Title:	MarFS-Requirements-Design-Configuration-Admin
Author(s):	Kettering, Brett Michael Grider, Gary Alan
Intended for:	Report
Issued:	2015-07-08

Introduction

This document will be organized into sections that are defined by the requirements for a file system that presents a near-POSIX (Portable Operating System Interface) interface to the user, but whose data is stored in whatever form is most efficient for the type of data being stored. After defining the requirement the design for meeting the requirement will be explained. Finally there will be sections on configuring and administering this file system.

More and more, data dominates the computing world. There is a "sea" of data out there in many different formats that needs to be managed and used. "Mar" means "sea" in Spanish. Thus, this product is dubbed MarFS, a file system for a sea of data.

Rationale

Many may question why a new product like MarFS is necessary. While there are products that are developing, none provides a scalable near-POSIX interface with adequate performance at this time.

Current object storage systems use erasure based disk systems to store the data, which is a positive thing. RAID (Redundant Array of Independent Disks) solutions, such as RAID-6, are not adequate data protection given the reliability and resilience required for all the hardware needed to hold this data. Products such as Cleversafe, Scality, and EMC ViPR are moving towards the "sea of data" concept where data can have multiple personalities including POSIX, Object, and HDFS (Hadoop Distributed File System). Currently, these object storage systems are immature and don't support near-POSIX interfaces. MarFS assumes you want a first class near-POSIX interface to your files. MarFS is trying to be the best of both worlds, allowing data scaling like an object storage system, metadata scaling like N POSIX name spaces, and both kinds of access to the same data, the true "sea of data" concept. In time, it is certainly possible that they will fill MarFS's role.

It is possible to put object storage systems under scalable file systems like GPFS (General Parallel File System) using a block interface over the object storage system, but the block write patterns of these PFSes (parallel file systems) are not well suited to benefit from these object storage systems' high performance. MarFS will be able to use any object storage system, including potentially using cloud-based services, as a back end storage repo.

The team has investigated existing open source projects, and there doesn't appear to be another one that provides the needed functionality. Ceph provides a file system on objects, but isn't known for scaled out metadata service. GlusterFS is probably the closest thing to an alternative, and indeed GlusterFS can be a global name space combining multiple file systems into one mount point. It also hashes the file names across the file systems, which is something MarFS is not currently designed to do. The main difference is the approach to what GlusterFS documentation refers to as unified file and object. GlusterFS has been integrated to be object storage for OpenStack Swift (for objects) and block storage for

OpenStack Cinder (for blocks). Conversely, MarFS is designed to put a near-POSIX interface over any object storage system, including OpenStack Swift.

Some may question why a HSM (hierarchical storage management) tool like HPSS (High Performance Storage System) or DMF (Data Migration Facility) is not used. These systems currently don't take advantage of the enormous industry investments in object storage. HPSS metadata performance is likely 1/10th or less of what MarFS metadata performance is expected to be. MarFS will leverage existing tools and be a small amount of code to combine these tools. To be fair, MarFS is not a HSM, although batch utilities could be written to move data around under MarFS to various kinds of storage systems that would be most appropriate for the data based on policy configurations. HSM systems, like HPSS, are not generally highly parallel. MarFS is designed for dozens to hundreds of metadata servers/name spaces and thousands or even tens of thousands of parallel data movement streams. HPSS is designed for about an order of magnitude less parallelism. There are some solutions emerging in the space of object systems back-ending DMAPI (Data Management API), particularly for GPFS like the DDN (Data Direct Networks) DMAPI to WOS (Web Object Scaler, http://www.ddn.com/products/object-storage-web-object-scaler-wos/) solution. The way DMAPI works is just quite heavy in that it tries to handle every POSIX case. MarFS has the principal of simplifying and not supporting some use cases in POSIX to accommodate easy/friendly use of Object Stores of all kinds. Of course you could implement a DMAPI back end that doesn't handle all POSIX cases, but you can't be guaranteed you will not see these requests unless you control the access tools people use or put a FUSE (File System in User Space) in front of GPFS to control the use cases (like update in place). Ultimately this simpler and more limiting model in MarFS does not mate well with DMAPI. The simpler goal for MarFS was chosen to allow for extreme flexibility for implementers.

There are products that are optimized for WAN and HSM metadata rates. For example, General Atomics Nirvana Storage Resource Broker, iRODS (Integrated Rule Oriented Data Systems). There are some capabilities for putting POSIX files over objects, but these methods are largely via NFS or other methods that try to mimic full file system semantics including update in place. These methods are not designed for massive parallelism in a single file, etc.

The team has looked at name space solutions. EMC's Maginatics is in its infancy and targeted at enterprise. An open source name space project called Camlistore appears to be targeted and personal storage. Bridgestore is a POSIX name space over objects, but it puts its metadata in a flat space so rename of a directory is horribly painful. Avere NFS over objects is focused at NFS so shared file N-1 will not be high performance.

We need an open source solution to deploy in production now that enables the described functionality. It is our hope that MarFS will set the bar high for fully integrated solutions to replace it.

Requirements and Design

This section defines the requirements and design elements that were crafted to meet the requirements.

Design Overview

This design will require:

- Linux system(s) with C/C++ and FUSE support
- MPI (Message Passing Interface) for parallel communication in pftool (a parallel data transfer tool, see https://github.com/pftool/pftool). Thus, most any MPI library with a C interface can be used.
- Communications with the MPI library can utilize many communications methods like TCP/IP, Infiniband OFED, etc.
- If you plan to use MarFS only to combine multiple POSIX file systems into one mount point, any set of POSIX file systems can be used.
- If you plan on using multi-node parallelism for the FUSE daemon, pftool, or the batch utility programs (MarFS software), all file systems, including MarFS file systems, must be globally mounted on all nodes running MarFS software. This includes NFS and other global file systems.
- If you plan to store data on an object store, that object store needs to be accessible by all nodes running MarFS software. The MarFS metadata component must be capable of POSIX extended attributes (xattr) and must support sparse files (files that have a non-zero size but that occupy no space).

The planned MarFS implementation will use GPFS file systems as the metadata component and Scality and/or ECS ViPR object stores as the data storage component. Of course, the data storage component can be one or more POSIX file systems. The data storage component should be selected to provide the best performance for the type of files that will be stored on it.

The interactions with the GPFS-based metadata component are via the normal POSIX interface. GPFS has some ILM (Information Lifecycle Management) capabilities for managing massive amounts of metadata that helps immensely with batch processing for management of the system.

The interactions with the Scality- and/or ECS ViPR-based data storage component are via the most efficient object protocols, such as Amazon S3 and CDMI. MarFS can put a file per object, pack many small files into one object, and spread a large file across many objects. Although the design does not call for using POSIX file systems as the data storage component, the design does not preclude it. If MarFS were configured to use a POSIX file system as the data storage component, then any such file system would work including PFSes (parallel file systems) like GPFS, Lustre, Panasas, etc., or non-PFSes, like NFSv3.

The GPFS file systems and object stores will be hidden from the users so that they cannot use them directly, but the MarFS components will know about them and how to use them efficiently.

FUSE Daemon for Interactive Use

A FUSE daemon will provide the system mount point and interactive use component of a MarFS file system. Of course there can be multiple MarFS file systems and consequently multiple FUSE daemons. This daemon will know that it will use the GPFS file systems for metadata operations and the specified object stores as the data storage component. The FUSE daemon on the interactive FTA nodes allows users to run interactive file system commands but the FUSE daemon has some drawbacks. It cannot pack multiple small files into one object. A utility program must do this after the FUSE daemon writes multiple small files. The FUSE daemon enforces writing only serially from byte zero (e.g. there is no update-in-place). This means if you want to update a file in place you need to copy it to a full service file system, modify it and put it back. Files can be read in any order of course and all metadata operations should work (chown, chmod, mkdir, etc.). If the file is stored on an object server that does not support update in place you can only truncate to zero, meaning files have to be completely overwritten, not partially. Currently, append is not supported, but that could be added at some point.

pftool for High Performance Parallel Data Movement

The parallel data movement utility, pftool, will likewise be modified to use the GPFS file systems as the metadata component and the object stores as the data storage component. pftool is a load balanced, highly parallel utility on one node or across multiple nodes. It can walk the file system tree in parallel, move data between file systems, and move small files in parallel or break-up big files to move them in parallel for any POSIX file system, including MarFS. It will be possible to write data to MarFS in parallel using pftool, or by writing one's own parallel data movement utility using the library. In our design, pftool will run on the batch (non-interactive) FTAs for performance and security reasons. Access to the object store needs to be controlled such that the FUSE daemon and pftool can access it on behalf of the users, but users cannot access it directly. pftool provides pcp (parallel copy), pls (parallel ls), and pcmp (parallel compare).

Utility Programs to Manage MarFS

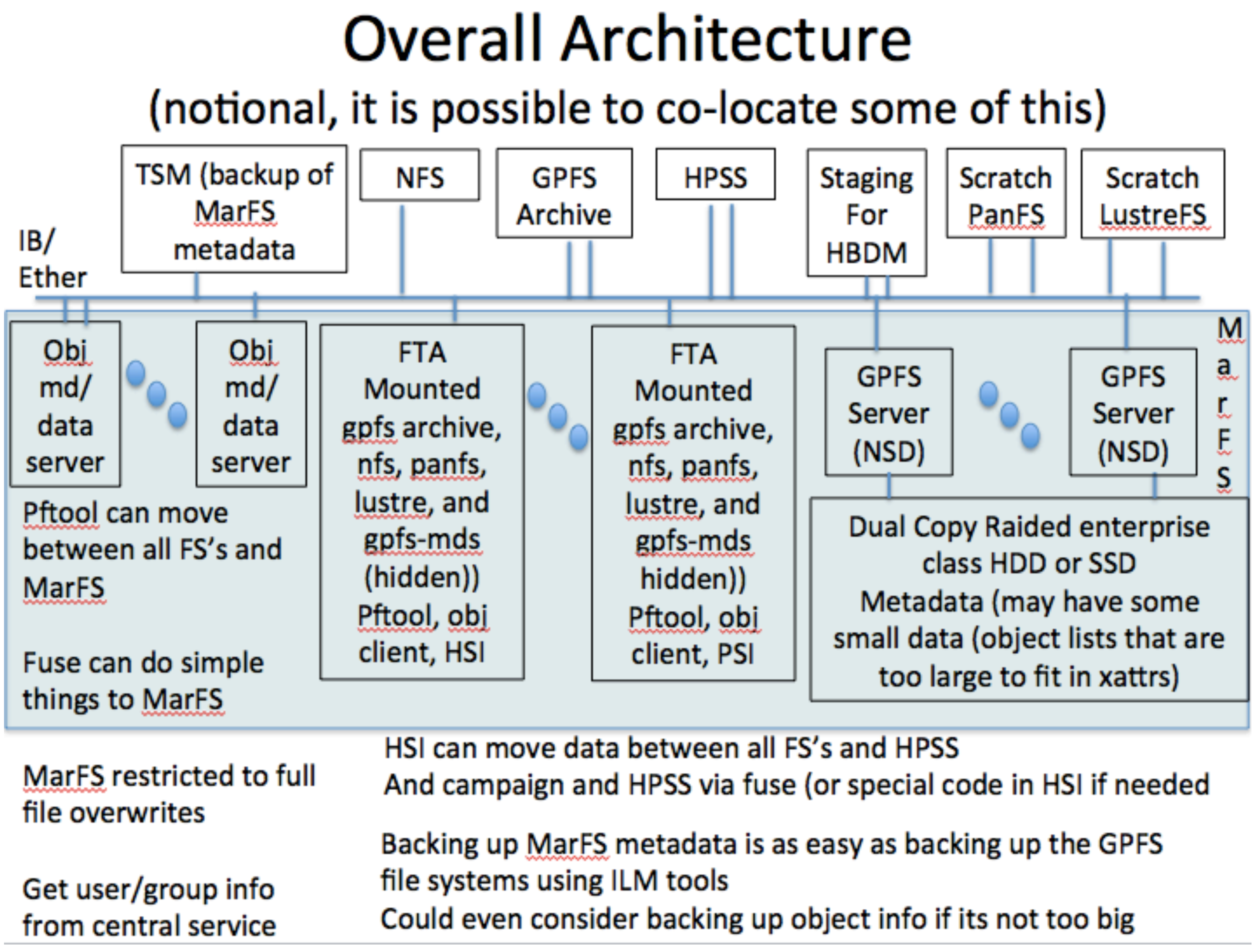
There will be some MarFS utility programs that will be run periodically to free deleted storage space and ensure that users do not exceed their assigned quotas. Other utility programs may be implemented in the future to manage other aspects of the file system that can be performed on an as-needed or periodic basis; such as for packing small files into an object of the best size for the data component.

Since our design uses GPFS as the metadata component, the GPFS ILM features will be utilized for very fast inode shadow table scanning, threaded name merges, etc. to look through millions of files in minutes to perform these management tasks. In the future, these scans could gather information needed for statvfs/statfs as well as other useful histogram information. For example, files/space, file sizes, based on dates such as files created or modified in the last X days, file types (e.g. Multi, Uni, Packed), directories, etc.; basically any information that the administrators find useful. One might want to exclude trash or account for trash separately, perhaps by walking the trashdir and subtracting that space or the like.

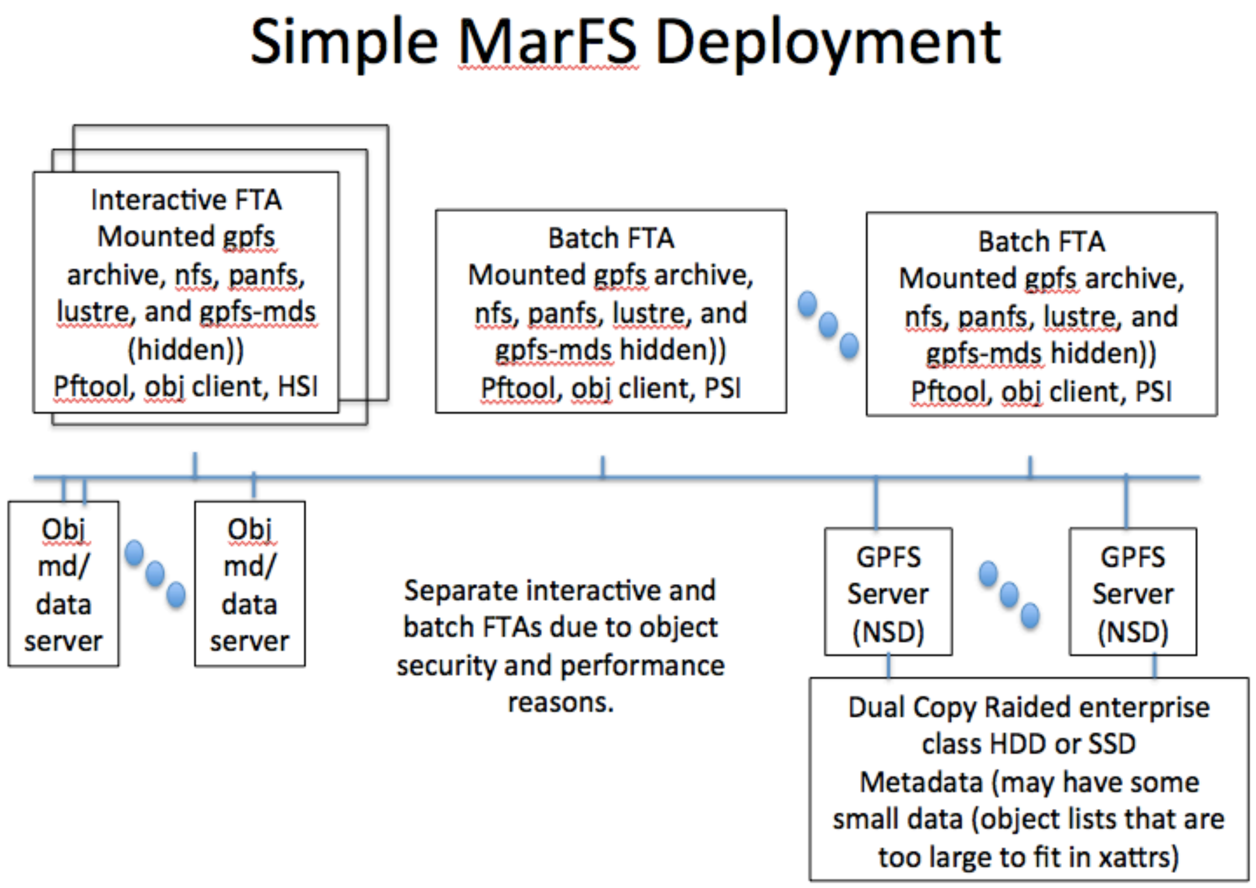
A way to list objects used in the object repositories might be nice. By this manner an infrequent pass through the data repositories can be made to remove objects that are no longer used.

Planned LANL Deployment

Following is a diagram that depicts a very rough concept of how MarFS campaign storage would fit into LANL's Turquoise network environment. One sees all the normal services on the top and the MarFS campaign storage on the bottom. The FTAs (File Transfer Agents) are where the FUSE and utility programs run and where the POSIX file systems (metadata component) are mounted, as well as all the other non-MarFS file systems (NFS, GPFS Archive, Scratch, etc.). The MarFS metadata component (in this case GPFS NSD) provides very fast parallel metadata. The data component (in this case object servers) provides a scalable data repository for MarFS files.


Overall Architecture
(notional, it is possible to co-locate some of this)
TSM (backup of MarFS metadata
NFS
GPFS Archive
HPSS
Staging For HBDM
Scratch PanFS
Scratch LustreFS
IB/ Ether
Obj md/ data server
Obj md/ data server
FTA Mounted gpfs archive, nfs, panfs, lustre, and gpfs-mds (hidden)) Pftool, obj client, HSI
FTA Mounted gpfs archive, nfs, panfs, lustre, and gpfs-mds hidden)) Pftool, obj client, PSI
GPFS Server (NSD)
GPFS Server (NSD)
MarFS
Pftool can move between all FS's and MarFS
Fuse can do simple things to MarFS
Dual Copy Raided enterprise class HDD or SSD Metadata (may have some small data (object lists that are too large to fit in xattrs)
MarFS restricted to full file overwrites
Get user/group info from central service
HSI can move data between all FS's and HPSS
And campaign and HPSS via fuse (or special code in HSI if needed
Backing up MarFS metadata is as easy as backing up the GPFS file systems using ILM tools
Could even consider backing up object info if its not too big

A closer look the recommended MarFS deployment follows. Notice that separate interactive and batch FTAs are recommended for security and performance reasons. Since the object security model does not match well with the POSIX security model, it makes sense not to allow users to have access to the keys needed to open/control the object stores. Thus, those keys should be kept on the batch FTAs and provided only to the FUSE daemons on the interactive FTAs using some secure method. The ability to access the object servers directly without going through the FUSE daemon must be controlled. Interactive FTAs would be used for "ls", "grep", "tar", etc. These are all small/serial tools that can run by as users gaining access to files through the FUSE daemon, which must run as a privileged user for a variety of management and security reasons.


Simple MarFS Deployment
Interactive FTA Mounted gpfs archive, nfs, panfs, lustre, and gpfs-mds (hidden)) Pftool, obj client, HSI
Batch FTA Mounted gpfs archive, nfs, panfs, lustre, and gpfs-mds hidden)) Pftool, obj client, PSI
Batch FTA Mounted gpfs archive, nfs, panfs, lustre, and gpfs-mds hidden)) Pftool, obj client, PSI
Obj md/ data server
Obj md/ data server
Separate interactive and batch FTAs due to object security and performance reasons.
GPFS Server (NSD)
GPFS Server (NSD)
Dual Copy Raided enterprise class HDD or SSD Metadata (may have some small data (object lists that are too large to fit in xattrs)

MarFS files are created into a POSIX metadata file system with no special metadata. The MarFS file metadata will be placed into one of many possible POSIX file systems that comprise the metadata store. In this way, MarFS is a global name space. Files are not hashed across POSIX file systems. Rather, the total MarFS namespace is decomposed into separate POSIX file systems by tree. All normal POSIX ownerships/permission/attributes like dates/sizes/etc. are obeyed. One can even add user supplied xattrs to the files.

The data is written to data component, which can be a POSIX file system or object store. All normal attributes are kept up to date in semantically reasonable ways like permissions, dates, and even file size. The file size is updated by truncating the POSIX metadata file to the size of the desired file even though there may be no actual data in the file itself. For this reason the POSIX file systems used for the metadata component must support sparse files. Further, some reserved POSIX xattrs are applied to the metadata component files to provide information so that where the data is and how to access it are preserved. For this reason, the POSIX file system used for the metadata component must support POSIX xattrs, where some xattr names are reserved and hidden from the user.

In this design all metadata operations like reading/creating directories, managing ownership, dates, permissions, user xattrs, etc. are all just performed on the POSIX file systems being used as the metadata component. Only operations involving space management for files on the data component, where said component includes object store(s), need to be handled differently from normal POSIX. In these operations, all the POSIX permissions are adhered to and additionally object permissions are also obeyed based on the operation. Creating, writing, and reading these files with their data in object store(s) uses/updates the POSIX attrs and xattrs appropriately to manage the access. For special space management operations like truncate and unlink, all references to the space being freed are renamed (in case of an unlink) or copied (in the case of a truncate) to a trash directory that can be used by a utility program for space reclamation from the object server. It does require a utility program to free the objects and clean out the trash. A trash recover utility could easily be written as well if desired.

Flexible Configuration

MarFS shall allow for use of one or more POSIX-compliant file systems as the metadata component.

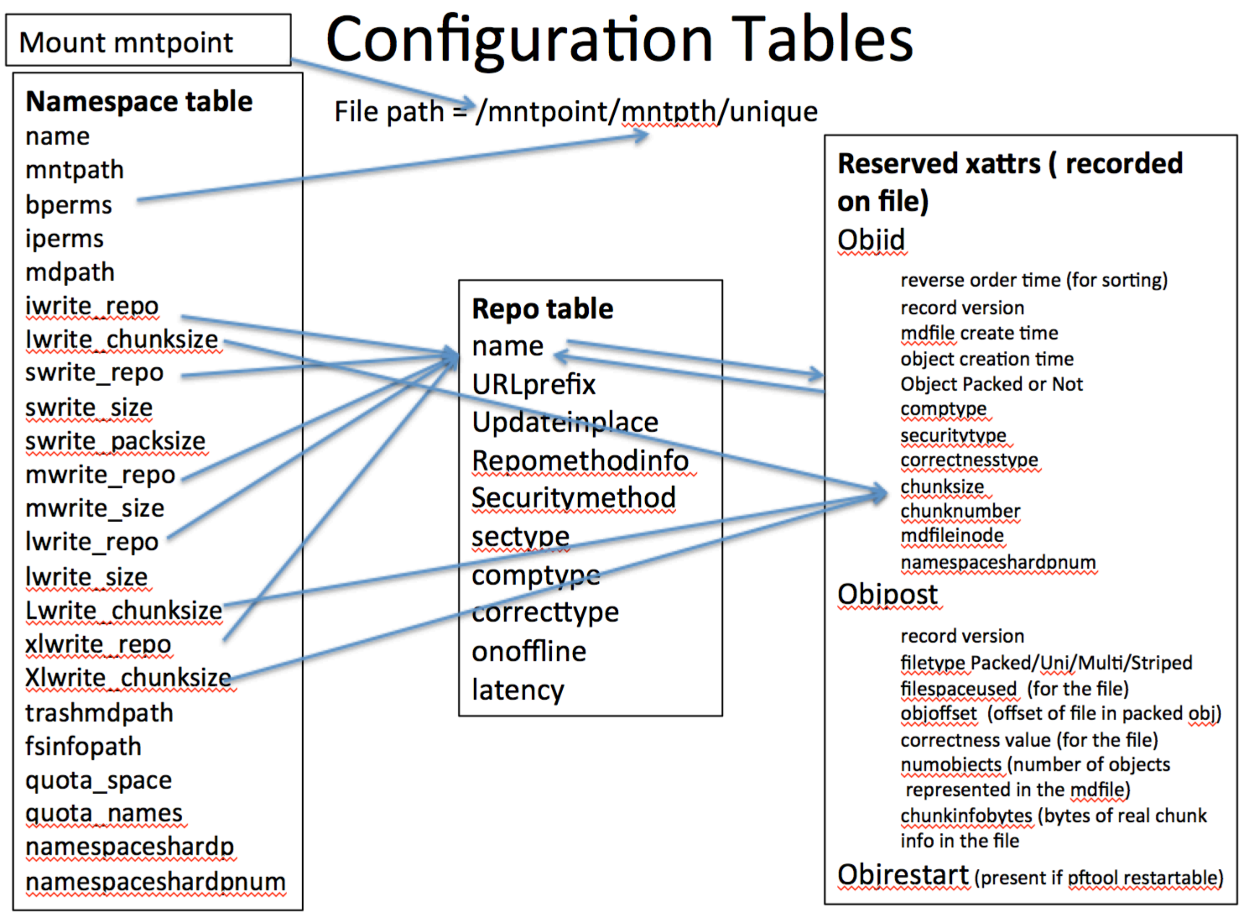
MarFS shall allow for use of one or more file system or object store as the data component.

MarFS shall allow for defining the parameters for the metadata and data component file systems and object stores to meet the needs of a given installation.

Design to Provide Flexible Configuration

The key to a MarFS installation is to understand the configuration information. A given MarFS instance is defined by describing the metadata component file systems and the data component file systems and object stores in terms of how to access them and the parameters for their use.

The subsequent figure shows the information that defines a MarFS instance. The purpose of these parameters follows the figure.



MarFS has a single mount point.

MAR_mnttop	This is the top-level directory under which all namespaces are placed. Specified as a path with slashes.

The FUSE daemon, pftool, and the utility programs will append MAR_mnttop on the front of all the namespace segments to construct a namespace tree. For example:

MAR_mnttop = /redsea
MAR_namespace.mntpath = /projecta
MAR_namespace.mdpath = /md/projecta

The user references /redsea/projecta and that refers to file's metadata file system, or namespace, in /md/projecta

MAR_namespace entries define the one or more namespaces that are supported under the MarFS mount point.

name	Name that refers to the namespace used in the objid that is

	stored as the name of the object in the data component.
mntpath	Specifies the path for this namespace, which is appended to the MAR_mnttop. It is specified as a path with slashes.
bperms	Specifies permissions for utility programs. These permissions are above and beyond the POSIX permissions (rwx/ugo). This is because external repositories may have special permissions that don't map exactly to POSIX permissions. The values are rmwmrdwdudtd. rm – read metadata wm – write metadata rd – read data wd – write data ud – unlink data td – truncate data An example of interesting use is to allow read and write in POSIX permissions, allow metadata changes but not allow writing of data. This value is not stored with the file, it is interpreted real time, so this is a fast way to shut of write of data or metadata etc. This item can change based on allowed activity against this namespace and the data/space it represents.
Iperms	Specifies permissions for access through the FUSE daemon. Same as bperms above.
Mdpath	Specifies the path for the POSIX file system that is to hold the metadata and potentially data for this namespace. If a file is to be written to an external repository, then only metadata is stored in this file system, but if data is to be stored into this file system then both data and metadata are used. Controlling whether data is written into the metadata file system is done in the repository configuration table using the repomethodinfo field so the repository to/from which one is writing/reading will be DIRECT (use the metadata file system) or some other external method like CDMI, S3, etc. This is specified using path notation using slashes. This can change if you have moved the metadata file system path for some reason. Though it might be hard to change on the fly.
Iwrite_repo	Specifies to which repository FUSE daemon accesses will write new files and points at a name in the repository table. This can be changed as it controls to where new files are written.
Iwrite_chunksize	Chunksize for FUSE daemon accesses.
Swrite_repo	Specifies to which repository utility programs will write new small files and points at a name in the repository table. This can be changed as it controls to where new files are written.
Swrite_size	Size below which is considered a small file. This can be changed as it controls to where new files are written.

Swrite_packsize	Size of object into which to pack multiple small files. If this value is zero then packing will not occur.
Mwrite_repo	Specifies to which repository utility programs will write new medium files and points at a name in the repository table. This can be changed as it controls to where new files are written.
Mwrite_size	Size below which is considered a medium file. This can be changed as it controls to where new files are written.
Lwrite_repo	Specifies to which repository utility programs will write new large files and points at a name in the repository table. This can be changed as it just controls where new files are written.
Lwrite_size	Size below which is considered a large file. This can be changed as it controls to where new files are written.
Lwrite_chunksize	Chunksize for large files.
xlwrite_repo	Specifies to which repository utility programs will write new xlarge files and points at a name in the repository table. This can be changed as it controls to where new files are written.
Xlwrite_chunksize	Chunk size for xlarge files.
trashmdpath	Specifies where in the namespace, information is stored on unlink and trunc/ftrunc operations, which could provide a trashcan function but is used by utility programs for reclaiming space, repacking, reconciliation of space which is needed for external repositories. All permanent deletion of data (both unlink and trunc) is done in batch for external repositories. For "DIRECT" repositories where the data is stored directly in the metadata file, unlink operations go to this path, but trunc'd space is not preserved. This is specified as a path with slashes. It is assumed that this is in the same metadata file system as the metadata file system for this namespace, as rename is used for unlink operations. This value could change but much care would have to be taken because entries into this path can be occurring all the time and information about reclaimable space lives in this path.
fsinfopath	This is a path name specified with slashes to a file that contains the values one would get in a statfs/statvfs call like how much space is in the file system, how much space is used, etc. This file must be updated in a lazy way via periodic batch scans of inode space etc. Since the space for the files in a namespace may not be in the metadata file system associated with a name space, it is required that this info be provided in some way to be chosen by the site. It could involve walking the metadata tree or inode space and adding up spaced used or it could involve querying an external repository for space etc. This value could be changed, but care needs to be taken, as statfs/statvfs calls will look in this file for providing information.
quota_space	Specifies the space quota for this name space. This value is

	compared to information in the fsinfopath file above about how much space has been used which is populated via lazy batch runs to determine and record space used. This can be changed at any time, but will not take effect immediately as quota's are done in a lazy way based on batch runs to update the fsinfopath file.
quota_name	Specifies the inode quota for this name space. This value is compared to information in the fsinfopath file above about how many inodes have been used which is populated via lazy batch runs to determine and record inodes used. This can be changed at any time, but will not take effect immediately as quotas are done in a lazy way based on batch runs to update the fsinfopath file.
namespaceshardp	Path to namespace shard metadata file systems
namespaceshardpnum	Max number of namespace shard metadata file systems to hash across

MAR_datarepo defines the one or more data repositories for each namespace. There is one of these for every repository that is referenced in the above namespace table, and for every repository that any file stored anywhere in this MarFS instance. The only way to know if you can get rid of a data repository in this list is to ensure no references exist in both the configuration namespaces and in the metadata for all the name spaces. It is really recommended that you don't delete anything, just add another row with a new repository. A repository is just a logical name that connects the data of files in any namespace to a particular use of place to store the file data. It is possible for multiple repositories to point at a single external object storage server with different characteristics like compress and don't compress etc. Repositories represent a method for talking to some back end store.

name	Name for this repository, this name is used in the namespace table above in the config file and it is also used stored with the file in xattr, so this can not be changed easily. It follows the same rules as deleting a repo in this list. This name should point at some name of a portion of an object repository. For example with Scality sproxyd access method, this repo name would match to a stanza in the sproxyd file which tells the object system the storage format for this repo name/stanza (like 30+6 or 40+8 along with other attributes).
URLprefix	This is a string associated with the repository used to access the repository. Object names will be repository URLprefex/bucket/object name Or really, URLprefex/namespace.repo.suffix/obj name, which is formed and stored in the MAR_objid xattr.
updateinplace	Updates in place for files in this repository are allowed. This lets you decide if a file is in a repo that can do update in place then the FUSE daemon and utility programs can allow update in place.

	If a repository doesn't allow this easily then you can forbid it. It is probably good practice to not allow this for all repositories used in a namespace but you don't have to do that. Update in place means that if you open for write, you have to overwrite the entire file from the beginning. It also means that you cannot truncate the file to any other value than zero. It also means that you cannot open with append and append to the file, although this capability might be changed at a later date. The software can use update in place for DIRECT as the repomethodinfo (which tells the software to put the file data in the metadata file). Values are yes/no. This can be changed, but it is not recommended.
repomethodinfo	Info about method for accessing the object repository, like S3 or CDMI or DIRECT means (use the metadata file system for the user data) .
securitymethod	Specifies a method for how security works on this repository (authentication/authorization), this can change as it is not recorded anywhere other than in this file but any backend storage system must be kept in sync with this method.
sectype	Specifies a method for encryption for data for the repository. This CANNOT change as all files that have data in this repository are encrypted with this type.
comptype	Specifies a method for compression for data for the repository. This CANNOT change as all files that have data in this repository are compressed with this type.
correcttype	Specifies a method for correctness for data for the repository. This CANNOT change as all files that have data in this repository have this correction information calculated and stored with this type.
onoffline	Specifies a method for bringing a repository online if the repository is of the type that allows it to be offline. This value can change as it is dynamic.
latency	Specifies a time it might take to bring a repository online. This value can change as it is dynamic.

For Uni and Packed files, the MAR_objid xattr holds the object id for these files. The MAR_post xattr holds the type of file (Uni, Packed), space used (for the file), correctness value (for the file), and numobjects which for Uni and Packed files will be zero.

For Multi object files, the MAR_objid xattr holds the object id for these files except the chunknumber changes based on where you are at in the file. The MAR_post xattr holds the type of file (Multi), space used (for the file total), correctness value (for the file total), and numobjects which for multi files will be the number of chunks in the first part of the metadata file that contain chunk information in them and the chunkinfobytes is the number of bytes of chunk info in the metadata file. In the case of a Multi file, the chunk information is stored in the metadata file, which implies that the chunksize for a Multi file must be larger than the space used, correctness info, and chunknumber fields.

The format for the objid info and post info in this metadata file is the chunknumber concatenated with the space used and correctness info for that chunk. These concatenated things with appropriate per chunk information are repeated for each object in the multipart object in order of offset into the file. NOTE: These chunknumber/space used/correctness values do not have to be inserted in chunknumber order, as out of order writing is allowed, but we plan to use a fixed record size, so you can take the chunk number and with math derive the file offset for the info for that chunk. So objects can be written to, but they are not officially in the file until this information is added to the metadata file and MAR_post xattr is updated appropriately. This activity records the chunks and sizes/correctness information for each chunk into the metadata file. This information is valuable so we know when objects are ready to be associated with a Multi file (for restarting etc.), how much compression was achieved, and lets you keep checksums/crc per object, etc. It may or may not be consulted during a read operation, but it is on a write.

Near-POSIX Interface

MarFS shall provide a near-POSIX interface.

MarFS shall provide a POSIX mount through which the user executes the supported file functionality.

MarFS is not required to allow users to update files in place for data repositories where update-in-place is not easy, like object stores.

MarFS is not required to provide an object interface to the data. This includes not being required to provide an HDFS interface.

MarFS is not required to provide file locking.

MarFS is not required to provide hard links, but shall provide symbolic links.

MarFS is not required to provide mmap and application execution, be a PFS, or a parallel archive.

Design to Provide Near-POSIX Interface

If you are looking for a way to provide a near-POSIX file system interface over multiple POSIX file systems or over one or multiple Object Storage Systems as the data storage component, MarFS might be the answer. MarFS can use one or more POSIX file systems to hold file system metadata. The FUSE daemon provides nearly full POSIX access with a few exceptions that are specifically discussed.

Whether using interactive Linux commands in a shell or pftool, the interface to MarFS will be through a near-POSIX interface that targets what looks like a normal POSIX mount point.

pftool safely allows multiple writers to write to a single file as well as multiple readers to read a file in parallel, but it does not protect against a user using different commands/programs from updating the same file concurrently.

All programs shall work unchanged except for programs that seek around in the file and write, or append to the end of a file, or try to truncate a file in a place other than zero offset. This means that reading files will work pretty much no matter what, but writing has to be done as a complete overwrite.

MarFS could eventually provide an object interface to data, as its metadata has object information and maps POSIX files onto Objects. With respect to a HDFS interface, MarFS is POSIX and HDFS does have the ability to use POSIX files. Optimization to provide layout information to an HDFS layer is possible.

MarFS provides symbolic links through its use of GPFS as the metadata component. Hard links are not supported because the GPFS metadata component may actually be multiple name spaces and/or GPFS file systems and hard links cannot be used across name spaces and file systems.

MarFS is primarily intended as a file system for large data collections and not for application execution. That said, mmap or execution might work if it behaves relatively well. One should be able to mmap and execute off of MarFS, but mmap writing may not work if writing is not serial. MarFS is not intended to replace a PFS, as it lacks important features on purpose, although it might suffice as a PFS in some settings. Likewise, MarFS is not intended to replace deep and/or parallel archives, such as HPSS, although in some settings it might also work for this function.

Scalability

MarFS shall provide a means to scale metadata handling as more capacity and file count is added. This scalability shall target the common use case scenarios for large HPC storage systems where there are many clients.

MarFS shall provide a means to scale data handling as more capacity and file count is added. This scalability shall target the common use case scenarios for large HPC storage systems where there are many clients.

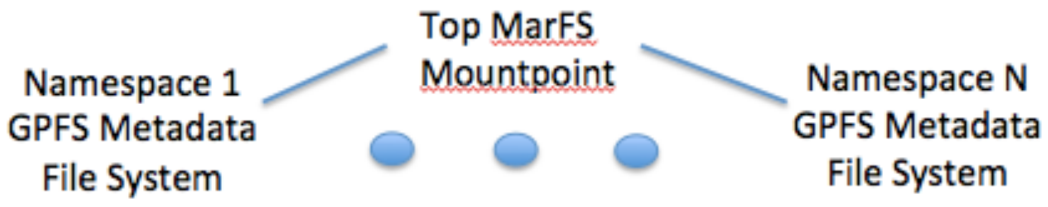
MarFS is not required to solve the scalabilty problem of very large single directories.

MarFS shall use data structures and techniques that as close to constant in execution time as possible as the file system size and file attributes increase.

Design to Provide Scalability

MarFS currently is concentrating on providing a best-in-class scalable metadata service over a best-in-class scale out object storage system. If you are looking for a way to scale metadata service, but not stripe or hash metadata, MarFS might be the answer.

As mentioned, MarFS will use GPFS as the metadata component. GPFS has many features in its ILM component that allows GPFS to be used as an efficient and scalable MarFS metadata server. MarFS will aggregate many metadata file systems together to create one, large logical file systems namespace. See the subsequent figure. In this implementation there is no scaling within a single directory. This capability will be considered as a future enhancement and is discussed later in this section.


Top MarFS Mountpoint
Namespace 1 GPFS Metadata File System
Namespace N GPFS Metadata File System

MarFS utilizes POSIX extended attributes (xattr) in its metadata component to place information about the data repository/objects that hold the data for files. Files are created without extended attributes and acquire them when the file is written. Xattrs will only exist for files in which the data exists in external data repositories. Therefore, most any POSIX file system can be used for holding metadata and data, but the requirements for storing metadata for files in which data exists in external data repositories requires xattr and sparse capabilities because xattrs are used to map to the external data repository and the POSIX size field is used to store the length of the file for files where the data lives in an external repository. GPFS's ILM capability is able to use xattrs to scalably and efficiently find files that match a specified criterion.

Here is a high level description of the reserved MarFS xattrs:

- MAR_objid is the object name (or the first object name in the case of a multi object file). This info doesn't change very often, basically only on a truncate. So it is set in stone for a file unless you throw out all the space associated with the file.
- MAR_post is information that can change from time to time. For example, it can be updated while the file is growing, etc.
- MAR_restart is used by restart on multi-files and thus won't be present very often.
- MAR_namespace_shard is used on directories for directory hashing number (future).

A couple of the xattrs, MAR_objid and MAR_post are one record each with many fields concatenated. The reasons for this are:

- These are all variables that are short enough to concatenate into one record to form the object name to help the admin figure out things if there are issues. Having this information in the name of the object will be handy.
- It makes the object name unique.
- It doesn't add information that is too long for the name of the object, like create time path of the metadata file.
- It takes time to insert xattrs, so making this be 10ish xattrs is inefficient and doesn't scale on a per file create or file read basis.

The MAR_objid is the basis of the name of the object for this file. Its full name is /bucketname/MAR_objid. The bucketname is /namespace.repo.suffix from the configuration file. So, the fully qualified object name is /namespace.repo.suffix/MAR_objid.

The MAR_objid xattr concatenated fields are:

Bucket (namespace.repo.suffix)	This is the object system bucket name or another way to match to some portion of a repository on the object system.
Reverseorder time stamp	This is a reverse order time stamp of some kind to make sorting easy when listing objects.
recordvervion	Version number for this record.
mdfilecreatetime	In the case of Uni and Multi files, this is the creation time from the metadata file. In the case of Packed files, this is the creation time of the first file being packed into the object.
Objcreatetime	This field is used to put current time stamp, in addition to the metadata file create time. This is used to "version" objects. For example, on a truncate to zero, which would put all the objects for that file into the trash, the names will be the same as create time and inode remains unchanged. This makes a unique name for the new objects for that file but yet they are still related by all fields except this one.
objtype	Packed if many files are being packed into the object or Not packed if not.
comptype	Compression type (future).
sectype	Security type (encryption, future).
correcttype	Correctness type (crc/checksum, future).
objchnksz	The size of a write in all objects, but the last, for Multi file. This value is initially populated from the repository configuration table for the file based on namespace/path. Chunk size is picked based on whether the file writing is interactive or batch. For batch, it is based on the size of the file being moved, (large, xlarge) and taken from the configuration file.
objchnknumber	If this is non-zero, this object is part of a Multi file. For Uni and Packed files there is only one object and it will be numbered zero. For a Multi file, the object name only changes by chunk number, which is calculated based on chunksize. So this is really just a placeholder value.
mdinode	Inode of the metadata file. For a Packed file it is the inode of the first file in the object.
namespaceshardpnum	Used for namespace shard number for files that are hashed into multiple directories (future). For now this is zero.

MARpost is information written to the file's xattr after or while the object(s) are written. Its concatenated fields are:

recordversion	Version number for this record.
objtype	Records how the file data is stored in an external object repo, there are currently 4 types: Uni – one object stores the entire file. Multi – a file is spread across multiple objects using chunk sized objects, object id's are recorded in the metadata file. Packed – multiple files in each object that require using the objoffset field. Striped – a file is round robin striped across multiple objects using chunksize from the configuration file. Object ids are recorded in metadata file.
spaceused	Space used in the object system for the entire file. May have to sum multiple object space used for a Multi file.
objoffset	Records offset into object where file data is. This is only used for a Packed files
correctnessvalue	Checksum or CRC for the entire file. May have to sum multiple checksums or CRC's for a Multi file (future).
numobjects	For Multi files, this records the number of objects in the metadata file that contain chunk information. The rest of the file is a sparse file to make the size of the metadata file equal to the size of the file.
Chunkinfobytes	Number of bytes of chunk info in the file.

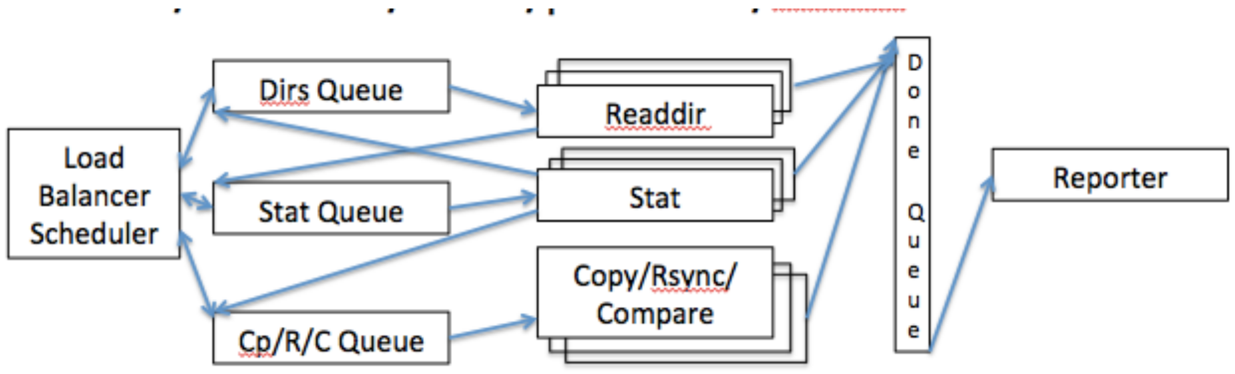
MAR_restart is a xattr that is used by pftool to indicate whether or not a complete copy was complete before pftool exited. If it was not the next invocation of pftool on this file will start copying this file again. This is only used for a Multi file.

MAR_namespace_shard is a xattr on directories that are hashed (future).

The design uses scale out data services, via the object stores, separately. Data and data movement can scale as N file systems or N object stores and it has features to be "friendly" to object systems by trying to form large multi-megabyte sized objects for efficient storage and tracking, including packing multiple small files into single objects that are sized for efficient handling by the object stores.

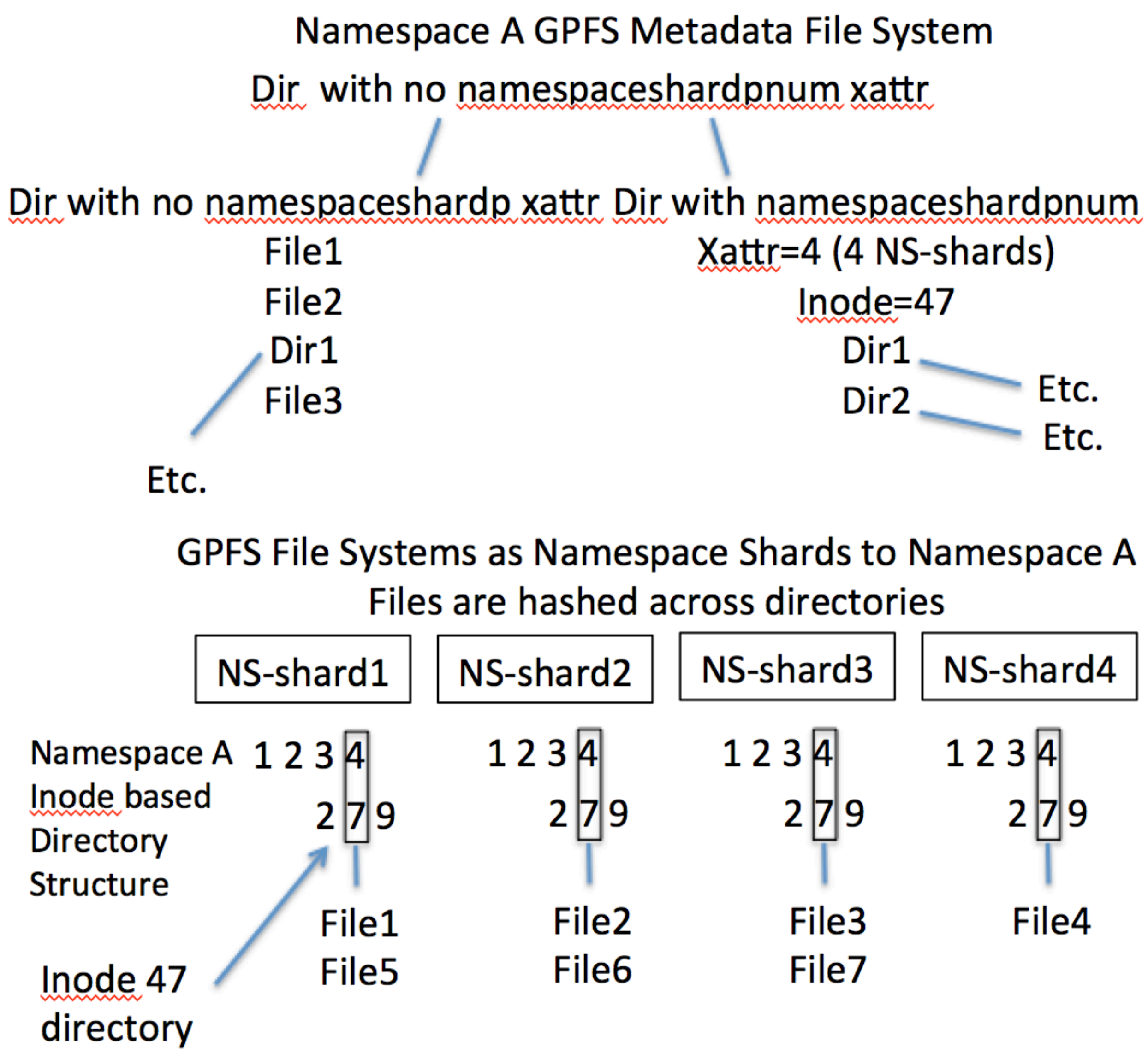
System administrators need to set up buckets that are listed in the configuration files on the object server(s) for every namespace/repository combination.

MarFS will use pftool as the workhorse to move large data collections to and from MarFS. pftool can walk the file system tree in parallel and write as many streams of objects as you have mover processors at your disposal, up to the limit of MPI (tens of thousands). It dynamically balances the data movement load amongst its worker processes. It breaks up large data elements and coalesces small data elements into larger ones to maximize throughput and minimize overhead. Here is a logical depiction of how pftool works to do high-performance, parallel data movement.



Currently the MarFS design does not hash or split-up single directories to address the problem of single directories containing a large number of files. It is possible for an implementation to use a metadata file system under MarFS that does hash or split-up single directories to address the scalability of directories with many files. And, future MarFS features might include hashed or split-up single directories. It is possible to hash file names (not directory names) across several namespace shard directories that use the main namespace directory inode number as a directory name on namespace shard file systems for hashing files across. See the next figure. To implement this capability would require:

- Using xattr on directories and that those xattrs be cached reasonably well across parallel client nodes.
- A Namespace Shard Directory Structure using Namespace A Directory inodes.
- Hashing files only (not directories) across namespace shards.
- Threading mkdir and metadata operations against directories, including file renaming.
- Making listing, reclaiming, etc. parallel (like map reduce).
- Scaling within a single directory to N GPFS file system directories.
- Renaming directories continues to work.
- Not requiring communications protocols, just using mounts.
- Pftool ability to set width of hash up to a max.


Namespace A GPFS Metadata File System
Dir with no namespaceshardpnum xattr
Dir with no namespaceshardp xattr
File1
File2
Dir1
File3
Etc.
Dir with namespaceshardpnum
Xattr=4 (4 NS-shards)
Inode=47
Dir1
Dir2
Etc.
Etc.
GPFS File Systems as Namespace Shards to Namespace A
Files are hashed across directories
NS-shard1
NS-shard2
NS-shard3
NS-shard4
Namespace A Inode based Directory Structure
1 2 3 4
2 7 9
File1
File5
File2
File6
File3
File7
File4
Inode 47 directory

PFS-like Performance and Eventual Consistency

This requirement is related to the near-POSIX interface. Maintaining locks, collecting released space, and enforcing quotas are things that a POSIX-compliant file system does, but that affect performance. MarFS is not required to provide these POSIX file system features.

MarFS, like other PFSes, is not required to check/lock to protect against multiple non-coordinated writers into the same file.

MarFS shall ensure that the file system's metadata and data are eventually consistent.

Design to Provide PFS-like Performance and Eventual Consistency

The specified operations are not necessary to be done in real-time, as they would adversely affect MarFS performance. Utilities that can be run periodically will be provided to reclaim deleted space and ensure that users do not severely overrun their quotas for storage space and file count.

MarFS could provide file locking, but does not currently do so. Like many PFSes, MarFS will entrust the responsibility to manage access to files to the application so that parallel performance can be maintained.

In this design where MarFS uses object stores for storing data, the metadata and object store systems require some reconciliation. All metadata, except for object-specific metadata, is stored in the MarFS POSIX file systems that store the metadata. As mentioned, periodic garbage collection of freed space will be done. MarFS attempts to minimize areas where truly transactional semantics are needed, but does not make any transactional guarantees. MarFS errs on the side of making it easy to run batch inode scans or tree walks due to the parallelism in the batch utilities and ability to use many POSIX file systems and metadata servers to make for easy management of the metadata/space/etc.

Variable File Collections

The nature of the large data stores is that different users have file collections with different attributes. Some users may have many small files while others have moderate collections of moderately sized files, and still others have a handful of very large files. MarFS shall allow for all these cases and yield high performance.

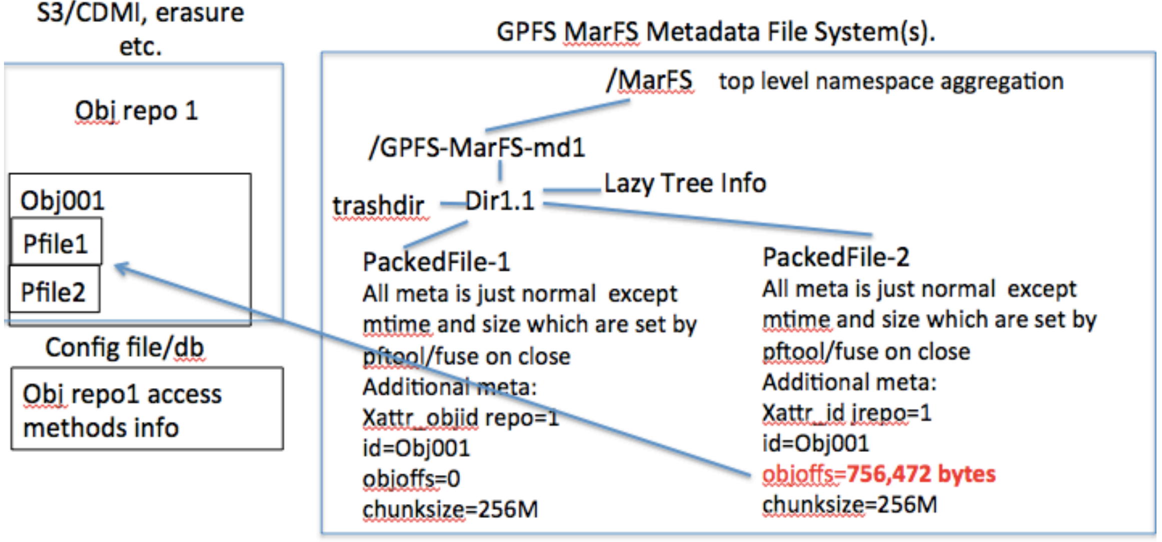
Design to Provide Variable File Collections

MarFS supports the concepts of packed files, uni-files, and multi-object files. Furthermore, it allows for multiple data storage solutions to be used as the MarFS data storage component. These different data storage solutions can be designed to efficiently handle and be high performance for files of different quantity and size attributes.

Packed files are targeted at the case where the user has many small files that are not efficiently handled by the data storage component. The metadata component allows the user to see these as the multiple files they logically are to the user, while the underlying data storage component collects many of them to be stored in a single object so that the data storage component exhibits high performance.

Here is a depiction of a Packed file in MarFS.

Packed File (same as previous but packed object)

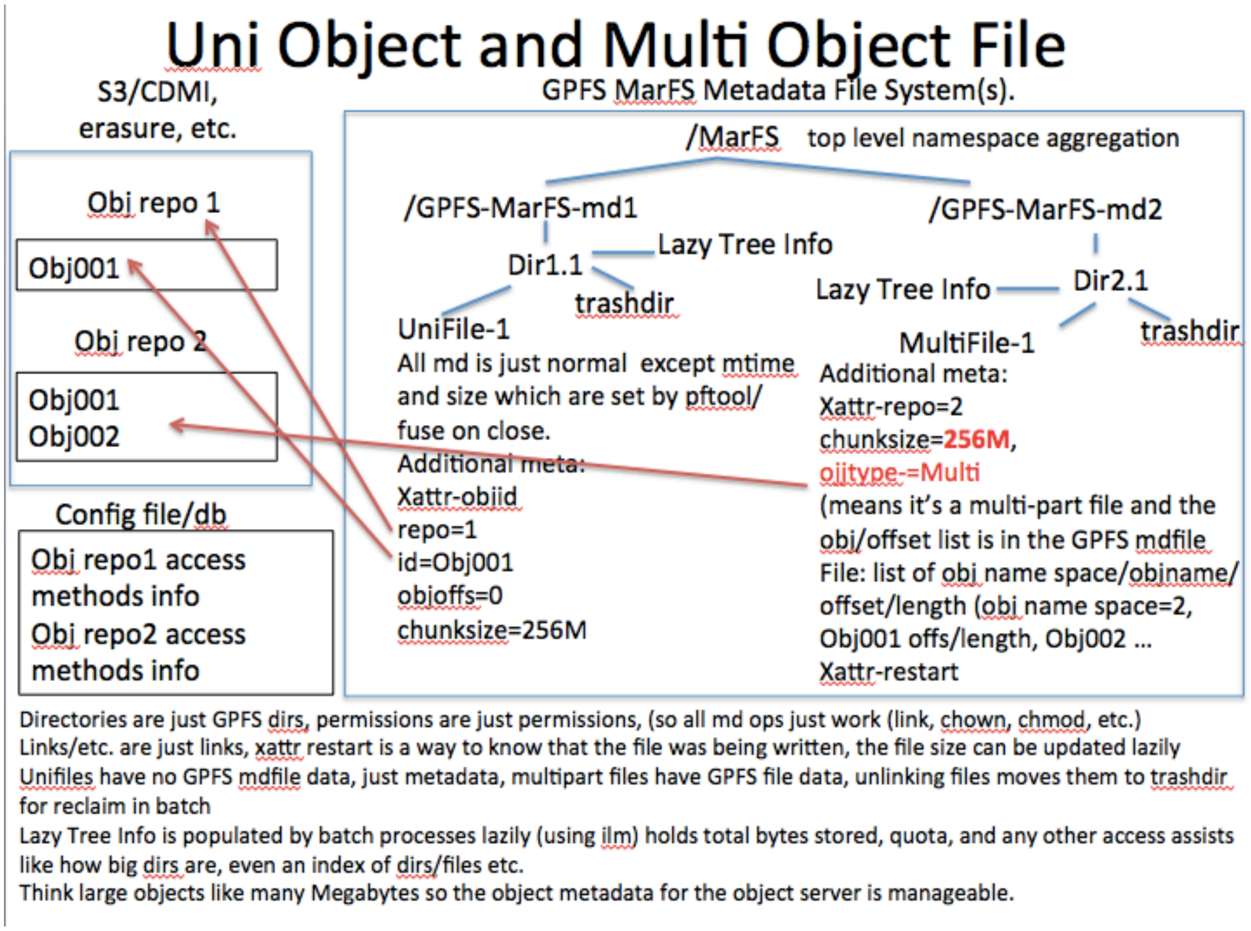


When files are overwritten (they have to be completely over written, no update in place, enforced by pftool and fuse), trunc'd/unlink'd files are moved to trash and clean up can repack and get space back in batch, not done interactively. Packed objects can get trash in then as well.

Uni-files are targeted at the case where users have moderate collections of moderately sized files. These files are large enough to be efficiently handled by the data storage component as individual files. Consequently, there is a 1:1 mapping of the logical file exposed to the user in the metadata component and the physical file stored in the data storage component.

Multi-object files are targeted at the case where users have a few very large files. The files are so large that they cannot be efficiently handled by the data storage component unless they are physically stored as multiple objects. Thus, the metadata component exposes the file to the user as a single file, but internally tracks all the objects that comprise it so that the data storage component can store it as multiple objects that are sized such that they can be efficiently managed.

Here we see MarFS diagrammed to show a GPFS POSIX metadata tree that has xattrs pointing at external object repos for a Uni file (one object per file) and a large Multi file (multiple objects per file). In the case of the Multi file, the list of objects that make up the file are stored in the metadata file, which implies that the chunksize of a large file has to be larger than an objectid, and the amount of real object id data in the metadata file is stored in an xattr so you know how much data is in the metadata file and the rest is unallocated space (a capability of sparse file representation in the metadata store) so that the total size of the metadata file is equal the logical file size (amount of data stored in the data store).


Uni Object and Multi Object File
S3/CDMI, erasure, etc.
GPFS MarFS Metadata File System(s).
Obj repo 1
Obj001
Obj repo 2
Obj001
Obj002
Config file/db
Obj repo1 access methods info
Obj repo2 access methods info
/MarFS top level namespace aggregation
/GPFS-MarFS-md1
/GPFS-MarFS-md2
Dir1.1
Lazy Tree Info
trashdir
Lazy Tree Info
Dir2.1
trashdir
UniFile-1
All md is just normal except mtime and size which are set by pftool/ fuse on close.
Additional meta:
Xattr-objid
repo=1
id=Obj001
objoffs=0
chunksize=256M
MultiFile-1
Additional meta:
Xattr-repo=2
chunksize=256M,
ojitype-=Multi
(means it's a multi-part file and the obj/offset list is in the GPFS mdfile
File: list of obj name space/objname/ offset/length (obj name space=2, Obj001 offs/length, Obj002 ...
Xattr-restart

Directories are just GPFS dirs, permissions are just permissions, (so all md ops just work (link, chown, chmod, etc.)
Links/etc. are just links, xattr restart is a way to know that the file was being written, the file size can be updated lazily
Unifiles have no GPFS mdfile data, just metadata, multipart files have GPFS file data, unlinking files moves them to trashdir for reclaim in batch
Lazy Tree Info is populated by batch processes lazily (using ilm) holds total bytes stored, quota, and any other access assists like how big dirs are, even an index of dirs/files etc.
Think large objects like many Megabytes so the object metadata for the object server is manageable.

Object naming

The name of the object for a file is named URLprefix/bucketname/MAR_objid). bucketname is /namespace.repo.suffix (from the configuration file). Thus, the fully qualified object name is URLprefix://namespace.repo.suffix/MAR_objid.

System administrators need to set up buckets on the object server(s) for every namespace.repo combination.

The URLprefix field for accessing the repository is in the configuration file in the repository record. repomethodinfo (cdmi, s3, etc.) tells how to access the repository.

Recall that repository name should point at some name of a portion of an object repository. For example with the Scality sproxyd access method, this repository name would match to a stanza in the sproxyd file that tells the object system the storage format for this repository, like 30+6 or 40+8 along with other attributes.

A suffix can be added to the bucket name or prefix to the object name in systems that require periodic bucket/prefix change.

The MAR_objid xattr portion of the object name contains:

- Reverse order time stamp of some kind to make sorting of object lists easier.
- Version number.
- File Create time. That is the create time for the metadata file, which for a Packed file is that of the first file in the packed object.
- Object creation time. This is needed because on operations like truncate or potentially on a restart, we may have objects that we need to "overwrite". This means we could have many objects with the same name (same chunk of same file after a truncate or something), and so this makes "versions" of that object in that file.
- Packed or not.
- Comptype.
- Sectype (encryption).
- Correcttype (crc, checksum).
- Chunksize.
- Chunknumber (for a Uni file, there will be only one chunk).
- Mdfile inode.
- namespaceshardpnum (future, zero for now, used for hashing files across namespace shard directories).

For a Multi file the second and beyond objects that comprise it will only change in the chunknumber attribute, which will be incremented.

Security

MarFS shall obey all POSIX security. The POSIX permissions are:

- r – read
- w – write
- x – execute

for:

- u – user
- g – group
- o – other

MarFS shall support additional permissions that may be supported by other data stores. These are in addition to the POSIX permissions .The values are rmwmrdwd:

- rm – read metadata
- wm – write metadata
- rd – read data
- wd – write data
- ud – unlink data
- td – truncate data

MarFS shall protect files that are deleted by the user, but not actually removed from the system until garbage space is collected.

Design to Provide Security

MarFS obeys all POSIX security. Additionally, special security may be added by configuration to manage which parts of the name space allow metadata and data update/read, and you can control these special permissions for interactive and batch separately.

By adding the additional data store permissions it is possible, for example, to allow read and write in POSIX permissions, allow metadata changes but not allow writing of data. This value is not stored with the file, it is interpreted real time, so this is a fast way to shut off write of data or metadata etc. This item can change based on allowed activity against this namespace and the data/space it represents.

Object Security could be provided by the following methods:

- Vault: Where a password or a key to open a password in a file for a given user is stashed.

  FUSE runs as root or similar so it can become stgadmin, read in the secret, then it is in the fuse daemons memory. Since it is in a different process space run by root users can't see that info.

  Pftool and other batch utilities would either run as stgadmin/root or use setuid sticky bits to gain access to the secret to open up the object repo. Since this is somewhat dangerous pftool will run batch processes on other nodes or containers. It will run remote of the user process where the user types a command and some number of machines go off and get the answer or do an operation on behalf of the user while the user does not have access to the batch machines except through a controlled interface.

  This method is easy to implement and uses a strong security method. It allows the the password to change on the object server and in the secret files owned by the user whenever one feels it is necessary to change the password. It's similar in nature to a group of people who have the combination to a shared vault.

  There may be more than one password, one per repo, one per namespace, one per whatever, but not too many, it needs to be manageable.
- Per Object: Where a data store vendor, such as Scality or EMC, adds the ability to require a secret on each object request. The secret would be stored into the POSIX metadata for each file. This would work as well and is pretty elegant too, but it doesn't allow one to change something simply to re-secure compromised information.
- Combination of Vault and Per Object: It is unclear that this is any better than Vault by itself. It may slow down the smart hacker, but not much.

- Encryption: One could sniff the wire and get the data. If the data is encrypted one could not sniff the wire and get to the data as it goes across the network. This would be more than we are doing now with mass data movement. Encryption at rest isn't an advantage because the data is erasure-coded on the server and that is a per object erasure-code, so one would have to physically steal a lot of disks to make any thing out of the data. Encryption would need to be by name space or by repo or by object. In all cases we have to stash the encryption keys somewhere and it essentially becomes Vault or Per Object or their combination.

Vault appears pretty elegant as you can change the secret often etc. It requires basically no coding, just some administration for the most part. Our design will use Vault, but leave room in our design for a per file password just in case we decide to go down that path. If at some point we find a better way that doesn't require setuid/sticky then we could remove the restriction of running pftool on batch nodes only. One can still run pftool's pcp or pls interactively, it runs a batch program on the batch servers and connects the console to one's interactive session.

Unlink and truncate operations leaves pointers to files and data in the trash directory for that namespace. It is important to protect the trash directory because it will contain trash names and space from various users/groups. It does not have a directory structure so maintaining control over access has to be managed since being able to find files requires the execute permission on the directory structure above a file. The FUSE daemon and pftool will deny all access to the trash directory. A trash utility will be created that allows users to interrogate the trash based on file ownership, groups, and/or POSIX permissions, all of which are preserved when files are moved to the trash directory and when files are truncated causing space to be moved to the trash directory.

Resilience

MarFS is intended for high performance access to information that must be available for years, possibly decades or longer. RAID6 is insufficient. Replication is too costly.

The MTTDL (mean time to data loss) shall be 100 years.

Design to Provide Resilience

The ability to recover data and be resilient to loss will be accomplished by three mechanisms: backup, erasure coding, and encoding recovery information.

Backup

The metadata service is broken up into multiple namespaces, each of which is a POSIX file system. Backing up the metadata will add to the system resilience. Backing up these file systems assumes there is not a lot of real data stored in them, which will be typical for our design.

IMPORTANT NOTE: The backup mechanism must honor space holes, so backups are not huge.

Metadata attrs and xattrs, and file data, which might contain object ids, must be preserved.

The MAR_post xattr numobjects field indicates if there is chunk information in the metadata file, and chunkinfobytes indicates how much real data is in the metadata file. Even if the contents of the metadata file that contains chunk information is lost, it can be recovered because the xattr MARobjid contains the name of the first object and the chunk size. The file size attr has the total file size. Thus you can know where all the objects are because the only thing that changes in the object names that are a Multi file is the chunk number, which can be calculated. If the metadata file content is lost, the only things really lost are the actual space used by the chunk and attributes that are not absolutely necessary to read the data from the data store. The data store will contain the information from which you can reconstruct the metadata file content.

Frequent backup of the metadata is wise and so keeping all user data out of the metadata file systems helps keep the cost of backing up the metadata low.

Erasure Coding

The data store systems are erasure coding based. Erasure coding can be setup to easily provide a MTTDL of 100 years. That said, it is important to configure these systems to provide adequate MTTDL while minimizing the overhead of storage devices that are used to provide that MTTDL and not provide actual data storage capacity. The data stores cannot be backed-up, as they will be very large in capacity.

Encoding Recovery Information

There has been an attempt in MarFS to encode as much of the metadata into the data objects as possible while not costing too much performance. This really means that because of the way objects work, at object create time, objects have fully re-creatable file information embedded in the object or its name.

Encoded in the object name is the xattr MARobjid, the contents of which are described in the Object Naming section. It is possible to use this information to help you figure out how to piece together metadata information on the metadata file system. It is much harder for a Packed file, as this object name just represents the first file in its object. Additionally there is even more recovery information embedded in the object data itself. Fileinfo is a record that has information mostly from stat() of the metadata file in human readable form. Remember this is recovery information captured at create time only, it is not updated upon metadata changes like chmod or chown or rename etc. It includes:

- Size of record.
- Version of record.
- inode.
- mode.
- uid.
- gid.
- mtime.
- ctimeand.

- full path of the file.

For objects used in Uni and Multi files the layout of the object is:
Data.

- Fileinfo record.
- The contents of Mar_post.
- Number of files (used in packed).
- Offset in this object where recovery info begins.

For packed files, the recovery is similar. pftool (or in the future a post-processing utility program) is the only way to create a Packed file, so it is possible to batch the recovery information for efficient writing at the end of the object. In this case, the packed object looks like this:

- DataFile1
- ...
- DataFileN.
- Length of Fileinfo1+Mar_post1, Fileinfo1 record, Mar_post1.
- …
- Length of FileinfoN+Mar_postN, FileinfoN record, Mar_postN.
- Number of files.
- Offset in this object where recovery info begins.

So normal reading of the object won't ever get to the recovery information held in the object data. The reason to put this in the data itself is the size might be large given full path is included. To recover (remember this is a create time only recover), you list objects and find the ones you are interested in (via bucket name, namespace.repo.suffix, and time stamp/etc.), get the header of the object which tells you the length of the object, read the last 2 words of the object (number of files and location of recovery info), use the location of the recovery information to read the recovery information and then recover the metadata (create time metadata) for the file.

This does not help you with metadata only changes like chmod, rename, chown, etc. If you want to protect yourself from loss of this info, frequent backup of the metadata is the answer. Potentially one might log metadata only updates at some point but there is no plan to do that now.

Lastly, if there is an easy way to dump a list of the objects in the buckets, the object names themselves occasionally, this would be both a good way to do reconciliation between MarFS metadata systems and object storage systems, but it also might be useful in an emergency recovery scenario of some kind.

Future Features

Of course, MarFS can be much more than its current design. Some envisioned future features are:

- File versioning. We have the ability to version the data behind files since unlink and truncate put the old file "space" in the trash.
- Telescoping/indexing/namespace with directories marked by directory xattr using indexfs or other directory pickling, both at single level directories and eventually multi level directories (telescoping index/namespace).
- Dual copy, probably implemented by a repository that does dual copy.
- Metadata update logging.
- Compression. The hard part is how to read compressed files/chunks in fuse.
- Encryption. The hard part is how to read encrypted chunks in fuse.
- Offline optimizations/sorting/indexing of attrs and user xattrs etc.
- Maybe append or sparse support, need to consider carefully, hard to do because of book keeping and because we chose to use formulaic striping instead of extent lists with sizes for multi files, we could use the multi mechanism that holds things like actual size of the chunk perhaps.
- Other access methods than CDMI/object, HPSS, remote, etc.
- HDFS alternate access of same data, via java HDFS lib, provide chunk info and query object store if it has info that matters (if its erasure allows locality).
- Would be nice to have restart for big files but that could be deferred.
- Packed file support could be later but would be nice to have sooner to make it easy on object system.
- Backup of object level metadata. List all objects in a bucket because all the metadata is in the object names.
- Would like to have V2 of erasure library sooner than later for improved bandwidth.
- Offline deep reconcile/repack, if trash is lost.
- Investigate GPFS keeping track of changes for further optimizations of utility programs. For example, only process changed parts of the tree.
- FUSE packing on write.
- FUSE multipart write.
- May need a special way to load data from HPSS to campaign that is very large, as we will need multi-part through FUSE or some other special way. In short term we could force this to go to scratch first for a while. Moving it from HPSS is the hard part. Once on scratch it would go fast to campaign, but need to figure a good way to do this at some point.
- May want to do other optimizations to HPSS.