# 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

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.

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 handing 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
4. Change directories to /marfs

### Copy a Single Uni File

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

### Copy a Single Packed File

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

### Copy a Single Multi File

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

### Copy a Directory Tree

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

### Copy a Non-Existent File

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

### Copy a Non-Existent Directory Tree

1. % cp -r non-existent-dir-tree /otherfs
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
4. Change directories to /otherfs

### Copy a Single Uni File

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

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

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

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

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

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

### Copy a Non-Existent File

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

### Copy a Non-Existent Directory Tree

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

## Rename MarFS Files

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

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.
4. Change directories to /marfs.

### List a Single Uni File

1. % ls uni-file
2. % ls -l 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 Packed File

1. % ls packed-file
2. % ls -l 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 Multi File

1. % ls multi-file
2. % ls -l 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 Directory Tree

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

### List a Non-Existent File

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

### List a Non-Existent Directory Tree

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

## Symbolic Links to MarFS Files

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

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

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 /gpfs/fs1. Assume that the MarFS data repository is accessible via S3 at http://10.140.0.15:9020/.

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/

# pftool for Batch Parallel Transfers

# 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.

Here is a listing of the filesets used for testing:

% 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

and the mount for MarFS is:

% df | grep marfs

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

## 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.

The top-level gpfs mount is used in the quota script in order to determine which gpfs mount to scan. The top level gpfs mount can be obtained by running:

% mount –t gpfs

determine top level fileset by looking at /dev/XXXXX

where XXXXX is the top-level fileset

The following tests assume that a trash namespace exists in the config.

**Create objects and gpfs files**

1. Run the fuse script make\_multi\_uni and make\_packed- % ./make\_multi\_uni % ./make\_packed

2. Run the quota script - % ./marfs\_quota –d /top-level/fileset –o log

3. Open the log and look for namespace matching the config file and the script namespace used to create files.

4. cd to the underlying gpfs defined in config and scripts. (The level right above the namespace).

5. Run du - % du –b ./namespace.

6. Verify that the size returned from du matches the total\_size in the corresponding namespace in the log

7. 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

8. %du –b –a ./namespace | wc –l

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

total\_file\_count is calculated as follows: 200+2+8+namespace+subdir+subdir………….

**Delete objects and gpfs files**

1. From the fuse mount, verify that the files created in the first test above exist – % ls /fuse/mount

2. Make note of number of files that exist in this directory- % ls /fuse/mount/d1/ | wc –l.

3. Use du to determine the total size of the files in this directory- % du –b /fuse/mount/d1 and make note.

3. Remove the files - % rm –rf d1

4. Run the quota script % ./marfs\_quota –d /top-level/fileset –o log

5. 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.

6. Find the trash namespace in the log and verify that the file count equals the number of files deleted \* 2 (for .path creations) + number of directories including namespace.

7. Verify that the file size matches by running du - % du –b ./trash\_namespace

8. 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 could get into 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 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

1. cd to the fuse mount and remove all the contents- rm –rf \*

2. Verify that the files and directories are gone from the fuse mount and underlying gpfs directory

3. cd to the underlying gpfs trash namespace and run the command % getfattr –d \*

4. Verify that that xattrs exist for all files in the directory

5. 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

6. Run the script and verify that the objects exist in the object system

7. Run the quota script to verify that files have been moved to the trash namespace

8. Run the garbage collection script- marfs\_gx –d /top-level/fileset –o log –f trash\_namespace

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

10. Run the script from step 5 to verify that the objects have been deleted

11. Verify that the underlying gpfs directory is empty