# Introduction

MarFS is a complex product, providing a near-POSIX (Portable Operating System Interface) mechanism to a file system that scalably managed metadata and data. The initial implementation will use one or more GPFS (General Parallel File System, from IBM) file systems as the metadata store, providing the scalable near-POSIX mechanism and one or more Scality object stores as the data store, providing scalable data repository.

A comprehensive test plan is necessary to assure ourselves that it is both functional and performing. We will employ both Black Box and White Box testing techniques. Black Box testing makes no assumptions about the internals of the implementation. It ensures that things do and perform as the end-user expects. White Box testing knows about the internal implementation and will check to make sure that things are done as the implementation intends in order to provide the required functionality and performance. Each test will discuss what it will require as success criteria for both Black Box and White Box testing.

The logical components of MarFS are the FUSE daemon running on the interactive FTA(s) (File Transfer Agents), pftool running on the batch FTA(s), and the utility programs for managing and maintaining MarFS.

# Unit Testing

Of greatest concern to us is to establish that the things we expect users to do to the file system work correctly and with sufficient performance. At a later date we may consider deepening the test capabilities with unit testing.

You can't test e.g. readdir() without testing opendir(). So, we need a set of simple flows that exercise all the functions, but do each one by means of a minimal path. For example, if we want to test readdir(), we shouldn't need llistxattrs() to work. We must know the graph of dependencies to run a particular test. We find the dependencies and create tests to try them, e.g. for readdir() we only need opendir().

Of course, we'd want an option to "test everything". This would provide regression tests to ensure sure nothing else breaks when an unrelated change is made.

An initial implementation could be a simple hand-coded test with a "switch" to pick from a series of minimal-flow tests. Give it a number on the command-line, and it runs that test. Give it 0 and it runs all of them. An enhancement would enable selecting classes of tests, such as "xattr tests", "directory tests", "file tests", "write tests", etc.

Make sure return codes for failures are appropriate.

# MarFS FUSE Daemon & PFTool

The MarFS FUSE daemon will only run on interactive FTAs. Users login to these FTAs to either do manual serial file moves between mounted file systems or to launch jobs onto the batch FTAs, which will be covered when we discuss pftool.

Since the MarFS FUSE daemon is the interactive FTA interface to MarFS, we expect the operations to be rather simple operations. This test plan will reflect that expectation.

MarFS PFTool will only run on batch FTAs. Users login to an interactive FTA and launch the PFTool script commands: pfcp, pfcm, and pfls; from the interactive FTA to be run on the batch FTAs.

All file system references will be representative. For a given installation the tester will need to translate these representative file system references to the actual ones used on the system being tested. The representative references will be:

* /marfs: The MarFS file system exported by the MarFS FUSE daemon.
* /mds: The GPFS file system used as the MarFS metadata file system.
* http://objstore.domain/: The RESTful designation of the object store used as the MarFS data repository.
* /otherfs: A non-MarFS file system that is locally mounted.

Here are some handy commands to use to inspect attributes on MarFS metadata files:

* attr -l lists the extended attributes of a file.
* attr -g <attr-name> displays the value of the extended attribute named for the file.

s3curl.pl is a handy utility for sending Amazon S3 commands to an object store. In order to use it, you must have a $HOME/.s3curl file that contains a "friendly-name" that specifies the object store user id that owns all MarFS objects and that user id's object access key. Here are some handy Amazon S3 commands to use to inspect MarFS objects:

* ./s3curl.pl --id=[friendly-name] -- http://objstore.domain/obj-name

## Copy Files from MarFS

This test will show that a user can copy a file on MarFS to some other file system. The test will include copying single files of all MarFS types and a directory tree of files. Scenarios will include copying a non-existent file and a non-existent directory tree.

All tests will require these steps:

1. Have files in /marfs that are known to be Uni, Packed, and Multi files.
2. Have a directory, dir-tree, on /marfs that is sufficiently sized to assure yourself that directory tree copying works
3. Login to an interactive FTA

### Copy a Single Uni File using FUSE daemon

1. % cp /marfs/fuse/uni-file /otherfs/fuse
2. % ls -l /otherfs/fuse
3. Should see uni-file in /otherfs/fuse
4. Verify that it is at least the same size as the source file on /marfs/fuse

### Copy a Single Uni File using PFTool

1. % pfcp /marfs/pftool/uni-file /otherfs/pftool
2. % pfls /otherfs/pftool
3. Should see uni-file in /otherfs/pftool
4. Verify that it is at least the same size as the source file on /marfs/pftool

### Copy a Single Packed File using FUSE daemon

1. % cp /marfs/fuse/packed-file /otherfs/fuse
2. % ls -l /otherfs/fuse
3. Should see packed-file in /otherfs/fuse
4. Verify that it is at least the same size as the source file on /marfs/fuse

### Copy a Single Packed File using PFTool

1. % pfcp /marfs/pftool/packed-file /otherfs/pftool
2. % pfls /otherfs/pftool
3. Should see packed-file in /otherfs/pftool
4. Verify that it is at least the same size as the source file on /marfs/pftool

### Copy a Single Multi File using FUSE daemon

1. % cp /marfs/fuse/multi-file /otherfs/fuse
2. % ls -l /otherfs/fuse
3. Should see multi-file in /otherfs/fuse
4. Verify that it is at least the same size as the source file on /marfs/fuse

### Copy a Single Multi File using PFTool

1. % pfcp /marfs/pftool/multi-file /otherfs/pftool
2. % pfls /otherfs/pftool
3. Should see multi-file in /otherfs/pftool
4. Verify that it is at least the same size as the source file on /marfs/pftool

### Copy a Directory Tree using FUSE daemon

1. % cp -r /marfs/fuse/dir-tree /otherfs/fuse
2. % ls -lR /otherfs/fuse/dir-tree
3. Should see dir-tree and all its subdirectories and files in /otherfs/fuse
4. Verify that the directory structure and file sizes are the same as the source directory on /marfs/fuse

### Copy a Directory Tree using PFTool

1. % pfcp -R /marfs/pftool/dir-tree /otherfs/pftool
2. % pfls -R /otherfs/pftool/dir-tree
3. Should see dir-tree and all its subdirectories and files in /otherfs/pftool
4. Verify that the directory structure and file sizes are the same as the source directory on /marfs/pftool

### Copy a Non-Existent File using FUSE daemon

1. % cp /marfs/fuse/non-existent-file /otherfs/fuse
2. Should get an error that non-existent-file does not exist.

### Copy a Non-Existent File using PFTool

1. % pfcp /marfs/pftool/non-existent-file /otherfs/pftool
2. Should get an error that non-existent-file does not exist.

### Copy a Non-Existent Directory Tree using FUSE daemon

1. % cp -r /marfs/fuse/non-existent-dir-tree /otherfs/fuse
2. Should get an error that non-existent-dir-tree does not exist.

### Copy a Non-Existent Directory Tree using PFTool

1. % pfcp -R /marfs/pftool/non-existent-dir-tree /otherfs/pftool
2. Should get an error that non-existent-dir-tree does not exist.

## Copy Files to MarFS

This test will show that a user can copy a file to MarFS from some other file system. The test will include copying single files of sizes that will yield all MarFS file types and a directory tree of files. Scenarios will include copying a non-existent file and a non-existent directory tree.

All tests will require these steps:

1. Have files in /otherfs that are known size to become Uni, Packed, and Multi files.
2. Have a directory, dir-tree, on /otherfs that is sufficiently sized to assure yourself that directory tree copying works
3. Login to an interactive FTA

### Copy a Single Uni File using FUSE daemon

1. % cp /otherfs/fuse/uni-file /marfs/fuse
2. % ls -l /marfs/fuse
3. Should see uni-file in /marfs/fuse
4. Verify that it is at least the same size as the source file on /otherfs/fuse
5. Get the attribute that tells what the object name is in the object store and verify that there is just one object
6. Use s3curl.pl to verify that the object exists and is the correct size.

### Copy a Single Uni File using PFTool

1. % pfcp /otherfs/pftool/uni-file /marfs/pftool
2. % pfls /marfs/pftool
3. Should see uni-file in /marfs/pftool
4. % pfcm -M /otherfs/pftool/uni-file /marfs/pftool/uni-file
5. Should see that the files compare to be the same.
6. Get the attribute that tells what the object name is in the object store and verify that there is just one object and it is designated an UNI file

### Copy a Single Packed File using FUSE daemon

Note: when using the MarFS FUSE daemon to write the file it will become a Uni File.

1. % cp /otherfs/fuse/packed-file /marfs/fuse
2. % ls -l /marfs/fuse
3. Should see packed-file in /marfs/fuse
4. Verify that it is at least the same size as the source file on /otherfs/fuse
5. Get the attribute that tells what the object name is in the object store and verify that there is just one object
6. Use s3curl.pl to verify that the object exists and is the correct size.

### Copy a Single Packed File using PFTool

1. % pfcp /otherfs/pftool/packed-file /marfs/pftool
2. % pfls /marfs/pftool
3. Should see packed-file in /marfs/pftool
4. pfcm -M /otherfs/pftool/packed-file /marfs/pftool/packed-file
5. Should see that the files compare to be the same.
6. Get the attribute that tells what the object name is in the object store and verify that there is just one object and it is designated a PACKED file

### Copy a Single Multi File using FUSE daemon

Note: when using the MarFS FUSE daemon to write the file it will become a Uni File.

1. % cp /otherfs/fuse/multi-file /marfs/fuse
2. % ls -l /marfs/fuse
3. Should see multi-file in /marfs/fuse
4. Verify that it is at least the same size as the source file on /otherfs/fuse
5. Get the attribute that tells what the object name is in the object store and verify that there is just one object
6. Use s3curl.pl to verify that the object exists and is the correct size.

### Copy a Single Multi File using PFTool

1. % pfcp /otherfs/pftool/multi-file /marfs/pftool
2. % pfls /marfs/pftool
3. Should see multi-file in /marfs/pftool
4. % pfcm -M /otherfs/pftool/multi-file /marfs/pftool/multi-file
5. Should see that the files compare to be the same.
6. Get the attribute that tells what the object name is in the object store and verify that there are multiple objects and it is designated a MULTI file

### Copy a Directory Tree using FUSE daemon

1. % cp -r /otherfs/fuse/dir-tree /marfs/fuse
2. % ls -lR /marfs/fuse/dir-tree
3. Should see dir-tree and all its subdirectories and files in /marfs/fuse
4. Verify that the directory structure and file sizes are the same as the source directory on /otherfs/fuse

### Copy a Directory Tree using PFTool

1. % pfcp -R /otherfs/pftool/dir-tree /marfs/pftool
2. % pfls -R /marfs/pftool/dir-tree
3. Should see dir-tree and all its subdirectories and files in /marfs/pftool
4. % pfcm -M -R /otherfs/pftool/dir-tree /marfs/pftool/dir-tree
5. Should see that the trees compare to be the same.

### Copy a Non-Existent File using FUSE daemon

1. % cp /otherfs/fuse/non-existent-file /marfs/fuse
2. Should get an error that non-existent-file does not exist.

### Copy a Non-Existent File using PFTool

1. % pfcp /otherfs/pftool/non-existent-file /marfs/pftool
2. Should get an error that non-existent-file does not exist.

### Copy a Non-Existent Directory Tree using FUSE daemon

1. % cp -r /otherfs/fuse/non-existent-dir-tree /marfs/fuse
2. Should get an error that non-existent-dir-tree does not exist.

### Copy a Non-Existent Directory Tree using PFTool

1. % pfcp -R /otherfs/pftool/non-existent-dir-tree /marfs/pftool
2. Should get an error that non-existent-dir-tree does not exist.

## FUSE/PFTool Copy Each Other's Files

This test will show that PFTool can copy a file written by FUSE and FUSE can copy a file written by PFTool from MarFS to POSIX.

All tests will require these steps:

1. FUSE daemon and PFTool copy tests to and from MarFS completed, such that Uni, Packed, and Multi files, and a directory tree created by each tool exist in /marfs.
2. Login to an interactive FTA

### Copy a Single Uni File using FUSE daemon

1. % cp /marfs/pftool/uni-file /otherfs/fuse
2. % ls -l /otherfs/fuse
3. Should see uni-file in /otherfs/fuse
4. Verify that it is at least the same size as the source file on /marfs/pftool

### Copy a Single Uni File using PFTool

1. % pfcp /marfs/fuse/uni-file /otherfs/pftool
2. % pfls /otherfs/pftool
3. Should see uni-file in /otherfs/pftool
4. % pfcm -M /otherfs/pftool/uni-file /marfs/fuse/uni-file
5. Should see that the files compare to be the same.
6. Get the attribute that tells what the object name is in the object store and verify that there are multiple objects and it is designated a UNI file

### Copy a Single Packed File using FUSE daemon

1. % cp /marfs/pftool/packed-file /otherfs/fuse
2. % ls -l /otherfs/fuse
3. Should see packed-file in /otherfs/fuse
4. Verify that it is at least the same size as the source file on /marfs/pftool

### Copy a Single Packed File using PFTool

1. % pfcp /marfs/fuse/packed-file /otherfs/pftool
2. % pfls /otherfs/pftool
3. Should see packed-file in /otherfs/pftool
4. % pfcm -M /otherfs/pftool/packed-file /marfs/fuse/packed-file
5. Should see that the files compare to be the same.
6. Get the attribute that tells what the object name is in the object store and verify that there are multiple objects and it is designated a PACKED file

### Copy a Single Multi File using FUSE daemon

1. % cp /marfs/pftool/multi-file /otherfs/fuse
2. % ls -l /otherfs/fuse
3. Should see multi-file in /otherfs/fuse
4. Verify that it is at least the same size as the source file on /marfs/pftool

### Copy a Single Multi File using PFTool

1. % pfcp /marfs/fuse/multi-file /otherfs/pftool
2. % pfls /otherfs/pftool
3. Should see multi-file in /otherfs/pftool
4. % pfcm -M /otherfs/pftool/multi-file /marfs/fuse/multi-file
5. Should see that the files compare to be the same.
6. Get the attribute that tells what the object name is in the object store and verify that there are multiple objects and it is designated a MULTI file

### Copy a Directory Tree using FUSE daemon

1. % cp -r /marfs/pftool/dir-tree /otherfs/fuse
2. % ls -lR /otherfs/fuse/dir-tree
3. Should see dir-tree and all its subdirectories and files in /otherfs/fuse
4. Verify that the directory structure and file sizes are the same as the source directory on /marfs/pftool

### Copy a Directory Tree using PFTool

1. % pfcp -R /marfs/fuse/dir-tree /otherfs/pftool
2. % pfls -R /otherfs/pftool/dir-tree
3. Should see dir-tree and all its subdirectories and files in /otherfs/pftool
4. % pfcm -M -R /otherfs/pftool/dir-tree /marfs/fuse/dir-tree
5. Should see that the trees compare to be the same.

## Rename MarFS Files

Renaming is only valid for the FUSE daemon. This test will show that a user can rename a file on MarFS. The test will include renaming single files of sizes that will yield all MarFS file types and a directory tree of files. Scenarios will include renaming a non-existent file and a non-existent directory tree.

All tests will require these steps:

1. Have files in /marfs that are known size to be Uni, Packed, and Multi files.
2. Have a directory, dir-tree, on /marfs that is sufficiently sized to assure yourself that directory tree renaming works.
3. Login to an interactive FTA.
4. Change directories to /marfs.

### Rename a Single Uni File

1. % mv uni-file uni-file-new
2. % ls -l /marfs
3. Should see uni-file-new in /marfs, but not uni-file.
4. Verify that it is the same size as the source file, uni-file.
5. Get the attribute that tells what the object name is in the object store and verify that there is just one object.
6. Use s3curl.pl to verify that the object exists and is the correct size.

### Rename a Single Packed File

Note: when using the MarFS FUSE daemon to rename the file it should remain a Packed File.

1. % mv packed-file packed-file-new
2. % ls -l /marfs
3. Should see packed-file-new in /marfs, but not packed-file.
4. Verify that it is the same size as the source file, packed-file
5. Get the attribute that tells what the object name is in the object store and verify that there is just one object.
6. Use s3curl.pl to verify that the object exists and is the correct size.

### Rename a Single Multi File

Note: when using the MarFS FUSE daemon to rename the file it should remain a Multi File.

1. % mv multi-file mult-file-new
2. % ls -l /marfs
3. Should see multi-file-new in /marfs, but not multi-file.
4. Verify that it is the same size as the source file, multi-file.
5. Get the attribute that tells what the object names are in the object store and verify that there are multiple objects.
6. Use s3curl.pl to verify that the objects exist and are the correct sizes.

### Rename a Directory Tree

1. % mv dir-tree dir-tree-new
2. % ls -lR /marfs/dir-tree-new
3. Should see dir-tree-new and all its subdirectories and files in /marfs, but not dir-tree.
4. Verify that the directory structure and file sizes are the same as the source directory, dir-tree.

### Rename a Non-Existent File

1. % mv non-existent-file non-existent-file-new
2. Should get an error that non-existent-file does not exist.

### Rename a Non-Existent Directory Tree

1. % mv non-existent-dir-tree non-existent-dir-tree-new
2. Should get an error that non-existent-dir-tree does not exist.

## Delete MarFS Files

Deleting is only valid for the FUSE daemon. This test will show that a user can delete a file on MarFS. The test will include deleting single files of sizes that are of all MarFS file types and a directory tree of files. Scenarios will include deleting a non-existent file and a non-existent directory tree.

All tests will require these steps:

1. Have files in /marfs that are known size to be Uni, Packed, and Multi files.
2. Have a directory, dir-tree, on /marfs that is sufficiently sized to assure yourself that directory tree deletion works.
3. Login to an interactive FTA.
4. Change directories to /marfs.

### Delete a Single Uni File

1. Get the attribute that tells what the object name is in the object store and verify that there is just one object.
2. % rm uni-file
3. % ls -l /marfs
4. Should see that uni-file is gone.
5. Use s3curl.pl to verify that the object is gone.

### Delete a Single Packed File

1. Get the attribute that tells what the object name is in the object store and verify that there is just one object.
2. Get the size of packed-file's data.
3. % rm packed-file
4. % ls -l /marfs
5. Should see that packed-file is gone.
6. Use s3curl.pl to verify that the object is of a different size. It should be the original size less the size of packed-file.

### Delete a Single Multi File

1. Get the attribute that tells what the object names are in the object store and verify that there are multiple objects.
2. % rm mult-file
3. % ls -l /marfs
4. Should see that multi-file is gone.
5. Use s3curl.pl to verify that the objects are gone.

### Delete a Directory Tree

1. Pick a few arbitrary files and get the attributes that tell what the object names are in the object store and verify that the objects exist.
2. % rm -rf dir-tree
3. % ls -lR /marfs/dir-tree
4. Should see that the dir-tree is gone.
5. Use s3curl.pl to verify that the few arbitrary objects are gone.

### Delete a Non-Existent File

1. % rm non-existent-file
2. Should get an error that non-existent-file does not exist.

### Delete a Non-Existent Directory Tree

1. % rm -rf non-existent-dir-tree
2. Should get an error that non-existent-dir-tree does not exist.

## List MarFS Files

This test will show that a user can list files on MarFS with and without "-l". The test will include listing single files of sizes that are of all MarFS file types and a directory. Scenarios will include listing a non-existent file and a non-existent directory tree.

All tests will require these steps:

1. Have files in /marfs that are known size to be Uni, Packed, and Multi files.
2. Have a directory, dir-tree, on /marfs that is sufficiently sized to assure yourself that directory tree listing works.
3. Login to an interactive FTA.

### List a Single Uni File using FUSE daemon

1. % ls /marfs/fuse/uni-file
2. % ls -l /marfs/fuse/uni-file
3. Should see the uni-file without and with metadata attributes.
4. Get the attribute that tells what the object name is in the object store and verify that there is just one object.
5. Use s3curl.pl to verify that the object is there.

### List a Single Uni File using PFTool

1. % pfls /marfs/pftool/uni-file
2. Should see the uni-file.
3. Get the attribute that tells what the object name is in the object store and verify that there are multiple objects and it is designated a UNI file

### List a Single Packed File using FUSE daemon

1. % ls /marfs/fuse/packed-file
2. % ls -l /marfs/fuse/packed-file
3. Should see the packed-file without and with metadata attributes.
4. Get the attribute that tells what the object name is in the object store and verify that there is just one object.
5. Use s3curl.pl to verify that the object is there.

### List a Single Packed File using PFTool

1. % pfls /marfs/pftool/packed-file
2. Should see the packed-file.
3. Get the attribute that tells what the object name is in the object store and verify that there are multiple objects and it is designated a PACKED file

### List a Single Multi File using FUSE daemon

1. % ls /marfs/fuse/multi-file
2. % ls -l /marfs/fuse/multi-file
3. Should see the multi-file without and with metadata attributes.
4. Get the attribute that tells what the object names are in the object store and verify that there more than one object.
5. Use s3curl.pl to verify that the objects are there.

### List a Single Multi File using PFTool

1. % pfls /marfs/pftool/multi-file
2. Should see the multi-file.
3. Get the attribute that tells what the object name is in the object store and verify that there are multiple objects and it is designated a MULTI file

### List a Directory Tree using FUSE daemon

1. % ls -R /marfs/fuse/dir-tree
2. % ls -lR /marfs/fuse/dir-tree
3. Should see the files and directories at all levels of the directory tree without and with metadata attributes.

### List a Directory Tree using PFTool

1. % pfls -r /marfs/pftool/dir-tree
2. Should see the files and directories at all levels of the directory tree.

### List a Non-Existent File using FUSE daemon

1. % ls /marfs/fuse/non-existent-file
2. Should get an error that non-existent-file does not exist.

### List a Non-Existent File using PFTool

1. % pfls /marfs/pftool/non-existent-file
2. Should get an error that non-existent-file does not exist.

### List a Non-Existent Directory Tree using FUSE daemon

1. % ls -R /marfs/fuse/non-existent-dir-tree
2. Should get an error that non-existent-dir does not exist.

### List a Non-Existent Directory Tree using PFTool

1. % pfls -r /marfs/pftool/non-existent-dir-tree
2. Should get an error that non-existent-dir does not exist.

## Symbolic Links to MarFS Files

Creating symbolic links is only valid for the FUSE daemon. This test will show that a user can create a symbolic link to files on MarFS. The test will include creating symbolic links to single files of sizes that are of all MarFS file types and a directory. Scenarios will include creating a symbolic link to a non-existent file and a non-existent directory tree.

All tests will require these steps:

1. Have files in /marfs that are known size to be Uni, Packed, and Multi files.
2. Have a directory, dir-tree, on /marfs that is sufficiently sized to assure yourself that directory tree symbolic link creation works.
3. Login to an interactive FTA.
4. Change directories to /marfs.

### Create a Symbolic Link to a Single Uni File

1. % ln -s uni-file uni-file-link
2. % ls -L uni-file-link
3. Should see the uni-file that this symbolic link references.
4. Get the attribute that tells what the object name is in the object store and verify that there is just one object.
5. Use s3curl.pl to verify that the object is there.

### Create a Symbolic Link to a Single Packed File

1. % ln -s packed-file packed-file-link
2. % ls -L packed-file-link
3. Should see the packed-file that this symbolic link references.
4. Get the attribute that tells what the object name is in the object store and verify that there is just one object.
5. Use s3curl.pl to verify that the object is there.

### Create a Symbolic Link to a Single Multi File

1. % ln -s multi-file multi-file-link
2. % ls -L multi-file-link
3. Should see the multi-file that this symbolic link references.
4. Get the attribute that tells what the object names are in the object store and verify that there more than one object.
5. Use s3curl.pl to verify that the objects are there.

### Create a Symbolic Link to a Directory Tree

1. % ln -s dir-tree dir-tree-link
2. % ls -L dir-tree-link
3. Should see the files and directories at the first level of the directory that this symbolic link references.

### Create a Symbolic Link to a Non-Existent File

1. % ln -s non-existent-file non-existent-file-link
2. Should get an error that non-existent-file does not exist.

### Create a Symbolic Link to a Non-Existent Directory Tree

1. % ln -s non-existent-dir-tree non-existent-dir-tree-link
2. Should get an error that non-existent-dir does not exist.

## MarFS File Extended Attributes

Manipulating xattrs is only valid for the FUSE daemon. This test will show that the MarFS FUSE daemon handles xattrs (extended attributes) correctly. There are xattrs that are reserved for MarFS use and the user shall not be able to see or affect them in any manner. Here are known MarFS xattrs: user.marfs\_post, user.marfs\_objid, user.marfs\_restart. The user shall be able to write, print, delete, and clear non-MarFS xattrs.

All tests will require these steps:

1. Have a file, xattr-test-file, in /marfs that will be the subject of our xattr tests on a file.
2. Have a directory, xattr-test-dir-tree, on /marfs that is sufficiently sized to assure yourself that directory tree xattr operations work.
3. Login to an interactive FTA.
4. Change directories to /marfs.

### Create an xattr on a File

1. % xattr -w test\_xattr1 "test xattr value 1" xattr-test-file
2. % xattr -w test\_xattr2 "test xattr value 2" xattr-test-file
3. % xattr -w test\_xattr3 "test xattr value 3" xattr-test-file
4. Verify that it completes without error.

### List xattrs on a File

1. % xattr -l xattr-test-file
2. Verify that only the test\_xattr\* xattrs and their values are listed. No MarFS reserved xattrs shall be listed.

### Print an xattr on a File

1. % xattr -p test\_xattr1 xattr-test-file
2. Verify that only the test\_xattr1 value is printed. No other xattr, MarFS reserved or otherwise, shall be printed.

### Delete an xattr from a File

1. % xattr -d test\_xattr1 xattr-test-file
2. % xattr -l xattr-test-file
3. Verify that only test\_xattr2 and test\_xattr3 and their values are listed. No MarFS reserved xattrs shall be listed.

### Clear xattrs from a File

1. % xattr -c xattr-test-file
2. % xattr -l xattr-test-file
3. Verify that no test\_xattr\* and their values are listed. No MarFS reserved xattrs shall be listed.

### Create an xattr on a Non-existent File

1. % xattr -w test\_xattr1 "test xattr value 1" nonexistent-file
2. Verify that this results in an error.

### List xattrs on a Non-existent File

1. % xattr -l nonexistent-file
2. Verify that this results in an error.

### Print an xattr on a Non-existent File

1. % xattr -p test\_xattr1 nonexistent-file
2. Verify that this results in an error.

### Delete an xattr from a Non-existent File

1. % xattr -d test\_xattr1 nonexistent-file
2. Verify that this results in an error.

### Clear xattrs from a Non-existent File

1. % xattr -c nonexistent-file
2. Verify that this results in an error.

### Create an xattr on a Directory Tree

1. % xattr -wr test\_xattr1 "test xattr value 1" xattr-test-dir-tree
2. % xattr -wr test\_xattr2 "test xattr value 2" xattr-test- dir-tree
3. % xattr -wr test\_xattr3 "test xattr value 3" xattr-test- dir-tree
4. Verify that it completes without error.

### List xattrs on a Directory Tree

1. % xattr -lr xattr-test- dir-tree
2. Verify that only the test\_xattr\* xattrs and their values are listed for all members of the directory tree. No MarFS reserved xattrs shall be listed.

### Print an xattr on a Directory Tree

1. % xattr -pr test\_xattr1 xattr-test- dir-tree
2. Verify that only the test\_xattr1 value is printed for all members of the directory tree. No other xattr, MarFS reserved or otherwise, shall be printed.

### Delete an xattr from a Directory Tree

1. % xattr -dr test\_xattr1 xattr-test- dir-tree
2. % xattr -lr xattr-test- dir-tree
3. Verify that only test\_xattr2 and test\_xattr3 and their values are listed for all members of the directory tree. No MarFS reserved xattrs shall be listed.

### Clear xattrs from a Directory Tree

1. % xattr -cr xattr-test- dir-tree
2. % xattr -lr xattr-test- dir-tree
3. Verify that no test\_xattr\* and their values are listed for all members of the directory tree. No MarFS reserved xattrs shall be listed.

### Create an xattr on a Non-existent Directory Tree

1. % xattr -wr test\_xattr1 "test xattr value 1" nonexistent- dir-tree
2. Verify that this results in an error.

### List xattrs on a Non-existent Directory Tree

1. % xattr -lr nonexistent- dir-tree
2. Verify that this results in an error.

### Print an xattr on a Non-existent Directory Tree

1. % xattr -pr test\_xattr1 nonexistent- dir-tree
2. Verify that this results in an error.

### Delete an xattr from a Non-existent Directory Tree

1. % xattr -dr test\_xattr1 nonexistent- dir-tree
2. Verify that this results in an error.

### Clear xattrs from a Non-existent Directory Tree

1. % xattr -cr nonexistent-dir-tree
2. Verify that this results in an error.

### Create a Reserved xattr on a File

1. % xattr -w user.marfs\_post "reserved xattr value 1" xattr-test-file
2. Verify that this results in an error.

### List Reserved xattrs on a File

1. % xattr -l xattr-test-file
2. Verify that no MarFS reserved xattrs are listed.
3. From a MarFS metadata file system server, verify that the MarFS reserved xattrs are listed.

### Print a Reserved xattr on a File

1. % xattr -p user.marfs\_post xattr-test-file
2. Verify that this reserved MarFS xattr is not printed.
3. From a MarFS metadata file system server, verify that the value of user.marfs\_post is printed.

### Delete a Reserved xattr from a File

1. % xattr -d user.marfs\_post xattr-test-file
2. Verify that this results in an error.
3. From a MarFS metadata file system server, verify that the value of user.marfs\_post is printed. That is, that it really was not deleted.

### Clear Reserved xattrs from a File

1. % xattr -c xattr-test-file
2. Verify that this results in an error.
3. From a MarFS metadata file system server, verify that the values of the MarFS reserved xattrs are listed. That is, that they really were not cleared.

### Create a Reserved xattr on a Directory Tree

1. % xattr -wr user.marfs\_post "reserved xattr value 1" xattr-test-dir-tree
2. Verify that this results in an error.

### List Reserved xattrs on a Directory Tree

1. % xattr -lr xattr-test- dir-tree
2. Verify that no MarFS reserved xattrs are listed.
3. From a MarFS metadata file system server, verify that the MarFS reserved xattrs are listed.

### Print a Reserved xattr on a Directory Tree

1. % xattr -pr user.marfs\_post xattr-test- dir-tree
2. Verify that this reserved MarFS xattr is not printed.
3. From a MarFS metadata file system server, verify that the value of user.marfs\_post is printed.

### Delete a Reserved xattr from a Directory Tree

1. % xattr -dr user.marfs\_post xattr-test- dir-tree
2. Verify that this results in an error.
3. From a MarFS metadata file system server, verify that the value of user.marfs\_post is printed. That is, that it really was not deleted.

### Clear Reserved xattrs from a Directory Tree

1. % xattr -cr xattr-test- dir-tree
2. Verify that this results in an error.
3. From a MarFS metadata file system server, verify that the values of the MarFS reserved xattrs are listed. That is, that they really were not cleared.

## Truncate File into Existence

Truncating a file into existence is only valid for the FUSE daemon. Include trying this for an existing file (should make it zero size) and for a non-existing and existing file to a non-zero size (should fail).

All tests will require these steps:

1. Login to an interactive FTA.
2. Change directories to /marfs.

### Create Initial File

1. % truncate --size=0 truncate-file1
2. % ls -l truncate-file1
3. Verify it exists and is 0 bytes in size.

### Grow Initial File by 1M

1. % truncate --size=+1M truncate-file1
2. % ls -l truncate-file1
3. Verify that an error is returned. Any non-zero size will result in an error, whether or not the file already exists.

### Create Initial File with Non-Zero Size

1. % truncate --size=1M truncate-file2
2. % ls -l truncate-file
3. Verify that an error is returned. Any non-zero size will result in an error, whether or not the file already exists.

## Concurrency Test

I'm looking at the fd\_consistency test from the PLFS regression test suite. This is supposed to exercise any FUSE daemon with multiple concurrent clients. It comes with a PLFS tarball and repeatedly untars it and builds it. I need to see if this is really applicable to MarFS.

In the meantime, we'll just do this:

for i in `seq 1 64`; do

dd if=/dev/urandom of=/marfs/test00/big.$i bs=1000 count=$i &

done

They should all complete successfully. To get a rough idea that they did what they were supposed to, list all the files.

% ls -l /marfs/test00/test.\*

Verify that the 64 test files are there and are the correct size.

## Implementation Hiding Tests

These tests ensure that the user cannot access the GPFS file system that is the MarFS metadata repository and the object store that is the MarFS data repository.

Assume that the MarFS metadata repository is mounted at /mds. Assume that the MarFS data repository is accessible via S3 at http://objectstore.domain/.

### Normal Operation

1. Login to an interactive FTA.
2. % df
3. Verify that you do not see /mds.
4. % cat $HOME/.awsAuth
5. Verify that this file does not exist.
6. % cat $HOME/.s3curl
7. Verify that this file does not exist.
8. Create a $HOME/.s3curl file and take your best shot at generating an id that will work. The content of the file looks like this:  
     
   %awsSecretAccessKeys = (  
    brettk => {  
    id => 'brettk',  
    key => '<alphanumeric-string-that-is-your-access-key',  
    },  
   );
9. % ./s3curl.pl --id=brettk -- http://objectstore.domain/

### Kill FUSE Daeamon

1. Login to an interactive FTA.
2. % kill -9 <pid-of-FUSE-daemon>
3. Verify that you do not see /mds.

## Permission Maintenance Tests

Maintaining proper permissions when copying a file is valid only for PFTool. The FUSE daemon uses the UNIX cp command. PFTool's copy command, pfcp, should mimic the behavior of the UNIX cp. These tests will show that pfcp behaves correctly. Scenarios will include using pfcp to copy files from a PFS to MarFS and vice-versa, and trying to pfcp files to which access should not be permitted.

All tests pfcp'ing from /otherfs to /marfs will require these steps:

1. Have files in /otherfs/perms/src that have the various permissions set for owner, group, and world accesses that we will use in the tests. These files will become the source for the copies back to /otherfs/perms/dst. Permissions for group files will have u:rwx and o:---. We will vary the (g)roup permissions. Permissions for other user files will have u:rwx and g:rwx. We will vary the (o)ther user permissions.
2. Login to an interactive FTA.
3. Change directories to /otherfs/perms/src.

### Copy a Read-Only for User file to MarFS as Owning User

1. ls -l u-r--
2. Verify that permissions are r-- --- ---
3. % pfcp u-r-- /marfs/perms/dst
4. ls -l /marfs/perms/dst/u-r--
5. Verify that permissions are r-- --- ---

### Copy a Read/Write for Owner file to MarFS as Owning User

1. ls -l u-rw-
2. Verify that permissions are rw- --- ---
3. % pfcp u-rw- /marfs/perms/dst
4. ls -l /marfs/perms/dst/u-rw-
5. Verify that permissions are rw- --- ---

### Copy a Read/Write/Execute for Owner file to MarFS as Owning User

1. ls -l u-rwx
2. Verify that permissions are rwx --- ---
3. % pfcp u-rwx /marfs/perms/dst
4. ls -l /marfs/perms/dst/u-rwx
5. Verify that permissions are rwx --- ---

### Copy a Read/Execute for Owner file to MarFS as Owning User

1. ls -l u-r-x
2. Verify that permissions are r-x --- ---
3. % pfcp u-r-x /marfs/perms/dst
4. ls -l /marfs/perms/dst/u-r-x
5. Verify that permissions are r-x --- ---

All tests pfcp'ing from /marfs to /otherfs will require these steps:

1. There will now be files in /marfs/perms/dst that have the various permissions set for user, group, and other users accesses that were pfcp'd from /otherfs/perms/src.
2. Login to an interactive FTA.
3. Change directories to /marfs/perms/dst.

### Copy a Read-Only for Owner file from MarFS as Owning User

1. ls -l u-r--
2. Verify that permissions are r-- --- ---
3. % pfcp u-r-- /otherfs/perms/dst
4. ls -l /otherfs/perms/dst/u-r--
5. Verify that permissions are r-- --- ---

### Copy a Read/Write for Owner file from MarFS as Owning User

1. ls -l u-rw-
2. Verify that permissions are rw- --- ---
3. % pfcp u-rw- /otherfs/perms/dst
4. ls -l /otherfs/perms/dst/u-rw-
5. Verify that permissions are rw- --- ---

### Copy a Read/Write/Execute for Owner file from MarFS as Owning User

1. ls -l u-rwx
2. Verify that permissions are rwx --- ---
3. % pfcp u-rwx /otherfs/perms/dst
4. ls -l /otherfs/perms/dst/u-rwx
5. Verify that permissions are rwx --- ---

### Copy a Read/Execute for Owner file from MarFS as Owning User

1. ls -l u-r-x
2. Verify that permissions are r-x --- ---
3. % pfcp u-rwx /otherfs/perms/dst
4. ls -l /otherfs/perms/dst/u-r-x
5. Verify that permissions are r-x --- ---

## Performance Tests

These are tests to check the performance of MarFS when doing transfers that we expect to see in production-sized installations.

### 7.5 TB Test

A small sized transfer is in the 7.5 TB range. In HPC (High Performance Computing) this size of transfer is done with a set of files where there is one (or some N) file(s) per process in the set, called N:N (N to N, as in N processes write/read N files), or where there is one large file for all the processes, called N:1 (N to 1, as in N processes write/read one file).

For the N:N case, we will use 7,500 1 GB files, as 1 GB is representative of the file size a process will write to save its state at the time of this writing, 22-Dec-2015.

For the N:1 case, we will use one 7.5 TB file.

#### N:N 7.5TB Test Case

1. % pfcp /otherfs/pftool/7.5TB\_NN /marfs/pftool/7.5TB\_NN
2. % pfls /marfs/pftool/7.5TB\_NN
3. Should see 7,500 1 GB files in /marfs/pftool/7.5TB\_NN
4. % pfcm -M /otherfs/pftool/7.5TB\_NN /marfs/pftool/7.5TB\_NN
5. Should see that the files compare to be the same.
6. % pfcp /marfs/pftool/7.5TB\_NN /otherfs/pftool/7.5TB\_NN
7. % pfls /otherfs/pftool/7.5TB\_NN
8. Should see 7,500 1 GB files in /otherfs/pftool/7.5TB\_NN
9. % pfcm -M /marfs/pftool/7.5TB\_NN /otherfs/pftool/7.5TB\_NN
10. Should see that the files compare to be the same.

#### N:1 7.5TB Test Case

1. % pfcp /otherfs/pftool/7.5TB\_N1 /marfs/pftool/7.5TB\_N1
2. % pfls /marfs/pftool/7.5TB\_N1
3. Should see one 7.5 TB file in /marfs/pftool/7.5TB\_N1
4. % pfcm -M /otherfs/pftool/7.5TB\_N1 /marfs/fuse/7.5TB\_N1
5. Should see that the files compare to be the same.
6. % pfcp /marfs/pftool/7.5TB\_N1 /otherfs/pftool/7.5TB\_N1
7. % pfls /otherfs/pftool/7.5TB\_N1
8. Should see one 7.5 TB file in /otherfs/pftool/7.5TB\_N1
9. % pfcm -M /marfs/pftool/7.5TB\_N1 /otherfs/pftool/7.5TB\_N1
10. Should see that the files compare to be the same.

### 20 TB Test

A medium-sized transfer is in the 20 TB range. Again, we'll test N:N and N:1 scenarios.

For the N:N case, we will use 20,000 1 GB files, as 1 GB is representative of the file size a process will write to save its state at the time of this writing, 22-Dec-2015.

For the N:1 case, we will use one 20 TB file.

#### N:N 20TB Test Case

1. % pfcp /otherfs/pftool/20GB\_NN /marfs/pftool/20GB\_NN
2. % pfls /marfs/pftool/20GB\_NN
3. Should see 20,000 1 GB files in /marfs/pftool/20GB\_NN
4. % pfcm -M /otherfs/pftool/20GB\_NN /marfs/pftool/20GB\_NN
5. Should see that the files compare to be the same.
6. % pfcp /marfs/pftool/20GB\_NN /otherfs/pftool/20GB\_NN
7. % pfls /otherfs/pftool/20GB\_NN
8. Should see 20,000 1 GB files in /otherfs/pftool/20GB\_NN
9. % pfcm -M /marfs/pftool/20GB\_NN /otherfs/pftool/20GB\_NN
10. Should see that the files compare to be the same.

#### N:1 20TB Test Case

1. % pfcp /otherfs/pftool/20TB\_N1 /marfs/pftool/20TB\_N1
2. % pfls /marfs/pftool/20TB\_N1
3. Should see one 20 TB file in /marfs/pftool/20TB\_N1
4. % pfcm -M /otherfs/pftool/20TB\_N1 /marfs/fuse/20TB\_N1
5. Should see that the files compare to be the same.
6. % pfcp /marfs/pftool/20TB\_N1 /otherfs/pftool/20TB\_N1
7. % pfls /otherfs/pftool/20TB\_N1
8. Should see one 20 TB file in /otherfs/pftool/20TB\_N1
9. % pfcm -M /marfs/pftool/20TB\_N1 /otherfs/pftool/20TB\_N1
10. Should see that the files compare to be the same.

### User File Set Test

Whatever computing campaign is going on at the time, acquire a representative N:N and N:1 file set from one of the computing campaign users. Use these files to transfer back and forth from the PFS (parallel file system) and Campaign Storage. The processes will be the same as those listed for the 7.5 TB and 20 TB test cases, but the file sets source and target locations will change to a place reserved for these file sets.

# Utility Programs for MarFS Management

The utility programs operate on a MarFS namespace. It is not required that a namespace be its own MarFS mount point. That is, there can be one or more namespaces in a given MarFS mount point. So, for example, if we were to have a mount point that we were using through the FUSE daemon called, “/marfs/fs1”, we might have a couple namespaces, “/marfs/fs1/atorrez” and “/marfs/fs1/jti” that could be seen as different namespaces and be managed and monitored separately by the utility programs. In this context, a namespace and fileset are the same.

Here is a example listing of filesets:

First determine the gpfs mount point:

df | grep gpfs

/dev/marfs-gpfs 5860543680 80687296 5779856384   2% /gpfs/marfs-gpfs

% mmlsfileset marfs-gpfs

Filesets in file system 'marfs-gpfs':

Name                     Status    Path

root                     Linked    /gpfs/marfs-gpfs

project\_a                Linked    /gpfs/marfs-gpfs/project\_a

project\_b                Linked    /gpfs/marfs-gpfs/project\_b

project\_c                Linked    /gpfs/marfs-gpfs/project\_c

trash                    Linked    /gpfs/marfs-gpfs/trash

The following tests make reference to both the gpfs mount point and the fuse mount point . In order to clarify the meaning of each of these mounts the following example commands can be used to obtain them.

gpfs\_mount obtained by

a) mount –t gpfs

b) pick desired mount e.g. user view of gpfs mount

fuse\_mount obtained by

a) mount | grep fuse

b) user view of fuse mount

This test procedure assumes the user has setup the config file namespace appropriately for testing.

## Quota Management

Verification of the quota management script can be accomplished using the Linux disk usage (du) command. The command can be targeted at various namespaces (gpfs filesets) in order to match the quota script output. Utility scripts are used to populate MarFS namespaces with various object types. The supporting marfs object/file generation scripts reside in /marfs/fuse/scripts. Modify scripts as necessary to point to the correct fuse mount point, and underlying directory as done in the config file.

The quota management script is named marfs\_quota and resides the marfs repository path: /marfs/utilities/gpfs/quota/src.

**Create objects and gpfs files**

1. Run the fuse the following fuse scipts:

./make\_multi\_uni /fuse\_mount/namespace/path

./make\_packed scality

NOTE: The make\_packed script requires an edit to configure

paths and object storage IP addresses. The applicable variables are MARFS, GPFS, and HOST

1. Run the quota script:

./marfs\_quota –d gpfs\_mount –o log

1. Open the log file (from –o option above) and look for namespace matching the config file and the script namespace used to create files.
2. cd to the underlying gpfs directory defined in the config file and scripts. Make sure that you are at the level right above the namespace directory.
3. Run du as follows:

du –b ./namespace.

1. Verify that the size returned from du matches the total\_size in the corresponding namespace in the log
2. Verify that the log uni\_count, multi\_count, and packed count match the scripts creation counts:

uni\_count = 200

multi\_count = 2

packed\_count = 8

1. Run du as follows:

du –b –a ./namespace | wc –l

1. Verify that the total\_file\_count equals the result from the previous step

total\_file\_count is calculated as follows:

200+2+8 – uni\_count +multi\_count+packed\_count

**Moving** **objects and gpfs files to trash**

1. From the fuse mount, verify that the files created in the first test above exist:

ls /fuse\_mount

Make note of number of files that exist in this directory:

ls /fuse\_mount/namespace/path/d1/ | wc –l.

1. Use du to determine the total size of the files in this directory:

du –b /fuse\_mount/namespace/path/d1 and make note.

1. Remove the files:

rm –rf d1

1. Run the quota script:

./marfs\_quota –d gpfs\_mount –o log

1. Verify that the namespace used in the first test above now shows a smaller size and file count because those files have been removed. The file count should represent the directories that still exist including namespace and directories below. Use ls –al to verify counts and sizes for those directories.
2. Find the trash namespace in the log and verify that the file count equals the number of files deleted.
3. Verify that the file size matches by running du:

du –b ./trash\_namespace

1. Verify that the original namespace listed in the log now shows a trash\_file\_count and trash size that matches the count and size that was noted in steps 2 and 3 above.

## Empty Trash

To verify the trash emptying we can traverse the metadata file system directory and look at the xattrs to find where the objects are. See that they are still there in the object system, empty the trash, and then see that they are gone from the object system. We have then verified that multi and uni are deleting the objects defined by the xattr as well as deleting the metadata files.

The garbage collection script is named marfs\_gc and is located in:

/marfs/utilities/gpfs/garbage\_collection/src

**Verify trash cleanup**

1. The previous test verified that files have been moved to the trash namespace
2. Verify that the files and directories are gone from the fuse mount and underlying gpfs directory and now exist in the trash
3. cd to the underlying gpfs trash namespace and run the command:

getfattr –d \*

1. Verify that that xattrs exist for all files in the directory
2. From the list of objects, create a script that does a GET of each object to verify that it exists.

Example script creation:

getfattr -d \* | grep marfs\_objid | sed -E 's/user\.marfs\_objid="/curl -X GET http:\/\/10\.135\.0\.22:81\//' | sed -E 's/\"//' > script\_name

1. Run the script and verify that the objects exist in the object system
2. Run the garbage collection script:

./marfs\_gc –d gpfs\_mount –o log –u root

1. Run the quota script again to verify that the trash namespace has no files left in it:

./marfs\_quota –d /gpfs\_mount –o log

1. Open the log and verify that no errors exist and the files and objects are logged as deleted
2. Run the script from step 5 to verify that the objects have now been deleted
3. Verify that the underlying gpfs directory is empty

**Verify incomplete packed object causes non-deletion**

This test verifies that a packed object will not be deleted if all the corresponding files of the pack do not exist in the trash directory.

1. Run the make\_incomplete\_packed script- %

./make\_incomplete\_packed

NOTE: The make\_incomplete\_packed script requires an edit to configure

paths and object storage IP addresses. The applicable variables are MARFS, GPFS, and HOST

1. Verify that 7 files exist under fuse mount and underlying gpfs
2. From the fuse mount run:

rm –f \*

1. Verify that the files have moved to the underlying gpfs trash directory
2. Verify that the xattrs in the packed files all have identical marfs\_objid xattrs
3. Verify that the object exists:

curl –X GET http://XX.XX.XX.XX:PP/object\_name

1. Run garbage collection:

./marfs\_gc –d gpfs\_mount –o log –u root

1. Verify that the files and object were NOT deleted by listing the trash directory, performing a curl –X get on the object and looking at the log file (no entries because nothing deleted).

**Verify restart xattr trash handling**

This test will verify that files containing restart xattrs due to a fuse daemon crash are handled correctly when moved to trash.

1. Get the pid of the fuse daemon by ps –ef | grep fuse.
2. Run the .make\_multi script:

./make\_multi /fuse\_mount/namespace/path

While the script is running, kill the fuse daemon with kill -9 pid (fuse daemon pid)

1. Verify that the files have restart xattrs:

getfattr –d /gpfs\_mount/namespace/path/\*

1. rm the files from the fuse\_mount path
2. Run garbage collection and verify that the log states the files and objects have been removed.

./marfs\_gc –d /gpfs\_mount –u root –o log

This test will verify that files containing restart xattrs due to a pftool failure are handled correctly when moved to trash.

1. Run pftool to start a multi file transfer as follows
2. Use dd to create an 11GB file:

dd if=/dev/zero of=/nfs/path/to/file/multi\_a bs=1G count=11

1. Run pftool as follows then ctrl-c it after a few 1G iterations

mpirun –x MARFSCONFIGRC –np 4 –H localhost /path/to/pftool –s 16M –S 10G –C 10G –r –p /nfs/path/to/file/multi\_a –c /fuse\_mount/namespace/path/filename –w 0 -vv

1. Verify that the files have restart xattrs:

getfattr –d /gpfs\_mount/namespace/path/\*

1. Remove the files from /fuse\_mount/namespace/path
2. Run garbage collection and verify that the log states that the files and objects have been deleted.

## Object Packing

Scanning inodes and determining which objects meet size requirements for packing is a method used to increase performance by reducing overhead. Once a packed object is created, xattrs associated with the previous objects are updated to reflect the new object name and associated offsets. Packing verification can be accomplished by running scripts to create small objects, running a script to read and save objects into files, running the packer utility, and running a script to read the packed object back using offsets to write to files that can then be compared to the original objects.

The packer utility script is named marfs\_packer and is located in:

/marfs/utilities/gpfs/packer/src

All other supporting scripts exist in the /marfs/fuse/scripts directory.

**Create packed object and verify**

1. Edit the make\_uni\_small script to create 100 small files in 4 directories: d1, d2, d3, and mixed. Run the make\_uni\_small script. This script will create the 4 directories and files. All the directories will contain 100 small files of size 40 bytes.
2. Run the make\_uni\_1M\_3 script. This will create 3 1MB files in the mixed directory.
3. Create a sub directory named testing in the current directory. This will be used to store files created from reading objects.
4. Run the get\_unpacked\_object script 4 times as listed below:

get\_unpacked\_object /gpfs/path/to/d1

get\_unpacked\_object /gpfs/path/to/d2

get\_unpacked\_object /gpfs/path/to/d3

get\_unpacked\_object /gpfs/path/to/mixed

This will read back all objects and create files containing the object data plus the recovery info inherent with all objects. Each of the directories (testing/\*) will now contain these objects in file form (uni.xxx.unpacked).

1. cd to the packer location and run the packer utility as follows:

./marfs\_packer –d gpfs\_mount –n namespace –l

The utility will return with a message that states:

“Found 403 objects to pack with a total size of XXXXXX”

1. Rerun the packer script, but this time leave off the “-l” argument
2. Once complete the packer utility should have packed the 404 objects into a new object.
3. Run the get\_packed\_object script 4 times as listed below:

get\_packed\_object /gpfs/path/to/d1 > d1.log

get\_packed\_object /gpfs/path/to/d2 > d2.log

get\_packed\_object /gpfs/path/to/d3 > d3.log

get\_packed\_object /gpfs/path/to/mixed > mixed.log

This will read back the objects using offsets from the file xattr and create files containing the object data plus the recovery info inherent with all objects. Each of the directories (testing/\*) will now contain these objects in file form (uni.xxx.packed). The script will then diff the unpacked files (uni.xxx.unpacked) with the files read back from the packed object (uni.xxx.packed). Review the d1.log,

d2.log, d3.log and mixed.log to verify That no ERRORS were reported (grep or search for “ERROR” in the log files).

**Create multiple packed objects and verify**

1. Run the make\_uni\_multi\_1M\_2200 script. This will create 2200 1M objects in the d4 directory.

make\_uni\_multi\_1M\_2200 fuse\_mount

1. Create a sub directory named testing in the current directory. This will be used to store files created from reading objects.
2. Run the get\_unpacked\_object script as listed below:

get\_unpacked\_object /gpfs/path/to/d4

This will read back all objects and create files containing the object data plus the recovery info inherent with all objects. Each of the directories (testing/\*) will now contain these objects in file form (uni.xxx.unpacked).

1. cd to the packer location and run the packer utility as follows:

./marfs\_packer –d gpfs\_mount –n namespace –l

The utility will return with a message that states:

“Found xxxxx objects to pack with a total size of xxxxx”

You may see multiple messages but the total count should add up to 2200.

This size does not include recovery info.

1. Rerun the packer script (step 5 above), but this time leave off the “-l” argument
2. Once complete, the packer utility should have packed the 2200 objects into a 3 new objects.
3. Run the get\_packed\_object script as listed below:

get\_packed\_object /gpfs/path/to/d4 > d4.log

This will read back the objects using offsets from the file xattr and create files containing the object data plus the recovery info inherent with all objects. The d4 directory will now contain these objects in file form (uni.xxx.packed). The script will then diff the unpacked files (uni.xxx.unpacked) with the files read back from the packed object (uni.xxx.packed). Review the d4.log, to verify that no ERRORS were reported (grep or search for “ERROR” in the log files).

**Object Repacking**

Repacking may become necessary if for some reason a gpfs file associated with a pack object is deleted prior to moving the packed object to trash. Garbage collection on packed objects verifies that the number of files for a particular packed object matches the number of chunks contained in the files xattrs. Garbage collection creates a file name tmp\_packed\_log that has a record of those object file count mismatches. The repack utility reads this file and proceeds to repack the object and update file xattrs accordingly.

The repack utility script is named marfs\_repack and is located in:

/marfs/utilities/gpfs/repack/scrc

All other supporting scripts exist in the /marfs/fuse/scripts directory.

Procedure is:

1. Run the make\_uni\_mixed script to create 403 files in the mixed directory.

./make\_uni\_mixed /fuse\_mount/namespace/path

1. pack objects by running the marfs\_repack utility:

//marfs\_repack ../../garbage-collection/src/tmp\_packed\_log –o log

1. CD to the associated gpfs directory and remove uni.023, uni.096 and uni.013 files
2. run the get\_partially\_packed script:

./get\_partially\_packed /gpfs\_mount/namespace/path/to/mixed/dir

This will retrieve file contents from the packed object based on only those files that remain after the removal in step 3

1. cd to the fuse\_mount directory and rm the mixed directory and its contents:

cd /fuse\_mount/namespace/

rm –rf mixed

1. run garbage collection in order to indentify the packed object that exists but missing the three files that were removed in step 3. Garbage collection will creat the tmp\_packed\_log file which identifies the object and the files found for that object.

./garbage-collection/src/marfs\_gc –d /gpfs\_mount –o log –u root

1. Run the repacker script to repack the incomplete object

./marfs\_repack –d ../../garbage-collection/src/marfs\_repack –o log

1. Run the verify\_repacked\_object script &> verify.log

./verify\_repack\_object mixed ../../garbage-collection/src/tmp\_packed\_log

1. Verify that no errors exists in verify.log

grep –I error verify.log

# Other Tests

## Require Authentication to Access Scality RING

The Scality RING will hold objects on behalf of users, but a single user owns the objects themselves.

On the batch FTAs a root-owned file holds the username and password that the MarFS FUSE daemon and PFTool will use to GET/PUT objects on the Scality RING.

On the GPFS servers a root-owned file holds the username and password that the management scripts will use to GET/PUT objects on the Scality RING.

### Test Compliance

To ensure that the Scality RING requires a username and password we will:

1) As root on the batch FTAs and the GPFS servers, su to a non-root user and show that the username and password file cannot be accessed. This will not result in a log or notification..

2) As root on the batch FTAs and the GPFS servers, show that the username and password file can be accessed.

3) As root on the batch FTAs and the GPFS servers, send a sproxyd GET or PUT command without the username and password to a known Scality RING server and show that it fails.

4) As root on the batch FTAs and the GPFS servers, send a sproxyd GET or PUT command with the username and password to a known Scality RING server and show that it succeeds.