqlite> select * from REQUEST_QUEUE;
OPERATION   REQ_NUM     TARGET_STATE  NUM_REPL    REPL_NUM    TAPE_POOL   TAPE_ID     TIME_ADDED  STATE
----------  ----------  ------------  ----------  ----------  ----------  ----------  ----------  ----------
0           3                                                             D01301L5    1515591742  1
```

For a description of the columns see SQLite tables.

The following is an overview of this initial transparent recall processing including corresponding links to the code parts:

`TransRecall::run`:

- `while not terminating (Connector::connectorTerminate == false)`
  - `wait for events: Connector::getEvents`
  - `create FsObj object according the recall information recinfo`
  - `determine the id of the first cartridge from the attributes`
  - `enqueue the job and request creation as part of the ThreadPool wqr executing the method TransRecall::addJob.`

`TransRecall::addJob`:

- `determine path name on tape`
- `add a job within the JOB_QUEUE table`
- `if a request already exists: if (reqExists == true)`
  - `change request state to new`
- `else`
  - `create a request within the REQUEST_QUEUE table`

2. Scheduling transparent recall jobs

After a transparent recall request is ready to be scheduled and there is a free tape and drive resource available to schedule this transparent recall request the following will happen:

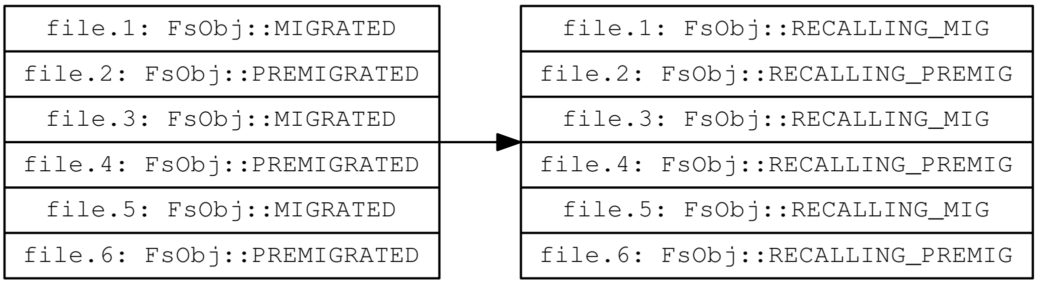
`Scheduler::run`:

- `if a transparent request is ready do be scheduled:`
  - `update record in request queue to mark it as DataBase::REQ_INPROGRESS`
  - `TransRecall::execRequest`
    - `call TransRecall::processFiles`
      - `respond recall event Connector::respondRecallEvent`
    - `if there are outstanding transparent recall requests for the same tape (remaining)`
      - `update record in request queue to mark it as DataBase::REQ_NEW`
    - `else`
      - `delete request within the REQUEST_QUEUE table`

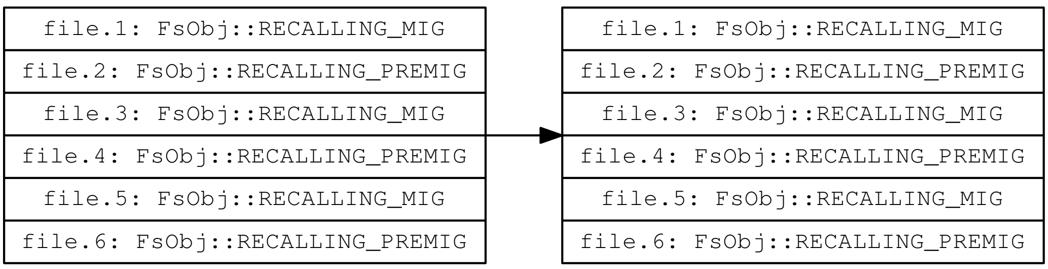
TransRecall::processFiles

The TransRecall::processFiles method is traversing the JOB_QUEUE table to process individual files for transparent recall. In general the following steps are performed:

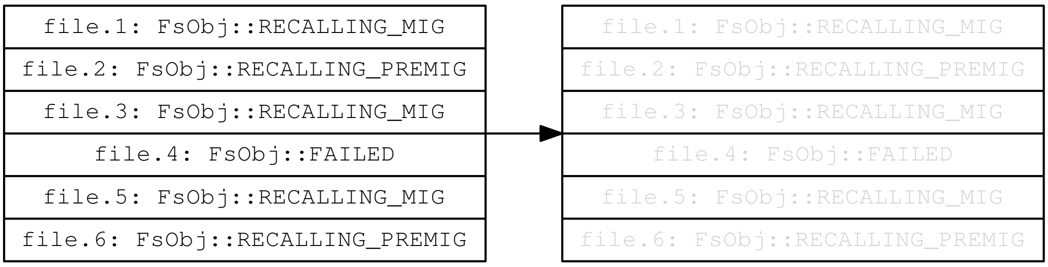
1. All corresponding jobs are changed to FsObj::RECALLING_MIG or FsObj::RECALLING_PREMIG depending if it is called for files in migrated or in premigrated state. The following example shows this change regarding six files:



2. Process all these jobs in FsObj::RECALLING_MIG or FsObj::RECALLING_PREMIG state which results in the recall of all corresponding files. For all jobs in FsObj::RECALLING_PREMIG state there will no data transfer happen. The result of each individual transparent recall is stored in a respinfo_t respinfo std::list object (no change within the JOB_QUEUE table):



3. The corresponding jobs are deleted from the JOB_QUEUE table:



4. All entries within the respinfo_t respinfo std::list object are responded if processing was successful or not (respinfo.succeeded) by calling Connector::respondRecallEvent.

In opposite to migration recalls are not performed in parallel. For an optimal performance the data should be read serially from tape in the order of the starting block of each data file.

TransRecall::recall

Recalling an individual file is performed according the following steps:

1. If state is FsObj::MIGRATED data is read in a loop from tape and written to disk.
2. The attributes on the disk file are updated or removed in the case of target state resident.

Chapter 3

Messaging

Messaging System

LTFS Data Management writes output to the console and to log files. Every output to these two locations should be regarded as a message even it is a single character. Tracing does not used messages since only values of variables are printed out. All messages are consolidated within a single file `messages.cfg` that is located in the root of the code tree.

There are two types of messages:

- Informational messages do not show up the message identifier. Those messages can be used by specifying the INFO() macro.
- Messages that should show up the message identifier. For those the MSG() macro should be specified.

The INFO() macro is especially used for the output of client commands.

The `messages.cfg` has a special format:

- Empty lines are allowed.
- A '#' character at the beginning of a line indicates a comment.
- Usually a message starts with a message identifier followed by the message surrounded by quotes.
- If a line starts without a message identifier the message text is added to the previous message.

The message text has to be written in c printf style format. E.g.:

```
LTFSDMX0001E "Unable to setup tracing: %d.\n"
```

The message identifier is assembled in the following way:

```
LTFSDM[X|C|S|D|F|L]NNNN[I|E|W]
```

The message identifier components have the following meaning:

characters	meaning
X	common message used in multiple parts of the code (client, server, ...)

C	a client message
S	a server message
D	a message used by the dmapi connector
F	a message used by the Fuse connector
L	a message used by LTFS LE
NNNN	a four digit number
I	an informational message
E	an error message
W	a warning

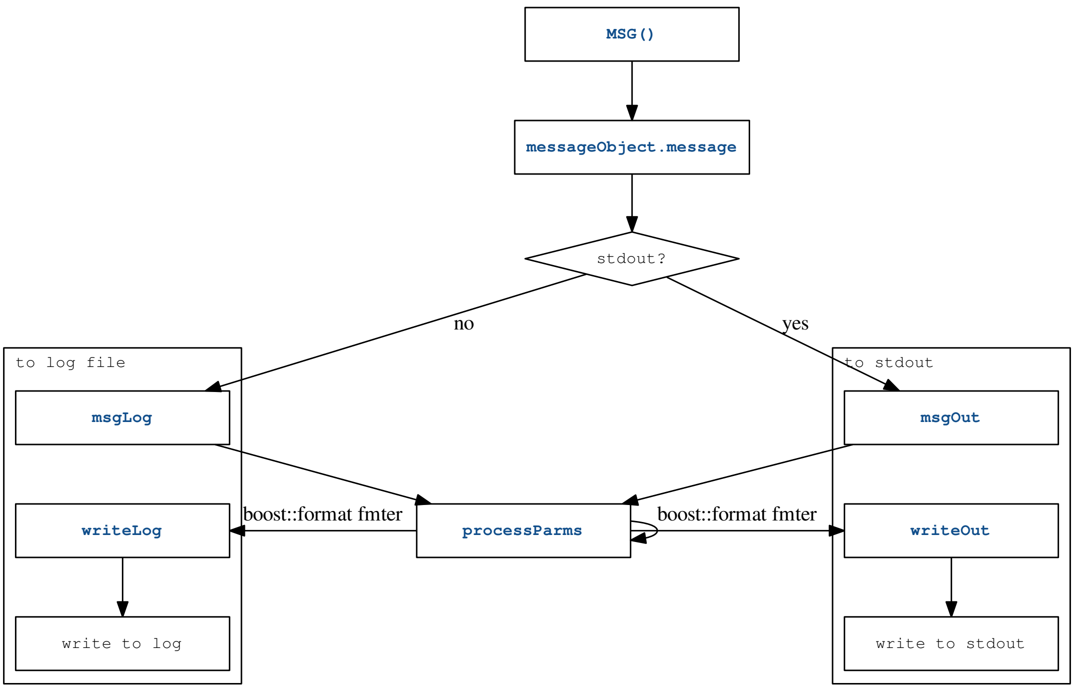
A line feed is not automatically added. It is necessary to add a "\n" sequence if required.

There is a message compiler msgcompiler.cc that transforms the text based `messages.cfg` message file into c++ code. This operation is done at the beginning of the build process. If that has not happened symbols are missing and integrated development environment may show up errors.

The MSG() macro automatically add the file name and the line number to the output. The class Message is responsible to process the message string and corresponding arguments. Internally the `Boost Format library` is used to perform the formatting.

For each process there exists a messaging object messageObject to perform the message processing. This messaging object should not be used directly but is used internally as part of the MSG() and INFO() macros.

The following gives an overview about the internal processing of a message:



Chapter 4

Tracing

The tracing system is used to print out values of variables of the source code into a trace file.

Todo enable tracing on the client side

At the moment tracing only is enabled on the server side.

There are five different trace levels:

trace level	numeric value	description
none	0	no tracing
always	1	print trace message always
error	2	print trace message in an error case
normal	3	standard tracing
full	4	perform a full tracing

Todo trace levels need to be reworked, do not make sense

The standard trace level is "normal". Different trace levels can be enabled by starting the backend directly: see server code how to start the backend in this way.

Todo use rsyslog facility to write trace output

The trace information is written to /var/rum/ltfsdm/LTFSDM.trc*.

In the following there is some sample output:

```
2017-12-07T15:42:46.366490:[004502:012271]:------TransRecall.cc(0063): filename('/mnt/lxfs/test2/file.262')
2017-12-07T15:42:46.366508:[004502:012271]:------TransRecall.cc(0067): tapeId(D01302L5)
2017-12-07T15:42:46.366653:[004502:031034]:--------Migration.cc(0113): fileName(/mnt/lxfs/test2/file.945), rep
```

The trace output has the following structure:

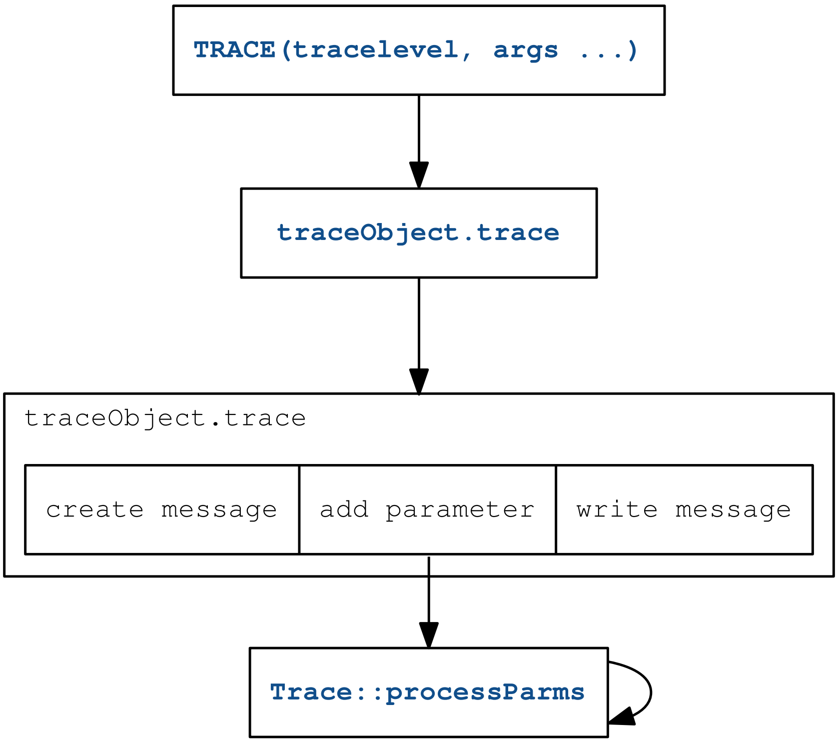
date T **time** :[**pid** : **tid**]:--- **file_name**(**line_number**): **variable_name_1** (**value_1**),**variable_name_2** (**value_2**),...

For each process there exists a tracing object traceObject that never should be used directly used but is used internally as part of the TRACE() macro. The macro TRACE() automatically adds the corresponding file name and the line number to the output.

The usage is the following:

```
TRACE(tracelevel, var1, var2, ...)
```

The following gives an overview about the internal processing of tracing:


TRACE(tracelevel, args ...)
traceObject.trace
traceObject.trace
create message
add parameter
write message
Trace::processParms