An Insight about GlusterFS and its Enforcement Techniques

Manikandan Selvaganesan and Mohamed Ashiq Liazudeen
Department of Products & Technologies, Red Hat
Bangalore, India
manikandancs333@gmail.com, ashiq333@gmail.com

***Abstract*— The world keeps contributing to the increase in data everyday drastically. Scientific applications, weather forecasting, researches, hospitals, military services are few such major contributors. As the amount of data increases, the need to provide efficient, easy to use solutions has become one of the main issues for these type of computations. The best solution to this issue is the use of Distributed File Systems(DFS). Some existing Distributed File Systems are too complex to deploy and maintain, although they are extremely scalable and cheap since they can be entirely built out of commodity Operating System(OS) and hardware. GlusterFS solves this problem. Gluster File System(GFS) is open source and is capable of scaling to several petabytes(upto 72 brontobytes) and handling thousands of clients. It is based on a stackable user space design and can deliver exceptional performance for diverse workloads. GlusterFS is an easy to use clustered file system that meets enterprise-level requirements. It is written in user space which uses FUSE(Filesystem in user space) to hook itself with the VFS layer. It takes a layered approach to the file system, where features are added/removed as per the requirement. Enforcer is a major component in Gluster which finds its extensive use in production. Enforcer is concerned more about limits. It helps to restrict the usage(both in terms of size and count) at directory or volume level in the file system. In this paper, we discuss the internal working of the translators and key architectural components of GlusterFS. Along with that, one can gain a deep insight about the enforcement techniques, few challenges, undergoing researches and projects of GlusterFS.**

Keywords: Distributed File System(DFS), Gluster File System(GFS), Translators, Enforcer, Marker

I. INTRODUCTION

GlusterFS is a scalable open source clustered file system that offers a global namespace, distributed front end, and scales to hundreds of petabytes without difficulty. It also offers extraordinary cost advantages benefits that are unmatched in the industry. No longer are users locked into costly, monolithic, legacy storage platforms. GlusterFS gives users the ability to deploy scale-out, virtualized storage scaling from terabytes to petabytes in a centrally managed and commoditized pool of storage,which is available to users in a single mount point, making it simple for the user.

At the heart of the design Gluster File System is a completely new view of how storage architecture should be done. The result is a system that scales linearly, is highly resilient, and offers extraordinary performance. Additionally, Gluster brings compelling economics by deploying on low cost commodity hardware and scaling horizontally as performance and capacity requirements grow.

A. Not only Storage but the Storage System

Storage does not scale linearly. One can think this is counter-intuitive on the surface since it is easy for someone to purchase another set of disks to double the size of available storage. An important limitation in doing so is that the scalability of storage has multiple dimensions, capacity being one of them. Adding capacity is only one dimension, there are few other factors which contribute as well such as the CPU capacity, the scalability of the file system to support the total size. Another major thing is the metadata telling the system where all the files are located must scale at the same rate disks are added and the network capacity available must scale to meet the increased number of clients accessing those disks. To be precise, as the title says, it is not the storage that needs to scale as much as it is the complete storage system that needs to scale.

B. Traditional Approach

With current Distributed File Systems(DFS) the problem is that systems scale logarithmically as discussed here[1]. This is re-factored in Gluster file system. With the former approach, storage's useful capacity grows more slowly as it gets larger. This is due to the increased overhead necessary to maintain data strength. This limitation is examined by testing the performance of some storage networks which clearly reflects that larger units offer slower aggregate performance than their smaller counterparts. It is necessary to completely revisit the underlying architecture to overcome this limitation. Any system that requires end-to-end synchronisation of metadata or offers a limited number of networking ports must be implemented efficiently from its base architecture. Those solutions that cannot act as a cluster of independent storage units are bound to find a scalability limitation sooner rather than later.

C. True Linear Scalability in GlusterFS

To achieve true linear scalability, there are some fundamental changes to how storage must be done

- Metadata synchronisation and updates could be eliminated.
- The way data is distributed to achieve scalability and reliability.
- Make use of parallelism to improve performance.

The impact of these change can significantly result in improved performance. GlusterFS is one such proven example.

To understand how these are acheived with GlusterFS, let's dig into how these changes work.

II. AN OVERVIEW OF GLUSTERFS

As mentioned above, GlusterFS has no metadata server and is capable of linear scaling and can handle upto thousand clients. Let's look a bit more deeper on how are these features affect performance of any distributed file system and how are they efficiently achieved in GlusterFS.

A. Meta Data's Impact on Performance

For distributed file systems, metadata is the heart and soul of how data is organized. Scaling and metadata are way dependent. A simple measure of performance involves simply timing of how long it takes to read or write a single large file. This brute force technique is called "sustained sequential access" and which is the basic thing any file system would be expected to do. Anyways that does not tell the customer about how the system would perform in a real world environment. The more complicated the workload, the more you would observe metadata also being exercised equally or greater in proportion to the number of I/O events directed towards the contents of each targeted file. The issue is even more if the files are been replicated or otherwise distributed so that users can simultaneously access the same data sets with complicated random access patterns, all being tested in the contest of permissions and other attributes contained in the metadata associated with each and every file and directory(or subdirectories).

An additional layer of complexity comes with a variety of settings distributed geographically, creating inherent complexity for the distributed metadata updates as well as the process by which any updates to the data itself must be carefully applied so as to be inherently correct(with chronological sequence preserved).

It is the fundamental nature of metadata that it must be synchronously maintained in lockstep with the data. Any time the data is touched in any way, the metadata must be updated to reflect this. Many people are surprised to learn that for every read operation touching a file, this requirement to maintain a consistent and correct metadata representation of "access time" means that the timestamp for the file must be updated, incurring a write operation. There are optimization techniques involving ways to collect these together and apply them in sets, but nevertheless, the more "expensive"(consumptive of CPU and other system resources) write operations associated with maintaining access time metadata are an overhead to read operations. To summarize the points, it becomes hectic in a distributed deployment, and the more they are geographically dispersed, the worse this overhead problem can affect the performance of the system.

Another important issue is that when metadata is stored in multiple locations, the requirement to maintain it synchronously also implies significant risk related to situations when the metadata is not properly kept in sync, or in the event if it is actually damaged. One big reason Gluster performs so extraordinarily well in actual customer deployments and across a wide variety of demanding benchmarks where real world workloads are more realistically simulated is simple: Gluster does not have a bottleneck regarding metadata. In fact, it does not need to scale its handling of metadata at all. Let's look on how the concept of "No Metadata" helps Gluster more scalable.

B. Gluster's Keys to Scalability

One of the most important advantages found in Gluster's architecture is its liberation from any dependency on metadata, unique among all commercial storage management systems. This fundamental shift in architecture addresses the core issues surrounding metadata in file systems. Gluster got rid of the complexities of metadata in a centralized or distributed environment which other file systems have. The unique 'no-metadata server' architecture of GlusterFS eliminates a bottleneck resulting and provides increased scalability, reliability and performance. The architecture within Gluster does not depend on metadata in anyway. Instead of applying complicated workarounds in the manner of traditional file system suppliers and vendors who seek to mitigate the problem with complex optimizations and shortcuts, Gluster instead eliminates the root cause entirely.

For all files and directories, instead of storing associated metadata in a set of static data structures(whether replicated and optionally distributed, or kept locally) also instead of applying the same inadequate band-aid to address the classic "metadata bottleneck" problem encountered by traditional distributed file system models, by moving metadata into a dedicated server with its own bottlenecks and issues - Gluster instead generates the equivalent information on-the-fly using algorithms. The results of those calculations are dynamic values acquired wherever needed in each of one or more nodes in a Gluster deployment. This eliminates the risk that metadata will never get out of synch because the algorithms are universal and omnipresent across the distributed architecture, and therefore for many fundamental reasons simply cannot ever be out of synch. And the implications for performance are staggering.

Gluster can process all data access operations independently at all locations throughout the distributed architecture because there is no requirement for the nodes to "stay in synch". That allows for linear scaling with no overhead. Gluster, however, quite simply scales in a truly linear manner because there is absolutely no reason for it not to do so. Also, by not depending on static data structures(whether single-instance or replicated throughout a distributed deployment).

C. Gluster is Faster, Achieves True Linear Scalability, and is Safer

Gluster is faster for every operation because it calculates metadata using algorithms unlike retrieving metadata from any storage media. It is faster and achieves true linear scaling for distributed deployments because each node is independent in its algorithmic handling of its own metadata, eliminating the need to synchronize metadata. It is safer in

distributed deployments because it eliminates all scenarios of risk which are derived from out-of-sync metadata.

D. Features and Advantages of GlusterFS

In precise, GlusterFS has the following merits or features which makes it better compared to existing cluster file systems available.

- GlusterFS can be deployed with the help of commodity hardware servers.
- No metadata server.
- N number of servers can access a storage that can be scaled up to several petabytes.
- Linear scaling and performance
- Aggregates on top of existing filesystems. User can recover the files and folders even without GlusterFS.
- GlusterFS has no single point of failure. Completely distributed. No centralized metadata servers.
- Extensible scheduling interface with modules loaded based on user's storage I/O access pattern.
- Modular and extensible through powerful translator mechanism.
- Supports Infiniband RDMA and TCP/IP.
- Entirely implemented in user-space. Easy to port, debug and maintain.

In the next section, let's look on how GlusterFS is implemented using various translators and types of volumes that GlusterFS has, followed by the overall working architecture.

III. ARCHITECTURE OF GLUSTERFS

Let's now explore how GlusterFS is implemented in the real world. Brick is the basic unit of storage in GlusterFS, represented by an export directory on a server in the Trusted Storage Pool(TSP). Volume is the collection of bricks and most of the gluster file system operations happen on the volume. Gluster file system supports different types of volumes based on the requirements. Some volumes are good for scaling storage size, some for improving performance and some for both. Depending on the need one can choose the type of volume. Following are the type of GlusterFS volumes:

A. Volume Types

- Distributed Glusterfs Volume - This is the default glusterfs volume i.e, while creating a volume if you do not specify the type of the volume the default option is to create a distributed type of volume. Here files are distributed across various bricks in the volume. Referring Fig. 1, file1 may be stored only in brick1 or brick2 but not on both. Hence there is no data redundancy. The purpose for such a storage volume is to easily scale the volume size. However this also means that a brick failure will lead to complete loss of data and one must rely on the underlying hardware for data loss protection.
- Replicated GlusterFS Volume - In this volume we overcome the data loss problem faced in the distributed volume. Here exact copy of the data is maintained on all bricks. The number of replicas in the volume can be decided by client while creating the volume. So we need to have at least two bricks to create a volume with 2 replicas or a minimum of three bricks to create a volume of 3 replicas. One major advantage of such a volume is that even if one brick fails the data can still be accessed from its replica brick. Such a volume is used for better reliability and data redundancy.

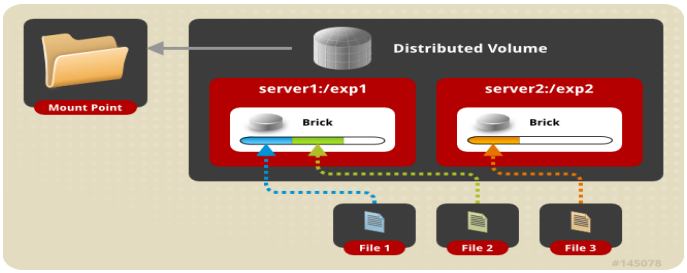

Fig. 1. Plain distribute volume

- Distributed Replicated GlusterFS Volume - In this volume files are distributed across replicated sets of bricks. The number of bricks must be a multiple of the replica count. Also the order in which we specify the bricks matters since adjacent bricks become replicas of each other. This type of volume is used when high availability of data due to redundancy and scaling storage is required. So if there were eight bricks and replica count 2 then the first two bricks become replicas of each other then the next two and so on. This volume is denoted as 4x2. Similarly if there were eight bricks and replica count 4 then four bricks become replica of each other and we denote this volume as 2x4 volume.
- Striped GlusterFS Volume - Consider a large file being stored in a brick which is frequently accessed by many clients at the same time. This will cause too much load on a single brick and would reduce the performance. In striped volume the data is stored in the bricks after dividing it into different stripes. So the large file will be divided into smaller chunks(equal to the number of bricks in the volume) and each chunk is stored in a brick. Now the load is distributed and the file can be fetched faster but no data redundancy provided.
- We have few other volumes like Distributed Striped GlusterFS Volumes which is pretty simple like the striped mechanism but additionally, the later is distributed. Also, the volume type can be EC(Erasure Coding) or a tiered volume or a disperse volume and so on. The disperse translator is a new type of volume for GlusterFS that can be used to offer a configurable level of fault tolerance while optimizing the disk space waste. It can be seen as a RAID5-like volume. To avoid triplication(three way replication) which is expensive and instead of wasting two redundant disks for every data, one can use erasure volume[2] which desires protection from double failure. To efficiently access(recently accessed) the files, one can use the

tiered volume which contains hot tier and cold tier where the recently accessed ones are stored in the former for a particular timeout(user formattable) and then is moved to the later. Choosing the volume type is based on the use case while deploying in the real world. Let us quickly move on to the next aspect(s).

So far it's been noted many times that GlusterFS is a User space file system. Let's dig more on how is this implemented in the next sub section.

B. GlusterFS - User Space File System

GlusterFS is a userspace filesystem. This was a decision made by the GlusterFS developers initially as getting the modules into linux kernel is a very long and difficult process. Being a userspace filesystem, to interact with kernel VFS, GlusterFS makes use of FUSE(Filesystem in Userspace)[3]. For a long time, implementation of a userspace filesystem was considered impossible. FUSE was developed as a solution for this. FUSE is a kernel module that supports interaction between kernel VFS and the non-privileged user applications and it has an API that can be accessed from user space. Using this API, any type of filesystem can be written using almost any language you prefer as there are many bindings between FUSE and other languages.

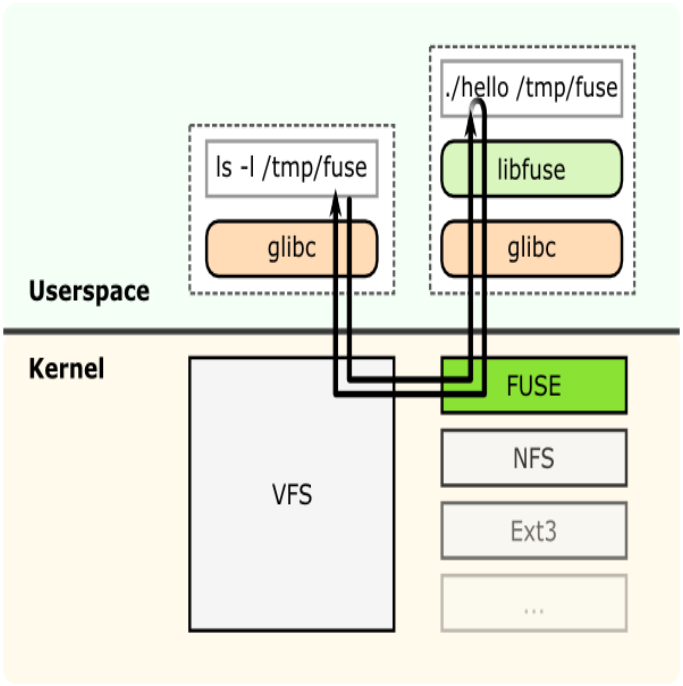

Fig. 2. Structural diagram of FUSE

Fig. 2 shows a filesystem "hello world" that is compiled to create a binary "hello". It is executed with a filesystem mount point /tmp/fuse. Then the user issues a command ls -l on the mount point /tmp/fuse. This command reaches VFS via glibc since the mount /tmp/fuse corresponds to a FUSE based filesystem, VFS passes it over to FUSE module. The FUSE kernel module contacts the actual filesystem binary "hello" after passing through glibc and FUSE library in userspace(libfuse). The result is returned by the "hello" through the same path and reaches the ls -l command. The communication between FUSE kernel module and the FUSE library(libfuse) is via a special file descriptor which is obtained by opening /dev/fuse. This file can be opened multiple times, and the obtained file descriptor is passed to the mount syscall, to match up the descriptor with the mounted filesystem. That's pretty much about FUSE and how GlusterFS makes use of it. Now, we clearly know that GlusterFS incorporates a lot of features. These features are implemented as translators. Though there are lot of translators, let's gain some insight on cluster and feature translators in the next section.

C. Translators in GlusterFS

A translator converts requests from users into requests for storage. A translator can modify requests on the way through : convert one request type to another(during the request transfer amongst the translators) and modify paths, flags or even data(e.g. encryption). Translators can intercept or block the requests(e.g. access control) or spawn new requests(e.g. pre-fetch). Translators make use of shared objects to communicate with one another. They are dynamically loaded according to volfile. Each translator sets up pointers to parent/child translators, call init(constructors) and call IO function through fops and has the provision for validating/passing options to one another. The configuration of translators(since GlusterFS 3.1) is managed through the gluster command line interface(cli), so one need not know in what order to graph the translators together. There are quite a few translators in GlusterFS.

D. Types of Translators

Gluster has different types of translators such as Storage, Debug, Cluster, Encryption, Protocol, Performance, Bindings, Schduler, features, etc. The default/general hierarchy of translators is shown in Fig. 3. All the translators hooked together to perform a function is called a graph. The left-set of translators comprises of Client-stack.The right-set of translators comprises of Server-stack. The GlusterFS translators can be sub-divided into many categories, but two important categories are - Cluster and Performance translators. Every translator has it's own functional purpose. Throughout the translators, it's been noted that "Extended Attributes" are important. Let's look more on this in a moment.

E. Extended Attributes

GlusterFS makes use of extended attributes in replication, distribution, striping etc. In DHT a directory must be present on all bricks and each directory copy will be assigned a hash range stored in its extended attribute -trusted.glusterfs.dht. A directory lookup will return the layout(hash ranges collected from the xattrs) which is stored in a table. This helps us to look for missing hash ranges(possible if the brick is down), overlaps, etc. In AFR the extended attribute - trusted.afr.* where * is the brick name, is used for recording operation failure. Consider two bricks brick0 and brick1 in a volume.

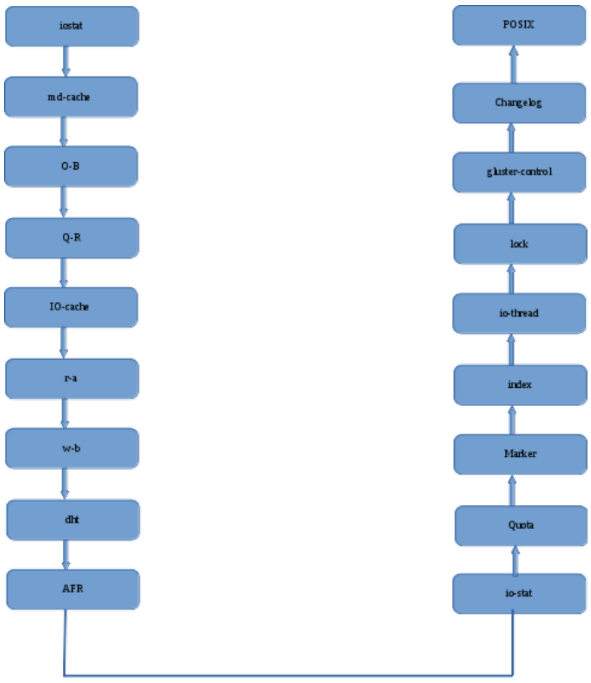

Fig. 3. Translators default hierarchy

A file on brick0 has the xattr trusted.afr.brick1 and a file on brick1 has the xattr trusted.afr.brick0. This is becasue if we store both the state of operation(success or failure) and the actual operation on the same brick and if that brick goes down, then there would be no way to recover from failure since we lose the state of the operations. Hence the operation and the state of operation are stored at two different places. The xattr works as a counter and records counts for three different kinds of operations data, metadata and entry. To perform an operation there are three stages: 1) Preop - whenever a modification is to be made all the counters will be incremented. 2) Op - here the operation is actually performed. 3) Postop - if the operation was successful then the counters are decremented. If the operation was successful across all the bricks then all counters would go back to zero. However in our example if the brick0 was down or had crashed before the operation was successfully completed then the counter for brick0 stored on brick1 will remain non-zero which implies that the operation on brick0 was unsuccessful. Now comes the feature translators. Let's look on how quota and marker handles the extended attributes. Quota and marker makes uses of xattr's such as : 1) size - to store the size of directory(or subdirectory). 2) contri - how much(size) of data is being contributed to the ancestor(s). 3) dirty - a flag to make sure the atomicity of operations.

Some of the other xattrs are trusted.gfid used to detect duplication in inode numbers. trusted.glusterfs.test stored in the root directory of every brick used for determining if xattrs are supported. Let's now look onto the different access mechanisms of GlusterFS.

F. Access Mechanisms

Native access mechanism being used in GlusterFS is Fuse. But FUSE however come with a small overhead due to context switches and s memory copies made during the data transfer operations. Hence in GlusterFS alternate access mechanisms[5] libgfapi, GlusterNFS, GlusterFS and NFS-Ganesha integration and using Samba(for windows environment) are implemented. Now let's collectively look on how things work altogether.

G. Overall Working of GlusterFS

Let's consider GlusterFS is installed on a server node. As soon as it is installed, a gluster management daemon(glusterd) binary will be created. This daemon should be running in all participating nodes in the cluster. After starting glusterd, a Trusted Server Pool can be created consisting of all storage server nodes(TSP can contain even a single node). Now bricks which are the basic units of storage can be created as export directories in these servers. Any number of bricks from this TSP can be clubbed together to form a volume. Once a volume is created, a glusterfsd process starts running in each of the participating brick. Along with this, configuration files known as vol files will be generated inside /var/lib/glusterd/vols/. There will be configuration files corresponding to each brick in the volume. This will contain all the details about that particular brick. Configuration file required by a client process will also be created. Now the file system is ready to use. We can mount this volume on a client machine very easily. When we mount the volume in the client, the client GlusterFS process communicates with the server's glusterd process. Server glusterd process sends a configuration file(vol file) containing the list of client translators and another containing the information of each brick in the volume with the help of which the client GlusterFS process can now directly communicate with each bricks glusterfsd process. The setup is now complete and the volume is now ready for client's service.

As shown in Fig. 4, when a system call(File operation or Fop) is issued by client(or application) in the mounted filesystem, the VFS(identifying the type of filesystem to be glustefs) will send the request to the FUSE kernel module. The FUSE kernel module will in turn send it to the GlusterFS in the userspace of the client node via /dev/fuse(this has been described in FUSE section). The GlusterFS process in client consists of a stack of translators called the client translators which are defined in the configuration file(vol file) sent by the storage server glusterd process. The first among these translators being the

- FUSE Translator which consists of the FUSE library(libfuse). Each translator has got functions corresponding to each file operation or fop supported by GlusterFS. The request will hit the corresponding function in each of the translator. Other main client translators include:
- DHT Translator- DHT translator maps the request to the correct brick that contains the file or directory required.

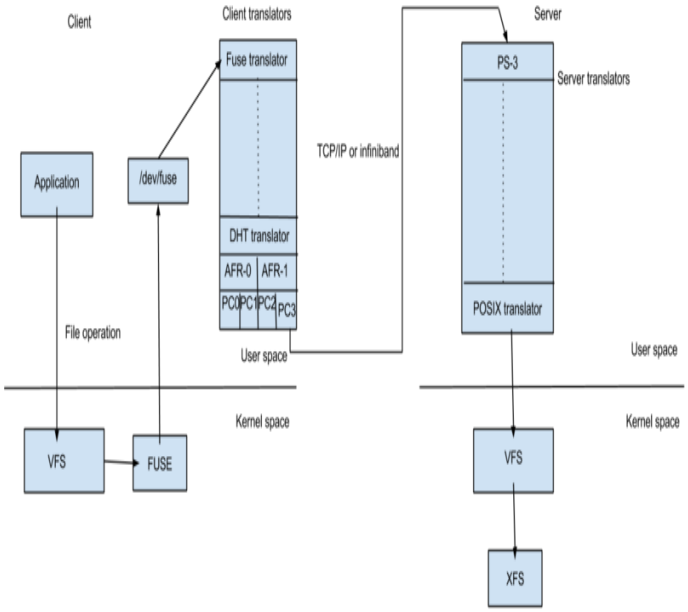

Fig. 4. Overall working of GlusterFS

- AFR translator- It receives the request from the previous translator and if the volume type is replicate, it duplicates the request and pass it on to the Protocol client translators of the replicas.
- Protocol Client Translator- Protocol Client translator is the last in the client translator stack. This translator is divided into multiple threads, one for each brick in the volume. This will directly communicate with the glusterfsd of each brick.
- In the storage server node that contains the brick in need, the request again goes through a series of translators known as server translators, main ones being: Protocol server translator and the final one being,
- POSIX Translator The request will finally reach VFS and then will communicate with the underlying native filesystem. The response will retrace the same path. If quota is enabled on the volume, quota and marker translators will be present in the server stack.

In the next section let us discuss more about how quota enforces and marker maintains accounting and communicate with other translators. Along with that, we will also explore on the techniques used by them.

IV. ENFORCER TECHNIQUES AND THE ACCOUNTING TRANSLATOR - MARKER

As mentioned in the previous section, quota does the the enforcing works and marker is the one which does the actual accounting. Quota enforcement is done on server side. It is not done in client side, because all the clients could not be trusted and to reduce the traffic on the network(if it is done on the client side, then we have to rely on lookup calls on file/directory(inode) to update the contribution). One can set the quota at the following levels :

- Directory level - limits the usage at the directory level.
- Volume level - limits the usage at the volume level.

User can set both hard-limit and soft-limits.

- Soft limit : Messages will be logged on reaching the soft limits.
- Hard limit : Writes will fail with EDQUOT (Disk quota exceeded error) once they reach hard limit.

Translators marker and enforcer sits on server side as shown in Fig. 5,

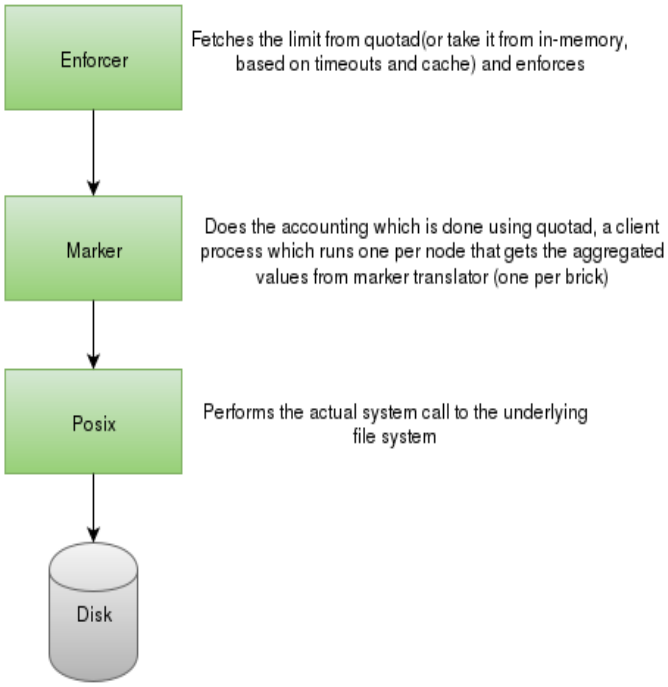

Fig. 5. Translators on server side

Quotad is a client process without a mount point. It has the same graph as client graph(client->dht->afr- >brick). In case of replica we take the one which shows higher size. We store inode-memory(limit) in inode context itself. Write operation is expensive if for every operation we contact quota daemon. So we have stored limit in inode memory as well. Accounting is done using the marker translator. Marker translator is present in each brick of the volume. Accounting process happens in the background and it doesnt happen while the file operation is being carried out simultaneously. Every time a delta value is calculated(the difference between the current size and older size) and then it is sent recursively upwards to the root of the volume, which is also stored as contri in every file/directory. During the initial lookup, extended attributes are created or whenever quota enable happens, a lookup is initiated and the extended attributes are created.

A. Quota Attributes

Extended attributes of quota[6] look like this :

- trusted.glusterfs.quota.00000000-0000-0000-0000-000000000001.contri=0x00000000000000000000000000000000000000000000000000000001
- trusted.glusterfs.quota.limit-set=0x0000000000500000ffffffffffffffff(5.0 is the hard limit thats set here)
- trusted.glusterfs.quota.size=0x00000000000000000000000000000000000000000000000000000001

In the above example:

- 'contri' xattr: 'contri' xattr contains the size of how much it contributes to the parent directory(recursive upto every directory upto root). Here, the value '00000000-0000-0000-0000-000000000001' before the .contri in the xattr corresponds to the gfid of the parent. About the value '0x00000000000000000000000000000000000000000000000001', '0x' represents hexadecimal and the first sixteen digits(or 64bits) corresponds to the actual size contributed, the next sixteen digits corresponds to the file count and the next sixteen digits corresponds to the dir count.
- 'limit-set' xattr: The 'limit-set' xattr, as the name says, helps to enforce(restrict) the usage on a specified directory. trusted.glusterfs.quota.limit-set=0x0000000000500000ffffffffffffffff(5.0 is the hard limit that is set here by the user) In the 'limit-set' xattr the first sixteen digits corresponds to hard limit and next sixteen digits corresponds to soft limit.
- 'size' xattr: trusted.glusterfs.quota.size=0x0000000000000000000000000000000000000000000001. The 'size' xattr corresponds to the size of the directory, where the first sixteen digits refer to the actual size, the next sixteen bits refer to file count and the next sixteen digits refer to the directory. Size xattr is present only for directories.
- Dirty xattr - In backend dirty xattr always will be zero. Whenever we need to account, we have to maintain consistency and dirty xattr is used for it. For example: Consider we have : '/dir/file' In 'file', contri is updated, and before we update 'dir', some problem happens(may be a brick has crashed), and so 'dir' is not updated. Now we have 'dirty' set to 1(whenever a write or some operation that modifies file happens, dirty xattr will be set to 1) and then once 'contri' is updated to 'dir', we change the 'dirty' value back to 0. This way it prevents from misaccounting. During lookup, we inspect the 'dirty' flag if it is zero, else(which means dirty is set to 1, probably an operation has happened and not updated successfully), we check the sum of size of files under the directory, update the size and contri in the directory(from leaf to root) and make dirty again set to 0.

The current mechanism stores count of objects/files as part of extended attribute of a directory. Each directory will have the number of files present in a tree with tree being considered as the root of the directory (Applies to subdirectories as well). Hence, crawling till root is not needed to retrieve the number of files(objects). There is something called as timeouts which also helps in the enforcement. Let's look on what is called as timeouts.

B. Timeouts

Memory cache size needs to be updated in order to make the enforcer work properly. For performance reasons, quota caches the directory sizes on client. You can set timeout indicating the maximum valid duration of directory sizes in cache, from the time they are populated. For example: If there are multiple clients writing to a single directory, there are chances that some other client might write till the quota limit is exceeded. However, this new file-size may not get reflected in the client till size entry in cache has become stale because of timeout. If writes happen on this client during this duration, they are allowed even though they would lead to exceeding of quota-limits, since size in cache is not in sync with the actual size. When timeout happens, the size in cache is updated from servers and will be in sync and no further writes will be allowed. A timeout of zero will force fetching of directory sizes from server for every operation that modifies file data and will effectively disables directory size caching on client side.

C. Challenges and the Enforcer Techniques

When you enable quota, glusterd initiates "find . | xargs stat", just to lookup on each file and create the xattr's. When you disable quota, we send "find . | xargs stat", which crawls each file, and removes the xttr. Consider we have a quota enabled volume. When quota is disabled on this volume let's see what are the steps being carried out.

- CLI will send the request to glusterd.
- Glusterd will initiate a backend process 'find / — xargs setxattr -x ....' to clean up all quota xattrs in the backend.
- CLI waits for the response from glusterd.
- Glusterd will continue to execute the operation that CLI initiated even when one kills CLI.
- As mentioned above, glusterd does a lookup on each file and then remove the quota xattr associated with the objects, but there could be problem under the scenario:
- The clean-up process may have terminated without completely cleaning-up the quota xattrs.
- One may try to enable quota before the clean-up process has completed.
- In simple when quota is enabled again, this can mess-up the marker accounting.

To be precise, let us consider an example, Example: Consider, we have /dir/file where 'file' is of size 5MB, so '/dir' and '/' will be accounted as 5MB. Here the cleanup process of cleaning xattr of 'file' is done, but failed to cleanup xattr on '/dir' and '/'. It may be because the process has terminated or due to a failure in node. Now if quota is enabled, 'file' will be accounted 5MB, this 5MB is added to '/dir' and '/' but '/dir' and '/' already has 5MB accounted which was not cleaned during cleanup and after adding 5MB the quota usage becomes 10MB which makes marker accounting

messy.. Hence as a solution to the above problem, Quota versioning came into picture.

D. Quota Versioning

A version number is suffixed for all the quota xattrs. This version number is specific to marker xlator (i.e) when quota xattrs are requested by quotad/client, marker will remove the version number which is suffixed in the key before sending the response. Previously the xattr's were just

- 'trusted.glusterfs.quota.size'
- 'trusted.glusterfs.quota.limit-set'. Now we are suffixing a version number at the end which makes,
- 'trusted.glusterfs.quota.size.<version number>'
- 'trusted.glusterfs.quota.limit-set.<version number>' where version number starts from 1 and keeps incrementing by 1 till N. The same concept applies to the 'contri' xattr as well.

With this approach, every time quota is enabled the xattr's gets attached with a new number and is treated like a separate session. Marker accounting goes perfectly with this approach, because, it takes care of xattr's belonging only to the particular time and works fine. Even though the problem of messing marker accounting is fixed, the previous xattrs are just left without cleaning up, leaving extra space. This can be fixed by checking up the version number in xattrs when quota is enabled and if xattrs with older version number exists, just cleanup them and then create new xattrs. Let's look on some more added features.

E. Usage of Quota Deem Statfs

One can create a report of the disk usage using the df utility by taking quota limits into consideration. To generate a report, one needs to turn on deem-statfs. In this case, the total disk space of the directory is taken as the quota hard limit set on the directory of the volume. Even if usable size is more(10GB), and the quota limit is set lesser(2GB), and when deem statfs is enabled, df gives, the limit-set quota value(2GB) as usable size. Deem statfs checks for the directories only for which quota limit is enabled. which is done in statfs lookup and accordingly the df output is shown. This is very helpful for anyone who sets quota.

F. Techniques that Needs to be Enhanced

- Performance Issue - Currently, when files are distributed across bricks, just for a lookup, currently we are sending, from client->dht ->replica- >brick, instead we can activate one process on one brick, which could increase the performance. This work is under progress.
- Recursive Directories Problem in Enforcer - For example: Consider the directory structure /1/2/3/4/5/6/7/f1. One has to keep checking till the root to check the limit and do the enforcement if it exceeds and perform accounting accordingly which could cause serious performance problem. In quota, we always need to crawl till the root to do the accurate enforcement. This needs to be fixed from the scratch.
- Issue in Enforcing - When a brick is down, then quotad, the client process do not get the aggregated value and it instead sends zero, and the write can happen more than the limit. Say, there are two bricks, b1 and b2, brick b1 has size 10GB written and the limit set on b1 is 15GB but b1 went down, so brick b2 thinks that 15GB is free in brick b1 and tries writing 15GB which could cause a problem in enforcement.

V. CONCLUSIONS

The Gluster file system is a revolutionary step forward in data management on every axis and in every dimension: absolute performance; scalability of performance and capacity; manageability at scale; ease of use; reduced cost of acquisition, implementation; daily operation, and per-terabyte for any particular level of desired redundancy or achieved-reliability. The complete elimination of metadata is at the heart of many of its fundamental advantages, including its remarkable resilience, leading to its reduced risk of data loss or data corruption down to conditional states near absolute zero by statistical calculation or logical extrapolation. Also, depending on the need, the features are exported to the users. One such important feature as seen in this article, is Enforcer. Enforcer finds quite a lot of real time applications in banking sectors, educational universities and generically places where sectors(or disk space) need to be allocated to the users.

As a whole, Gluster brings completely new technology that delivers on a wholly new philosophy for storage: the focus is on the compute host, not on the disk drives and shelves. This implies a new level of independence which guarantees new levels of performance, scalability, manageability, and reliability never before seen in any other storage system, with opportunities for integration with various technologies at the application level. To completely precise the entire article, here is the bottom line, "Gluster - data management for the 21st century, leaving the past behind and Enforcer, an important component in GlusterFS, which will never let you off limits and is extremely useful for the real time applications."

REFERENCES

[1] Gluster Whitepaper, 2010, "GlusterFS Architecture", [Online]. Available: http://download.gluster.com/pub/gluster/documentation /Gluster_Architecture.pdf. [Accessed: 22-Dec-2015].

[2] Dan Lambright, SA Summit, 2014, "Erasure Codes and Storage Tiers on Gluster", [Online]. Available: http://www.slideshare.net/Red_Hat_Storage/erasure-codes-and-storage-tiers-on-gluster. [Accessed: 19-Nov-2015].

[3] Gluster Community, 2015, "GlusterFS Documentation", [Online]. Available: https://gluster.readthedocs.org/en/latest/Quick-Start-Guide /Architecture/. [Accessed: 07-Oct-2015].

[4] Gluster Community, 2010, "Cloud Storage for the Modern Data Center, An Introduction to Gluster Architecture", [Online]. Available: http://moo.nac.uci.edu/ hjm/fs/An_Introduction_To_Gluster _ArchitectureV7_110708.pdf. [Accessed: 12-Oct-2015]

[5] Vijay Bellur, Vault Conference, 2015, "An Introduction to GlusterFS", [Online]. Available: https://lwn.net/Articles/637437/. [Accessed: 20-Jan-2016]

[6] Manikandan Selvaganesh, 2016, "GlusterFS - Quota", [Online]. Available: https://manikandanselvaganesh.wordpress.com/. [Accessed: 10-Feb-2016]