IBM
Spectrum
Scale

IBM Spectrum Scale™
Automation of storage services

Nils Haustein, Achim Christ | EMEA Storage Competence Center | Version: 1.7

Contents

Acknowledgements

Thanks to Sven Zachoval (IBM TSS, Germany) for giving me the chance to develop this framework in the context of a client project.

Thanks to Simon Lorenz, Mathias Dietz, Andreas Koeninger and Markus Rohwedder from the Spectrum Scale development team for designing and implementing the sending of custom events.

Thanks to Alton Hill from our business partner Sirius for his patience testing the code and continuously providing feedback. This has led to many improvements.

Introduction

IBM Spectrum Scale™ is a software-defined scalable parallel file system providing a comprehensive set of storage services. Some of the differentiating storage services are the integrated backup function and storage tiering. These services typically run in the background according to pre-defined schedules. This whitepaper presents a flexible framework for automating Spectrum Scale storage services.

One may ask:

“Why having a framework for automating backup and the like? Just start it!”

Well it is not that trivial. For example, if you want to run backup in a cluster, on which node do you schedule it? And what happens if this node happens to be down? Will your backup run? And what happens if the file system is not available when the backup is scheduled? How do you manage log files for processes running in the background? All these questions and more are addressed in this framework. It is extensible and may address even more challenges in the future.

This paper is intended for Spectrum Scale administrators who have experience with administration, backup and storage tiering. The code snippets presented herein are examples only. The actual code and latest readme describing the functionality and the parameters and options are is available in the open source repository [6].

For the latest information refer to the readme file:
https://github.com/IBM/SpectrumScaleAutomation

Automation

In this chapter we present a flexible framework for automating Spectrum Scale storage services that typically run in the background in accordance to pre-defined schedules. This framework is based on processes that can be implemented as script programs and scheduled through standard scheduling services such as cron or systemd Timers in Unix or Linux. Some sample code for this framework is stored in an open source repository [6].

We focus on automating commonly used storage services such as backup (using `mmbackup`) and storage tiering (using `mmapplypolicy`). The framework, however, is not limited to these services and can easily be extended. Some storage services can also be scheduled via the Spectrum Scale GUI (e.g. snapshots).

The framework relies on the following components:

- The *control components* selects the appropriate cluster node initiating the storage service, starts the storage service if the node state is appropriate, manages logging, log files and return codes. The control component is typically invoked by the scheduler and the storage services being started might be backup or storage tiering (see section Control component)
- The *backup component* performs the backup using the `mmbackup`-command (see section Backup component)
- The *storage tiering component* performs pre-migration or migration using the `mmapplypolicy`-command (see section Storage tiering component)
- The schedule invokes the control component according to pre-defined schedules (see section Scheduling)

Figure 1 shows the relations and control flow between the components of the automation framework.

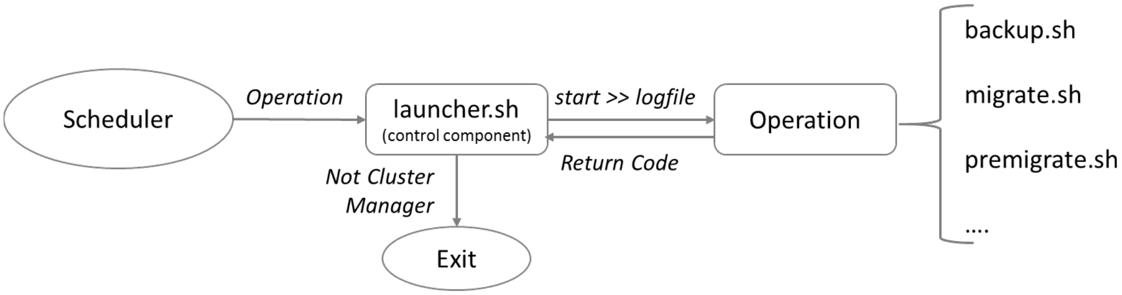


Figure 1: General flow of automation framework

As shown in Figure 1, the scheduler invokes the control component that is represented by the script `launcher.sh`. The schedule is implemented on at least all nodes that have a manager role. The control component determines if it is the cluster manager and if this is the case it starts the operation and redirects the console output to the log file. The operation can be backup (`backup.sh`) or migrate (`migrate.sh`) or re-migrate (`premigrate.sh`). Likewise the control component can start other customized scripts.

The framework requires that all cluster nodes with a manager role assigned must be able to run the automation components. These nodes must not necessarily be the nodes performing the storage service operation (such as backup or migration).

Control component

The control component (`launcher.sh`) is installed on every node that can become cluster manager, i.e. on every quorum node. It is typically invoked by the scheduler and performs the following steps:

- Checks if the local node is the cluster manager (see section Node selection)
- Checks the status of the local node and file system (see section Node status)
- Assigns the storage service (see section Assigning storage service)
- Manages log files (see section Logging)
- Starts the storage service (see section Running storage service)
- Checks return codes and sends notification (see section Notification based on return codes)

These steps are explained in the subsequent sections. Section Further considerations describes some additional enhancements.

For the latest information about the control component, refer to the readme file: https://github.com/IBM/SpectrumScaleAutomation

Node selection

One of the first challenges with running storage services that have a cluster-wide scope is the selection of the node where the operation is started. Most storage services such as backup must be started only on one node. The method for the selection of the cluster node must be reliable and tolerate that any node in the cluster can be down.

The trick leveraged in the framework is to run the storage service on the cluster manager. As long as the cluster is active there is exactly one cluster manager. The cluster manager is elected among all quorum nodes. All quorum nodes run the control component at the same time, but only the node with the cluster manager role will actually continue the operation. All other quorum nodes that are not cluster manager will terminate.

The following script snippet checks if the node is the cluster manager and only if this is the case it will continue:

```
#
# The following snippet is part of the script launcher.sh
#--------------------------------------------------------------------
#
# initialize the local node name
localNode=$(mmlsnode -N localhost | cut -d'.' -f1)

# initialize the cluster manager name
clusterMgr=$(mmlsmgr -c | sed 's|.*(||' | sed 's|)||')
# check if the local node is the cluster manager and if not exit 0
if [ "$localNode" != "$clusterMgr" ];
then
  echo "INFO: this node ($localNode) is not cluster manager
($clusterMgr), exiting."
  # exit with good return code because this is not an error
  exit 0
fi

# … perform the next steps …
```

The above code will exit with return code 0 (good status, see section Return codes) if the node is not the cluster manager. Otherwise the script will go on to the next steps.

Node status

The next challenge is to make sure that the node initiating the storage service operation has the proper status. This node must be active and it must have the file system mounted. Even though the cluster manager is always active – otherwise the cluster would be offline – checking for the node to be active is used to identify transitions of the cluster manager role.

The following script snippet shows the logic for checking the node and file system status:

```
#
# The following snippet is part of the script launcher.sh
#---------------------------------------------------------------------
#
# define the file system name
fsName=myfs

# check if this node is active
state=””
state=$(mmgetstate -Y | grep -v ":HEADER:" | cut -d':' -f 9)
if [[ ! "$state" == "active" ]];
then
  echo "ERROR: node $localNode is not active (state=$state), exiting."
  exit 2
fi

# check if file system is mounted on the local node
mounted=0
mounted=$(mmlsmount $fsName -L | grep "$localNode" | wc -l)
if (( mounted == 0 ));
then
  echo "ERROR: file system $fsName is not mounted on node $localNode,
exiting."
  exit 2
fi

# … perform the next steps …
```

The name of the local nodes (`$localNode`) was defined in the previous step (see section Node selection). The name of the file system is pre-defined, but can also be given as a parameter to this components (see section Multiple file systems).

The above code will exit with return code 2 (error status, see section Return codes) if the node is not in status active or if the file system is not mounted. The consequence of this condition is that the storage service operation cannot be performed at this time. Otherwise the script will carry on with the next steps.

Assigning storage service

The control component can start a variety of storage service processes. Each storage service process might be represented by a unique script or unique parameter of a common script. In order to decide which storage service process to be started the control component obtains the name of the storage service from the caller (the caller can be an administrator using the CLI or a scheduler such as *cron* or *systemd* Timers).

The following snippet shows the assignment of the storage service script in accordance to the parameter given to the control component:

```
#
# The following snippet is part of the script launcher.sh
#----------------------------------------------------------------------
#
# assign parameter, check parameter and assign script
scriptPath="~/myscripts"
op=$1

case $op in
"backup")
  # runs Backup
  cmd=$scriptPath"/backup.sh";;
“migrate")
  # runs migration (scheduled)
  cmd=$scriptPath"/migrate.sh";;
“premigrate")
  # runs premigration (scheduled)
  cmd=$scriptPath"/premigrate.sh";;
*) echo "ERROR: wrong operation code: $op"
   echo "Syntax: $0 <operation> <filesystem-name>"
   echo "Operation can be: backup, migrate or premigrate"
   exit 2;;
esac

# … perform the next steps …
```

According to the example above, the following parameters are defined:

- backup: assigns the backup script (`backup.sh`) and runs it
- migrate: assigns the migration script (`migrate.sh`) and runs it
- premigrate: assigns the premigration script (`premigrate.sh`) and runs it.

Consequently the control component is invoked with one of this parameters, for example:

```
# launcher.sh backup
```

This approach can be flexibly expanded by adding new storage service scripts and parameters to the above code.

If there are multiple file systems in the cluster the file system name can be given as another parameter to the control component (see section Multiple file systems).

Logging

One of the key requirement for running automated processes in the background is proper logging. Each process should create its own log file and a certain number of log files should be kept. Log files are typically node local, i.e. each node writes its own log file.

The assignment of the log file name can be integrated to the control component because it also invokes the storage service script and can redirect its output to the proper log files. Version management can either be performed by the control component as shown in the example below or by leveraging the '*logrotate*' service of the Linux operating system (see section Log rotation in Linux).

The following snippet shows the assignment of a log file name according to the storage service names (see Assigning storage service) and the current date and manages the versioning:

```
#
# The following snippet is part of the script launcher.sh
#---------------------------------------------------------------------
#
# directory for log files
logDir="/var/adm/ras/storageservice"

# number of log file versions to be kept, including this run
verKeep=3

# additional log file versions to be kept in compressed format
verComp=3

# current date will be part of the log file name
curDate="$(date +%Y%m%d%H%M%S)"

# assing logfile name
logF=$logDir"/"$op"_"$curDate".log"

# delete and compress older  logfiles prior to logging anything
lFiles=$(ls -r $logDir/$op*)
i=1
#echo "DEBUG: files=$lFiles"
for f in $lFiles;
do
  if (( i > verKeep ));
  then
    if (( i > (verComp+verKeep) ));
    then
      rm -f $f >> $logF 2>&1
    else
      gzip $f >> $logF 2>&1
    fi
  fi
  (( i=i+1 ))
```

```
done

# … perform the next steps …
```

At the end of this example there is a unique log file according to the storage service process and previous log files are managed according to their age. More precisely, this code will keep 6 log files (including the one for the current process), the 3 oldest are compressed. As mentioned before the log files are node local, i.e. each node writes its own log files.

Running storage service

When the control component has determined that this node is cluster manager and that it is active with the file system mounted it has assigned the storage services script and log file name and can now run the operation:

```
#
# The following snippet is part of the script launcher.sh
#----------------------------------------------------------------------
#
# run the script and log stdout and errout to file $logF
eval $cmd >> $logF 2>&1
rc=$?

# analyze return code and notify the admin when required…
```

The assigned storage script (`$cmd`, see section Assigning storage service) is run by the eval-command and the return code is checked. In addition the output of the script is logged to a log file (`$logF`).

Important hint: Certain operations such as backup and migration require certain software to be installed on the node where this operation is started. For example a node running `mmbackup` requires the B/A client to be installed and configured. Likewise the `mmapplypolicy` command for hierarchical storage management with IBM Spectrum Protect or IBM Spectrum Archive requires the HSM client to be installed on the node running this command. If the control component starts the storage service on the cluster manager then each quorum node must have the appropriate software installed. This is because each quorum node can become cluster manager. Alternatively, the control component can delegate running the storage service on a different node (see section Selecting different node).

Event Notification based on return codes

Because the control component launches the storage services scripts (e.g. backup.sh or migrate.sh) it is important to provide consistent return codes within these scripts. The control component can evaluate the consistent return codes and trigger appropriate alerts. To keep it, simple 3 return codes are defined and exported, as shown in the example below:

```
#
# The following snippet is part of the script launcher.sh
#----------------------------------------------------------------------
```

```
#
#define return codes
export rcGood=0  # successful run
export rcWarn=1  # run was ok, some warnings however
export rcErr=2   # failed
```

Exporting the return code definitions makes these available to the storage services scripts started by the control component. These script can now use consistent return codes.

Based on the return code definition alerts can be issued based upon Warnings (return code 1) and errors (return code 2). There are several ways to issue alerts such as sending an email to the administrator or creating a custom event that it surfaced by the Spectrum Scale event monitor and the GUI (see section Raising custom events)

Raising custom events

With Spectrum Scale version 4.2.3 and above there is a way to define and raise custom events. Custom events have to be defined on each node that can become cluster manager (all quorum nodes, see section Node selection) and on all GUI nodes. The nodes that can become cluster manager run the launcher script (`launcher.sh`) which will raise events when required. The GUI nodes need to know about all defined events regardless if they run the launcher script or not.

The event definitions are stored in JSON files in directory `/usr/lpp/mmfs/lib/mmsysmon/`. There is one JSON file for each component. In order to define custom event a new file has to be create and name `custom.json` in this path.

Attention: Do not edit any other event definition file.

On all quorum and GUI nodes create the file `/usr/lpp/mmfs/lib/mmsysmon/custom.json` with the following content:

```
{
"automation_warning":{
  "cause":"An automated process ended with warnings.",
  "user_action":"Check the log files to determine the root cause. Run
the process again.",
  "entity_type":"FILESYSTEM",
  "scope":"NODE",
  "code":"888332",
  "description":"An automated process ended with warnings, check the
message section.",
  "event_type":"INFO_EXTERNAL",
  "message":"Process {1} for file system {0} ended with WARNINGS on
node {2}. See log-file {3}. Run this process again and observe the
results.",
  "severity":"WARNING",
  "require_unique": true
},
"automation_error":{
  "cause":"An automation process failed.",
  "user_action":"Check the log files to determine the root cause.
Correct the problem and run the process again.",
  "entity_type":"FILESYSTEM",
  "scope":"NODE",
  "code":"888333",
  "description":"An automation failed, check the message section.",
  "event_type":"INFO_EXTERNAL",
  "message":"Process {1} for file system {0} ended with ERRORS on node
{2}. See log-file {3} and determine the root cause before running the
process again",
  "severity":"ERROR",
  "require_unique": true
}
}
```

The first event definition is for warning events. The event code for this event is 888332. The second event definition is for an error event and has event code 888333.

Attention: ensure that the event code you are using is not used by any other event in any of the other event definition files.

In order to prevent this file to be removed during a Spectrum Scale software update it is recommended to move this file to a persistent path (e.g. `/var/mmfs/`) and create a logical link outside of this directory:

```
# ln -s /var/mmfs/custom.json /usr/lpp/mmfs/lib/mmsysmon/custom.json
```

Subsequently restart the system monitor on each node where the new event definition file was created:

```
# mmsysmoncontrol restart
```

And restart the GUI on all nodes running the GUI:

```
# systemctl restart gui
```

Once the custom event definition has been created, test sending events on all quorum nodes that can potentially become cluster manager and run the launcher script. According to the example provided above, there are two events that can be sent. The first event definition with event code 888332 is for warning events. The second event with code 888333 is for error messages.

The event message defines parameters that can be given during runtime. For example, the following message includes 4 parameters:

```
"message":"Process {1} for file system {0} ended with ERRORS on node
{2}. See log-file {3} and determine the root cause before running the
process again"
```

The first parameter {0} is the file system name. The second parameter {1} is the name of the process. The third parameter {2} is the node name and the fourth parameter {3} is the name of the log file. These parameters can be set when the event is raised using the command:

```
# mmsysmonc event filesystem code fsname
“fsname,processname,nodename,logfilename”
```

For example to send an event for file system gpfs1 signaling a WARNING message during backup on node name node1 with the log file being backup.log use the following command:

```
# mmsysmonc event filesystem 888332 gpfs1
"gpfs1,backup,node1,backup.log"
```

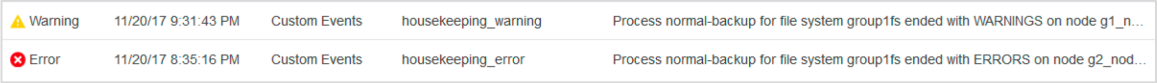
Likewise in order to raise an ERROR event for the above process use the command:

```
# mmsysmonc event filesystem 888332 gpfs1
"gpfs1,backup,node1,backup.log"
```

To check the events use the following command for the node where the event has been raised. It might take a couple of minutes before the event is surfaced in the node event log:

```
# mmhealth node eventlog --hour [-N nodename]
```

Check the event log in the GUI for the event. It might take a couple of minutes until the event is surfaced in the GUI. An event in the GUI may look like this:



⚠ Warning	11/20/17 9:31:43 PM	Custom Events	housekeeping_warning	Process normal-backup for file system group1fs ended with WARNINGS on node g1_n...
⊗ Error	11/20/17 8:35:16 PM	Custom Events	housekeeping_error	Process normal-backup for file system group1fs ended with ERRORS on node g2_nod...

In order to send custom events as event notification via email or SNMP the *custom* event type has to be selected in the event notification setup dialog of the Spectrum Scale GUI.

Now, raising custom events can be integrated into the control component, namely the launcher script (`launcher.sh`), as shown in the following example:

```
#
# The following snippet is part of the script launcher.sh
#---------------------------------------------------------------------
#
# run the script and log stdout and errout to file $logF
eval $cmd >> $logF 2>&1
rc=$?

# send event notification based on the definition in file
/usr/lpp/mmfs/bin/mmsysmon/custom.json
echo "INFO: Raising appropriate Event for return code $rc" >> $logF
if (( rc == 1 ));
then
  # send warning event
  mmsysmonc event filesystem 888332 $fsName
"$fsName,$op,$localNode,$logF" >> $logF
elif (( rc > 1 ));
then
  # send error event
  mmsysmonc event filesystem 888333 $fsName
"$fsName,$op,$localNode,$logF" >> $logF
fi

exit $rc
```

Backup component

The backup component is represented by a script (`backup.sh`) that performs backup using the `mmbackup`-command [1]. This script is invoked by the control component (see sections Assigning storage service and Running storage service).

The node running the `mmbackup`-command must be a backup node with IBM Spectrum Protect Backup / Archive client installed and configured. Since the backup component is invoked by the node selection component which must run on a manager node, the B/A client has to be installed and configured on all manager nodes. Alternatively the node selection component can select another node for running the backup (see section Selecting different node)

Backup can be performed for file systems or file sets. Furthermore, it can be performed from snapshots. In the example below the backup for a pre-defined file system is being performed using the `mmbackup`-command:

```
#
# The following snippet is part of the script backup.sh
#----------------------------------------------------------------------
#
```

```
# file system name
fsName="myfs"

# name of TSM server
tsmServ="myTSM"

# directory for temp files for policies and mmbackup
workDir="/myFs/.mmbackupTmp"

**************************  MAIN ***************************

#start mmbackup
echo "$(date) BACKUP: Starting mmbackup"

mmbackup $fsName --tsm-servers $tsmServ -N nsdNodes -v --max-backup-
count 4096 --max-backup-size 80M --backup-threads 1 --expire-threads 1
-s $workDir
rc=$?

# exit according to the return code definition
echo "$(date) BACKUP: operation ended on $(hostname) with rc=$rc."
if (( rc == 0 ));
then
  exit $rcGood
elif (( rc == 1 ));
then
  exit $rcWarn
else
  exit $rcErr
fi
```

The script example above writes messages to STDOUT. Because this script is started by the control component with all output redirected to the log file, all messages will appear in the log file initialized by the control component (see section Logging).

This script also leverages the return codes exported by the control component (see section Return codes) by ending with a pre-defined return code. This return code can be used by the control component to trigger an alert.

The mmbackup command example above includes a number of parameters that have to be adjusted. The name of the file system ($fsName) and the TSM server (--tsmServers $tsmServ) are predefined at the beginning. The nodes executing the backup operation are all NSD server (-N nsdNodes). The command also uses a special directory for temporary files (parameter -s $workDir) that is located in the file system to be backed up. This directory must exist. Using a special directory is recommended if there is a large number of files in the file system to be inspected and to be backed up.

To support multiple file systems the file system name ($fsName) can be passed to the backup components as an additional parameter (see section Multiple file systems).

Note, mmbackup should be run with the -q options periodically in order to keep the shadow file in sync. For example, to run the backup with the –q option approximately every Sunday the following logic can be implemented:

```
#
# The following snippet is part of the script backup.sh
#---------------------------------------------------------------------
#
#run mmbackup with -q every other day
rebuild=""
d=$(date +%u)
if (( d == 7 ));
then
   rebuild="-q"
fi

# … add $rebuild to the mmbackup command …
```

For the latest information about the backup components, refer to the readme file: https://github.com/IBM/SpectrumScaleAutomation

Storage tiering component

Storage tiering allows moving data from one file system storage pool to another. A file system storage pool can be represented by disk (internal pool) or by tape storage (external pool). A tape storage pool can be implemented with IBM Spectrum Archive (LTFS EE) or IBM Spectrum Protect for Space Management (TSM HSM).

While storage pool migration can be triggered by thresholds encoded in an active policy, it is recommended to run scheduled migration jobs periodically. Such scheduled migration jobs can be automated with this framework.

In this example we elaborate on pre-migration and migration from an internal system pool to the Spectrum Protect server using TSM HSM. The migration component is represented by a script (`migrate.sh`, see section Migration process) that performs migration according to a pre-defined policy using the Spectrum Scale command `mmapplypolicy`. The pre-migration component is represented by a script (`premigrate.sh`, see section Pre-migration process) that performs pre-migration of all files in the subject file system. These scripts are invoked by the control component (see sections Assigning storage service and Running storage service). Migration and pre-migration is initiated by the `mmapplypolicy`-command [2].

The node running the `mmapplypolicy`-command must be a HSM node with HSM client installed and configured. Since the storage tiering component is invoked by the node selection component which must run on a manager node, the HSM client has to be installed and configured on all manager nodes. Alternatively the node selection component can select another node for running the backup (see section Selecting different node)

For the latest information about the migration components refer to the readme file: https://github.com/IBM/SpectrumScaleAutomation

Migration process

The migration process migrates files from the system pool of the subject file system to the Spectrum Protect server that are older than 30 days. The migration policy looks like this:

```
/* define macros */
define(is_migrated, (MISC_ATTRIBUTES LIKE '%V%'))
define(is_empty,(FILE_SIZE=0))
define(access_age_days, (DAYS(CURRENT_TIMESTAMP) - DAYS(ACCESS_TIME)))

/* define exclude rule */
RULE 'exclude' EXCLUDE WHERE (PATH_NAME LIKE '%/.SpaceMan/%'
                 OR NAME LIKE '%dsmerror.log%'
                 OR NAME LIKE '%.mmbackup%'
                 OR NAME LIKE '%mmbackup%'
                 OR PATH_NAME LIKE '%/.snapshots/%'
                 OR PATH_NAME LIKE '%/.mmbackupTmp/%')

/* Define hsm storage manager as an external pool */
RULE EXTERNAL POOL 'hsm' EXEC '/var/mmfs/etc/mmpolicyExec-hsm.mig' OPTS
'-v'

/* Rule migrate */
RULE 'MigrateData' MIGRATE FROM POOL 'system' TO POOL 'hsm'
  WHERE (access_age_days > 30) AND NOT (is_migrated) AND NOT (is_empty)
```

Certain files and directories are excluded using EXCLUDE rules. Please note the external pool script is named `/var/mmfs/etc/mmpolicyExec-hsm.mig`. This is an exact copy of the sample script located in `/usr/lpp/mmfs/samples/ilm/mmpolicyExec-hsm.sample`. This new instance of the script is used to distinguish from the external pool script used for pre-migration. Also note in the policy above that files with a size of 0 are not selected for migration, because typically these files generate error messages during migration.

The policy above is stored in file `policy_mig.txt`. This policy is invoked by the migration script (`migrate.sh`) shown below:

```
#
# The following snippet is part of the script migrate.sh
#---------------------------------------------------------------------
#
# file system name
fsName="myfs"

# policy file name for premigrate
polName="myfs/.work/scripts/policy_mig.txt"

# directory for temp files for policies and mmbackup
workDir="/myfs/.mmbackupTmp"

#************************* Main ***************************
#start selective migration
echo "$(date) MIGRATE: Starting selective migration"
mmapplypolicy $fsName -P $polName -N nsdNodes -m 1 -B 256 -s $workDir
rc=$?

# exit according to the return code definition
echo "$(date) MIGRATE: operation ended on $(hostname) with rc=$rc."
if (( rc == 0 ));
then
  exit $rcGood
elif (( rc == 1 ));
then
  exit $rcWarn
else
  exit $rcErr
fi
```

The script example above writes messages to STDOUT. Because this script is started by the control component with all output redirected to the log file, all messages will appear in the log file initialized by the control component (see section Logging).

This script also leverages the return codes exported by the control component (see section Return codes) by exiting with a pre-defined return code. This return code can be used by the control component to trigger an alert.

The `mmapplypolicy`-command example above uses a number of parameters that have to be adjusted. The file system name (`$fsName`) and the name of the policy-file are defined at the beginning. The nodes executing the migrate operation are all NSD servers (`-N nsdNodes`). The command also uses a special directory for temporary files (parameter `-s $workDir`) that is located in the file system to be backed up. This directory must exist. Using a special directory is recommended if there is a large number of files in the file system to be inspected and to be migrated. Note, the TSM server name cannot be given with the `mmapplypolicy`-command. It is assumed that either there is only one TSM server used with its name defined in the dsm.opt file or that the name of the TSM server has been given when the file system has been added for space management (`dsmmigfs add -Server=<tsm-server>`).

To support multiple file systems the file system name (`$fsName`) can be passed to the migration components as an additional parameter (see section Multiple file systems).

Pre-migration process

In contrast to migration, pre-migration creates a copy of files in the Spectrum Protect server. Thus a pre-migrated file is dual resident, in the internal file system pool and in TSM HSM.

In this example the goal of the pre-migration is to pre-migrate all files from the pool *system* to TSM HSM. The following policy can accomplish this:

```
/* define macros */
define(is_resident,(MISC_ATTRIBUTES NOT LIKE '%M%'))
define(is_empty,(FILE_SIZE=0))

/* Define the exclude list and migrated / premigrated */
RULE 'exclude' EXCLUDE WHERE (PATH_NAME LIKE '%/.SpaceMan/%'
                  OR NAME LIKE '%dsmerror.log%'
                  OR NAME LIKE '%.mmbackup%'
                  OR NAME LIKE '%mmbackup%'
                  OR PATH_NAME LIKE '%/.snapshots/%'
                  OR PATH_NAME LIKE '%/.mmbackupTmp/%')

/* Define hsm storage manager as an external pool */
RULE EXTERNAL POOL 'hsm' EXEC '/var/mmfs/etc/mmpolicyExec-hsm.pmig'
OPTS '-v'

/* Rule Premigrate all */
RULE 'PremigrateData' MIGRATE FROM POOL 'system' TO POOL 'hsm'
  WHERE (is_resident) AND NOT (is_empty)
```

Please note the external pool script is named `/var/mmfs/etc/mmpolicyExec-hsm.pmig`. This is an adjusted version of the sample script located in `/usr/lpp/mmfs/samples/ilm/mmpolicyExec-hsm.sample`. The adjustment in this script ensures that files are pre-migrated and comprises one change in the source script:

```
# insert the parameter -premigrate to the $MigrateFormat
  $MigrateFormat = "%s %s -premigrate -filelist=%s";
```

With this adjustment any invocation of this external pool script with a migration rule will pre-migrate the files. Note, this script has to be adjusted all nodes running TM HSM.

The policy above is stored in file `policy_pmig.txt`. This policy is invoked by the pre-migration script (`premigrate.sh`) shown below:

```
#
# The following snippet is part of the script launcher.sh
#---------------------------------------------------------------------
```

```
#
# file system name
fsName="myfs"

# policy file name for premigrate
polName="myfs/.work/scripts/policy_pmig.txt"

# directory for temp files for policies
workDir="/myfs/.mmbackupTmp"

#************************* Main ***************************
#start pre-migration
echo "$(date) PREMIGRATE: Starting pre-migration"
mmapplypolicy $fsName -P $polName -N nsdNodes -m 1 -B 256 -s $workDir
rc=$?

# exit according to the return code definition
echo "$(date) PREMIGRATE: operation ended on $(hostname) with rc=$rc."
if (( rc == 0 ));
then
  exit $rcGood
elif (( rc == 1 ));
then
  exit $rcWarn
else
  exit $rcErr
fi
```

The script example above writes messages to STDOUT. Because this script is started by the control component with all output redirected to the log file, all messages will appear in the log file initialized by the control component (see section Logging).

The parameters given with the `mmapplypolicy` command in this example require some adjustment, see section Migration process. To support multiple file systems the file system name (`$fsName`) can be passed to the backup components as an additional parameter (see section Multiple file systems).

This script also leverages the return codes exported by the control component (see section Return codes) by ending with a pre-defined return code. This return code can be used by the control component to trigger an alert.

Scheduling

There are different methods to schedule storage services operations within a Spectrum Scale cluster:

- Using the operating system scheduler (see section Using cron and Using systemd Timers)
- Using the Spectrum Protect server scheduler (see section Using Spectrum Protect)

The framework requires that all cluster nodes with a manager role assigned must be able to run the automation components. This is because only the node that is cluster manager initiates the operation. These nodes must not necessarily be the nodes performing the storage service operation (such as backup or migration).

Using cron

Cron is the traditional Unix / Linux daemon responsible for scheduling tasks in a system. It is controlled by one or multiple configuration files (crontab), and can be used to schedule storage service operations within a cluster. This requires that all automation components are installed on those cluster nodes that participate in the operation.

The participating nodes must be all nodes with a manager role defined, because potentially any of these manager-nodes can become cluster manager. The participating nodes must not necessarily be the nodes performing the storage management operation such as backup. Performing nodes are denoted in the `mmbackup` or `mmapplypolicy`-command via the `-N` parameter. In addition, the crontab entry must be configured identically on all participating nodes.

The automation components can be installed on the local disk of all participating nodes or in a Spectrum Scale file system that is accessible by all nodes. The latter approach has the advantage that there is one version of all automation components to be maintained and all nodes have instant access to it. The disadvantage is that if this file system is not available no node can actually run the scripts.

The crontab entry essentially invokes the control component at the defined time with the according parameter. The semantics of the crontab configuration file is the following [3]:

```
# ┌───────────── min (0 - 59)
# │ ┌───────────── hour (0 - 23)
# │ │ ┌───────────── day of month (1 - 31)
# │ │ │ ┌───────────── month (1 - 12)
# │ │ │ │ ┌───────────── day of week (0 - 6) (0 to 6 are Sunday to
# │ │ │ │ │              Saturday, or use names; 7 is also Sunday)
# │ │ │ │ │
# │ │ │ │ │
# * * * * *  command to execute
```

The example below starts backup at 6 AM in the morning:

```
PATH=/usr/bin:/usr/sbin:/usr/lpp/mmfs/bin

00 06 00 00 00  /path-to-scripts/launcher.sh backup
```

This crontab entry is configured on all participating nodes which causes the control component (`launcher.sh`) to be started at 6 AM on each of these nodes with the parameter “backup”. At first each node will check if it is the cluster manager (see Node selection) and since there is only one cluster manager at any given point in time only one node will continue to run the `launcher`-script.

Note, if Spectrum Scale sudo wrappers are activated then the launcher script must be started by a non-root user who has sudo permissions to administer the Spectrum Scale cluster. A simple change in the crontab entry above can accommodate this:

```
PATH=/usr/bin:/usr/sbin:/usr/lpp/mmfs/bin

00 06 00 00 00  gpfsadmin /bin/sudo /path-to-scripts/launcher.sh backup
```

This entry assumes that the user *gpfsadmin* has permissions to administer the cluster using sudo wrapper.

Using systemd Timers

A modern alternative to the traditional cron daemon are *systemd* timer units. As most Linux distributions have transitioned to the *systemd* system and service manager, tasks can be scheduled and automated within this framework. Since *systemd* keeps track of running services and events it maintains a very detailed state of the entire system including network connectivity and mounted file systems. This allows for starting certain tasks when certain conditions are met, such as when certain filesystems are mounted.

The system timer unit can be used to start the control components, find below a simple example for such unit definition:

```
[Unit]
Description=A simple launcher to automate backup tasks in the cluster
After=gpfs.service

[Service]
ExecStart=/path-to-scripts/launcher.sh backup
```

This unit definition is stored in a file within the system unit directory, such as `/etc/systemd/system/launcher.service`. It can be manually started for testing with the command `systemctl start launcher`. Because *systemd* tracks the state and return code of the tasks it starts the status of the launcher task can be queried with the command: `systemctl status launcher`. The line ‘`After=`’ in the unit definition above ensures that the control component is only run after Spectrum Scale was started.

To automatically run the control component encapsulated in a systemd unit “launcher” at predefined time intervals a timer unit file can be created in file `/etc/systemd/system/launcher.timer`:

```
[Unit]
Description=A simple launcher to automate backup tasks in the cluster

[Timer]
OnCalendar=23:00
OnUnitActiveSec=24h
Persistent=true
Unit=launcher.service
```

```
[Install]
WantedBy=timers.target
```

After enabling this timer with `systemctl enable launcher.timer` and starting it with `systemctl start launcher.timer` you can verify when the timer has last run, when it will run the next time, and what the result of the last run was with the command `systemctl list-timers`. The above example will run the launcher task at 23:00h every day, or after 24 hours if this moment was missed. Note that systemd timers are flexible and can be scheduled in many different ways - e.g. after certain events or after another task has ended - the above is only a minimal example [4].

Using Spectrum Protect

The Spectrum Protect server scheduler in combination with client schedules can be used to schedule the storage services operation, leveraging the automation components presented above.

Important, this requires that all nodes with the manager role must have installed the TSM client and enabled the client acceptor daemon. This is because the control component will only execute the operation on the cluster manager at the time.

As mentioned before the automation components must be installed on all cluster nodes with manager role. In addition the TSM client acceptor daemon must be configure on of these nodes. The nodes executing the operations can be different and are specified with the –N parameter in the mmbackup- and mmapplypolicy command of the appropriate component.

Important, in the control component (launcher.sh) the necessary PATH environment must be exported:

```
EXPORT PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/lpp/mmfs/bin
```

First install the TSM backup client on all manager nodes (and others when required) and start the client acceptor daemon.

Now you can define a server schedule for the backup process. Use the parameters Action=command and Objects=<script-path-name>. The following example schedules the backup process every day at 6 AM:

```
DEFine SCHedule domain_name gpfs_backup  Type=Client ACTion=Command
 OBJects='/path-to-scripts/launcher.sh backup'
 STARTDate=today
 STARTTime=06:00:00
 Active=yes
```

Now you have to associate the client schedule with the Spectrum Scale nodes that have manager role and the TSM acceptor daemon installed. Use the real TSM node names not the proxy nodes. For example, if you have three nodes in your cluster (g1_node1, g1_node2, g1_node3) that are bound to one proxy-node (g1_proxy) in domain g1_dom, use the normal node names:

```
DEFine ASSOCiation g1_dom gpfs_backup g1_node1, g1_node2, g1_node3
```

Finally check if the schedule works properly.

Further considerations

This section summarizes some further consideration taking into account that multiple file systems might have to be processed (see section Multiple file systems). It also elaborates on the use of the log-rotation function in Linux (see section Log rotation in Linux)

Multiple file systems

In the example of the automation framework the file system name is hard-coded in the scripts. This framework can be extended to allow the storage service operation for different file systems by not hard-coding the file system name but passing the file system name as parameter to the control component. Find below some basic guidance how to enable this.

The control component (`launcher.sh`) is extended to require a second parameter which must be the file system name. The control component is invoked like so:

```
# launcher.sh <operation> <filesystem>
```

Within the control component (`launcher.sh`)the file system is assigned to the parameter `$fsName`:

```
#
# The following snippet can be part of the script launcher.sh
#----------------------------------------------------------------------
#

fsName=$2
if [[ -z $fsName ]];
then
   echo "ERROR: file system name not specified, exiting"
   exit $rcErr
else
  export $fsName
fi
```

The file system name is later used in the control component to check the file system status. If the file system was miss-spelled then the script will terminate. Furthermore, the file system name is exported and can be used by scripts that are called by the control component (e.g. `backup.sh`).

Alternatively, the file system name can be explicitly passed to the other components. This allows for explicitly running these scripts from the command line for testing purposes without the control component. Here is an example to accomplish this according to the example in section Assigning storage service:

```
#
# The following snippet is part of the script launcher.sh
#----------------------------------------------------------------------
```

```
#
# file system name has been assigned before

# assign parameter, check parameter and assign script
scriptPath="~/myscripts"
op=$1

case $op in
"backup")
  # runs Backup
  cmd=$scriptPath"/backup.sh $fsName”;;
“migrate")
  # runs migration (scheduled)
  cmd=$scriptPath"/migrate.sh $fsName”;;
“premigrate")
  # runs premigration (scheduled)
  cmd=$scriptPath"/premigrate.sh $fsName”;;
*) echo "ERROR: wrong operation code: $op"
   echo "Syntax: $0 <operation> <filesystem-name>"
   echo "Operation can be: backup, migrate or premigrate"
   exit 2;;
esac
```

As shown above the script performing the storage service operation will be invoked with the file system name. Consequently these scripts (`backup.sh`, `migrate.sh` and `premigrate.sh`) must be adjusted to use this parameter. The adjustment is done at the beginning of these scripts where the parameter `fsName` is defined:

```
fsName=$2
if [[ -z $fsName ]];
then
   echo “ERROR: file system name not specified, exiting”
   exit $rcErr
else
  export $fsName
fi
```

With this approach multiple file systems can be scheduled for storage services operations such as backup and migration.

Log rotation in Linux

In section Logging we discussed a simple version management of log files that can be easily integrated into the control component script. However, the Linux operating system has a flexible framework for this exact purpose: logrotate. The logrotate service provides various options for managing log files by means of automatic rotation, compression and email delivery.

The logrotate function is typically controlled with the configuration files `/etc/logrotate.conf` or with supplemental configuration files stored in `/etc/logrotate.d/`. A minimal sample configuration file could be implemented like this:

```
/path-to-logfiles/*.log {
  daily
  rotate 6
  compress
}
```

Store this configuration in file `/etc/logrotate.conf` or create a supplemental configuration file such as `/etc/logrotate.d/launcher.rotate`. This configuration will rotate the log files daily and keep 6 versions of the files matching the path and file name pattern `/path-to-logfiles/*.log`. Each log file will get a timestamp added to the file name, and will be compressed using gzip.

With this mechanism it is also possible to keep more or less log files for certain operations by adjusting the log file name. For example the log files for the backup operations may be named backup* and log files for the migrate operation may be named migrate*. So using the syntax above there could be more version of backup log files (by using the filename `/path-to-logfiles/backup*.log`) than migrate log files (by using the filename `/path-to-logfiles/migrate*.log`). There are plenty of additional options available [5].

Logrotate is invoked with the system scheduler such as cron. Typically logrotate is preconfigured in cron to run daily. Check the cron configuration (directory `/etc/cron.daily`) and when required add logrotate to run daily.

Selecting different node

The control component (`launcher.sh`) starts the appropriate storage service script (such as `backup.sh` or `migrate.sh`) on the local node on which it runs. This means that all Spectrum Scale nodes with quorum (and manager) role are candidate for being selected as launcher node, and that all these nodes need to be able to run e.g. backup and migration tasks (see section Node selection). As a consequence, all quorum nodes may need a backup client package installed – which might not always be appropriate. For this reason, we present an alternative approach for running the storage service on a different node than the launcher node itself. In the example below we focus on the backup task.

Let’s assume a cluster with dedicated nodes running the backup process (`mmbackup`). The first step is to create a node class and add all nodes to this class which can run the backup task (i.e. have the TSM backup client installed and configured):

Check the defined node classes:

```
# mmlsnodeclass --all
```

Create a new node class *backupClients* and add the nodes that have the backup client installed and that can run the backup task:

```
# mmcrnodeclass backupClients -N nodenames
```

Next, the control component needs to be adjusted to select an active node from this node class for running the backup task. The node selected must be active and must have the file system mounted. The local node is preferred, when it is part of the node class. If no node class is defined, the storage service is run on the local node.

```
#
# The following snipped is an extended part of launcher.sh
# ---------------------------------------------------------------
#
# select an active node of a node class
#
# determine the node to run this command based on node class
# if the local node is part of the node class, prefer this
nodeClass=backupClients
localNode=$(mmlsnode -N localhost | cut -d'.' -f1)
allNodes=""
sortNodes=""
# if node class is set up determine node names in node class
if [[ ! -z $nodeClass ]];
then
   allNodes=$(mmlsnodeclass $nodeClass -Y | grep -v HEADER | cut -d':'
-f 10 | sed 's|,| |g')
   if [[ -z $allNodes ]]
   then
     echo "CHECK: WARNING node class $nodeClass is empty, using local
node" >> $logF
     sortNodes=$localNode
   else
     # reorder allNodes to have localNode first, if it exists
     for n in $allNodes;
     do
       if [[ "$n" == "$localNode" ]];
       then
         sortNodes=$localNode" "$sortNodes
       else
         sortNodes=$sortNodes" "$n
       fi
     done
   fi
else
   # if no node class is defined set the local node
   sortNodes=$localNode
fi

# select the node to execute the command based on node and file system
states
echo "INFO: The following nodes are checked to run the operation:
$sortNodes" >> $logF
execNode=""
for n in $sortNodes;
do
  # determine node state
  state=$(mmgetstate -N $n -Y | grep -v ":HEADER:" | cut -d':' -f 9)
  if [[ "$state" == "active" ]];
  then
    # determine file system state on node
    mNodes=$(mmlsmount $fsName -Y | grep -v HEADER | grep -E ":RW:" |
cut -d':' -f 12)
    for m in $mNodes;
    do
      if [[ "$m" == "$n" ]];
      then
```

```
        execNode=$m
      fi
    done
    # if we found a node then leave the loop
    if [[ ! -z "$execNode" ]];
    then
      break
    fi
  fi
done

# if we have not found a node that qualifies for the job then exit
if [[ -z "$execNode" ]];
then
  echo "$(date) CHECK: ERROR no node is in appropriate state to run the
job, exiting." >> $logF
  exit $rcErr
fi
```

Finally, the control component uses ssh to run the storage service on another node. This requires that the node running the control component is an admin node (has shared its ssh-key with all other nodes):

```
#
# The following snippet is extended part of the script launcher.sh
#---------------------------------------------------------------------
#
# run the script on another node and log stdout and errout to $logF
ssh $selNode $cmd &> $logF
rc=$?

# analyze return code and notify the admin when required…
```

Every time a new node performing the backup function is added to the cluster it must be added to the dedicated node class (in this example backupClients).

Appendix

References

[1] mmbackup command reference for Spectrum Scale version 4.2.1
https://www.ibm.com/support/knowledgecenter/STXKQY_5.0.0/com.ibm.spectrum.scale.v5r00.doc/bl1adm_mmbackup.htm

[2] mmapplypolicy command reference for Spectrum Scale version 4.2.1
https://www.ibm.com/support/knowledgecenter/STXKQY_5.0.0/com.ibm.spectrum.scale.v5r00.doc/bl1adm_mmapplypolicy.htm

[3] Explanation of cron
https://en.wikipedia.org/wiki/Cron

[4] Explanation of systemd Timers
https://www.freedesktop.org/software/systemd/man/systemd.timer.html

[5] logrotate - rotates, compresses, and mails system logs
https://linux.die.net/man/8/logrotate

[6] Link to open source repository (may only be accessible for IBM):
https://github.com/IBM/SpectrumScaleAutomation

Disclaimer

This document reflects the understanding of the author in regard to questions asked about archiving solutions with IBM hardware and software. This document is presented "As-Is" and IBM does not assume responsibility for the statements expressed herein. It reflects the opinions of the author. These opinions are based on several years of joint work with the IBM Systems group. If you have questions about the contents of this document, please direct them to the Author (nils_haustein@de.ibm.com).

The Techdocs information, tools and documentation ("Materials") are being provided to IBM Business Partners to assist them with customer installations. Such Materials are provided by IBM on an "as-is" basis. IBM makes no representations or warranties regarding these Materials and does not provide any guarantee or assurance that the use of such Materials will result in a successful customer installation. These Materials may only be used by authorized IBM Business Partners for installation of IBM products and otherwise in compliance with the IBM Business Partner Agreement."

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both: IBM, IBM Spectrum Scale and IBM Spectrum Protect.

Linux is a registered trademark of Linus Torvalds

Other company, product, and service names may be trademarks or service marks of others.