Suggested updates to MarFS 09/2015

In order of importance:

Multiple object servers support (no update in place): (we may need this for initial deployment if we cant get an object server to scale well enough) (pretty simple change I think)

It is possible that we will encounter a situation where one object server will not scale well enough for a repo. It seems wise, if not very wise) to support multiple object servers. Of course object servers do not support update in place, so this turns out to be pretty easy as we wont be writing to such a repo in parallel with fuse, we will only parallel write to this type of repo with pftool. We can only write to a repo in parallel with fuse if the repo supports update in place (which today is only Direct and the only other repo type that supports update in place that is in plan is semi-direct).

To add this striping across multiple object servers the following changes are required:

1. in the configuration file, the repo record would have to have an added field, numcomponents. This describes the number of object servers we will stripe across.
   1. This is a value that cant easily be changed for a repo, so once you choose this for a repo you cant really change it, or its hard to do and keep things balanced. This is because you would have to “restripe” data if you ever added or took away components. You can over specify the numcomponents however. For example you could specify 8 object servers when you actually only have 4. The configuration of the paths to the object servers underneath MarFS could be made in such a way that 2 stripes would be written to each object server. This numcomponents is just a logical entity to stripe over and only creates different paths to get/put objects from/to. If you over specify, then restriping could be made much easier.
2. The xattr name of the object would change to have component number near the beginning of that string. (this value would be used to target different object servers for different objects)
3. The post xattr should probably contain number of components the file was created with (this is just the numcomponents from the repo from the config file when the file was created).
4. When putting, to determine what object server (component) to target, for uni and multi you run some hash on the combo of namespace.mdshard.inode of the mdfile which is unique in marfs. In the case of a uni file you would write the file to the component that hashes to. For a multi you would write the first chunk of the file to the component that hashes to and then you would round robin through the remainder of the components, one chunk per component for subsequent chunks. For Packed, you would hash for the first file in the packed object. Packed files do not span objects, so each file’s mdfile will have the xattr that points at this object.
5. For getting, you just get the xattr of where the object is and do the normal thing except for multi – where as the chunk number goes up so does the component in the round robin.

I believe this is basically all there is to striping across multiple object servers.

It seems like a nearly trivial change and allows us to easily stripe data (without update in place) across arbitrary numbers of object servers.

Of course if you just have one component in the configuration all the objects are written to that component/object system which is just a hash that only can produce one result, which handles the 1 component object repo just fine, which is the same as what we do now.

One additional consideration, it would be possible to support remote file systems used as object servers through the use of something like httpfs which puts object get/put on top of an hdfs file system. So we could use one or multiple remote file systems as object servers using this httpfs concept.

One or multiple file servers support (no update in place) (we may need this upon initial deployment if we cant get an object server to scale well enough) (pretty simple change assuming someone has worked out how to use IOFSL and put the security comm method in):

This section is to allow us to use one or multiple file systems as a non update in place storage repo.

This functionality does not replace Semi-direct. Semi-direct is a staging area that allows update in place in parallel, where this file type repo is meant to be used like one or multiple object stores but using one or multiple file systems in parallel for repo space.

This particular section is about how this would be done but without update in place.

Another section will discuss update in place, which can only be done with repo type Direct and Semi-direct

Issues with the file type repo:

The primary issue with using posix to talk to a repo is the fact that you do not want to replicate the user metadata directory tree in the repo, as that creates huge headaches to keep in sync, especially when you generalize to striping across N file systems as this would be keeping N file system trees in sync (a bad idea). Instead you want to just use a flat or pseudo flat name space to hold the objects. You want to name objects/paths to objects in the file system that makes up the file type repo (or component of that repo) in such a way that renames of trees/files do not effect the naming of the objects/paths to objects. Since you are using a file system to hold objects, you don’t want to have all the objects (which are files in the file type repo) to be in the same directory as you would have millions of files in the same directory. So you want to scatter the objects (which are files in the file type repo) across some directory structure which is agnostic to the real user file name space directory structure. This has the ill effect that protections provided by the user directory structure are no longer there and it would violate posix to try to break this paradigm. You really want the file type repo to act much like an object server is used in MarFS where the objects (which are files) to be owned by some third party user and you want some other protection mechanism than posix to be used to control access to these objects (files). The best option for this appears to be something like IOFSL which is a posix forwarding layer. It has a library interface which fuse and pftool could link with to get access to remote file systems. It has protocol layers for IB verbs, tcpip, etc. Additionally IOFSL is stateless which makes this a very clean protocol. We could add a security mechanism like object servers have which would give us the ability to send crypto headers on every remote forwarded file system command just like using object server protocols like S3/sproxyd.

The implementation of this would be to just provide an alternate put and get subroutine/function that uses iofsl with our secured comm method which would then treat the remote file server/servers as object servers except the get/put functions would flow via iofsl instead of http.

With the above “Multiple object servers support” added to Marfs to allow use of multiple object servers, this use of one or multiple file servers becomes just a conguration of repo type is file and update in place turned off.

If the repo type is file and update in place is off, then you just call the alternate remote secure file put and remote secure file get functions instead of the aws / sproxyd/s3 get/put functions. There would need to be a scatter tree in place just like in the trash mechanism but we would assume the system admins would produce this. We would need to agree on what this scatter tree looks like.

The striping mechanism would work exactly like described above in the “multiple object servers support” mods.

This method would allow for one to use multiple ZFS or ZFS on ZFS or GPFS on ZFS or … as repos (not update in place).

Preserve MarFS use of Setuid bits for directories and files (discussion) (this isn’t strictly needed for initial deployment but we would want to not have to go back and change any bits users have set, so disallowing this now makes sense) (pretty simple checks on chmod and on stat to not allow setuid and mask out on stat):

We may need to use some bits in the stat structure for our own use. I propose that all set and get mode bits don’t allow setting of setuid for directories or files. This will preserve the bits for our private use. These are about the only things in the stat structure that we can “reuse” that users can set. So we would hide these bits and use them for our own purposes.

LATER

Update in place (discussion) ( this would be very nice to have pretty soon after initial deployment because it solves the (how do we pull from hpss and write to marfs in a scalable way without having to go through a scratch or something):

The direct and semi-direct repo types provide update in place capability.

The direct repo type is just putting the file data into the mdfile – in parallel if the mdfs is a parallel file system.

The update in place function requires director or semi-direct type repo (something where updating parts of files will work).

Direct is simple passthru of ops to the mdfile.

Semi-direct is a way to remap where the file data goes to an alternate file system, not the mdfs.

Issues to consider for Semi-direct:

Since you are not storing data in the mdfile, you want the file in the semi-direct repo to have a calculatable name/path – just like in trash or a file type repo, one that is not effected by renames etc. as you don’t want to have to keep two name spaces synced. You have the same problems with a semi-direct repo as you have with file type repo security/rename/flattening/scatter-tree issues so all access to these semi-direct repos must be done via the secure remote file protocol (IOFSL + security).

The other major issue with update in place is how to deal with file size and mtime. Update in place implies that apps can seek around and write where they want, overwriting parts of files or even extending the file length. Further it implies that more than one fuse daemon (and they don’t know about one another) could be updating different parts of the file and more than one could be extending the file concurrently. Some of this we can assume away – like overlapped writes – we can call that user error for MarFS. What is difficult to deal with is how to extend the file size and mtime in a scalable way both for writing and for reading/stating. This is a classical problem.

Recall, the primary reason for allowing update in place via fuse is to allow things like globus and psi/hsi/pftp/hpss to be able to write to MarFS in parallel (a single file striped). Reading a file from Marfs striped should work but writing one wont because that is really update in place. We were thinking of this direct/semi-direct repo area as a staging area for MarFS for fuse access methods so we allow for parallel writing via fuse and we write all small sized little files to this area s well and we would later pack/reshape files into on object or file (non update in place) repo type later to clear this staging area.

It is easy to see why you cant use an object repo type for this update in place activity because there is no updating in place of objects, so update in place would be a horrible extent based garbage collection nightmare.

It is less obvious why you cant use a file type repo for this update in place repo.

First, the file type repo mimic’s the object type repo in that it breaks big files up in to parts, could be thousands or millions of parts. It also allows packing, which we could turn off for update in place. The huge problem is how do you know the file size and mtime for a file stored in a file repo. If you don’t allow update in place, then its easy, the close updates that info and there is only one close because you don’t allow update in place so multiple processes cant be extending the file in any meaningful way. In the case of an update in place repo, multiple non related (from a fuse point of view) distributed writers can be extending the file. There are really two ways to deal with that:

1. locking on the mdfile and updating the file size and mtime and xattr and each process extends the file if it needs to and updates the mtime if it needs to. This of course doesn’t scale well because first of all these file systems are remote and second of all you may have hundreds or even thousands of processes writing. Further you don’t have much control over the applications that are writing in parallel. They might play nice and they might not, (big ops with no overlaps etc.).
2. The other way to deal with mtime and file size is to defer it. This means that you don’t keep track of mtime and file size during write, you calculate it at stat/read time. The way you do that is you go out to all the parts and stat each one and figure out the file size and latest mtime. This is the way the Q file system worked. It is pretty awful for big/wide systems. Of course we could write a batch utility to update the size and mtime in the mdfile later.

One of the big issues is all the edge cases. What happens if apps are extending and contracting the end of the file without writing, and other totally crappy ways to treat a file. There are lots and lots of edge cases, some we care about and others we don’t. This ability to deal with mtime and file size on N to 1 writing is why we have parallel file systems. They exist for these very reasons. If we are to stick to the principle of Marfs, don’t recreate wheels, like use a file system for posix metadata, use an erasure enabled object or file store for data repo, etc., then you would come to the conclusion that semi-direct needs to be a parallel file system (unless you never do N to 1 write, then it could be just a non parallel file system or a set of non parallel file systems. If you used N non parallel file systems, even without N to 1 write, you still have big balance problems. Again parallel file systems exist for these very reasons.

So, I propose that semi-direct be almost identical to direct. Just like the file no update in place repo is almost exactly like the object no update in place repo.

1. I propose we use a parallel file system for semi-direct repo because we have N to 1 and because we can then do anything a parallel file system can do in fuse/multiple fuses and it will all just work under the marfs namespace and then we can easily migrate under the covers. We cover all edge cases with ease.
2. I propose we don’t put recovery info in the file as it is just like direct just in a different file system
3. I propose we don’t use trash for these files (although we could), we just delete them
4. I propose we use the iofsl secure remote file access as a way to talk to these files because we don’t want to have to replicate the tree/etc.
5. I propose we don’t pack on this semi-direct repo
6. I propose we use the file setuid bit to signify that this is a semi-direct files and we use that info to know that we need to go to the semi-direct repo to fine out the file size and mtime when stating . If we don’t use something like this, then we have to do an Xattr look up on every file in marfs to see if the size and mtime are correct or not for all stats. Since we will stat the mdfile we will then know if we need to stat the semi-direct file or not. The assumption is that most files wont be semi-direct so you would save doing a get-xattr on all but the semi-direct files where the mtime and size is wrong in the mdfile.

So the semi-direct spec needs to be rewritten given this update in place info:

secure remote file protocol, mtime/file size, setuid bit, etc. and just like the file type repo (no update in place) is the same as the object (no update in place) repo, the semi-direct will be just like the update in place, for the most part with just the path you are operating on being different, using the iofsl/remote secure access, setuid bits to indicate file size and mtime are remote.

I will update the semi-direct spec accordingly. This turned out to simplify greatly semi-direct, as it follows direct almost exactly.

Much Later:

Sharded metadata (discussion) ( this is really not needed for some time I suspect, it is a fundamental way to solve metadata scaling, but since we will be able to use N mdfs’s (one per user/project) we will be pretty scalable already – more so than any archive we have ever deployed – without this feature. There is quite a bit of unknown in this yet, like links and things. This is for follow on md scaling.:

Just like with the file type repo, you want to just use a flat or pseudo flat name space to hold the mdfile entries because you don’t want to replicate the tree. You want to name mdfiles in a way that rename of file or directory don’t matter. You are forced to a flat or pseudo flat name space using the namespace.directory inode as the mechanism for scattering and naming of md directories. This leads you to the same security problem of trying to use posix to secure but you cant because you don’t have the entire tree replicated. The answer appears to be the same as for file type repos, using a lower level protocol to provide the security or IOFSL.

So there is an addition to the specs for sharded metadata, instead of using directory operations like mkdir etc. locally, the operations must flow through the remote secure file system access.

One other potential addition is the use of the suid bit on the directories in the master name space mdfs (where the directory structure is held). This could be used to know if the directory has extended attrs. We of course would hide the suid bits from the user on any user called operation like stat. If we are making a directory that is to be sharded, we set the uid bit on the directory and poke the xattr about directory shard width as discussed in the sharded metadata specs. This would mean that you would know whether to do a getxattr call or not for a directory just from stat info which you already have typically. This isn’t strictly needed but would be a nice addition. It might be likely that only a few directories are big enough to be sharded and this would streamline this entire set of operations when walking a tree where most directories are sharded abut some are.

I will update the mdsharding spec accordingly.