GlusterFS and its Distribution Model

Sakshi Bansal
sabansal@redhat.com

Contents

Why we need distributed file system…?

- Storing and accessing files in a client/server architecture.
- Uses multiple servers to store data and use multiple clients (local or remote).
- Organizes and displays files and directories from multiple servers as if they were stored in your local system.
- It is easier to distribute documents to multiple clients, provide a centralized storage system
- Client machines are not using their resources to store the data.

GlusterFS concepts

Brick	Brick is the basic unit of storage, represented by an export directory on a machine
Server	The machine which hosts the actual file system in which the data will be stored. A node is a server capable of hosting bricks.
Client	The machine which mounts the volume (this may also be a server).
Trusted Storage Pool	A storage pool is a trusted network of storage servers. You can dynamically add or remove nodes from the pool. Only nodes that are in a trusted storage pool can participate in volume creation

Volume	A volume is a logical collection of bricks. Most of the gluster management operations happen on the volume. Multiple volumes can be hosted on the same node.
Translator	These are building blocks of GlusterFS. All functions are implemented as translator They are stacked together for achieving a desired functionality.
Volfile	.vol files are configuration files used by glusterfs process. A set of translators stacked to achieve a functionality is stored in a volfile. Volfiles will be usually located at /var/lib/glusterd/vols/volume-name/.

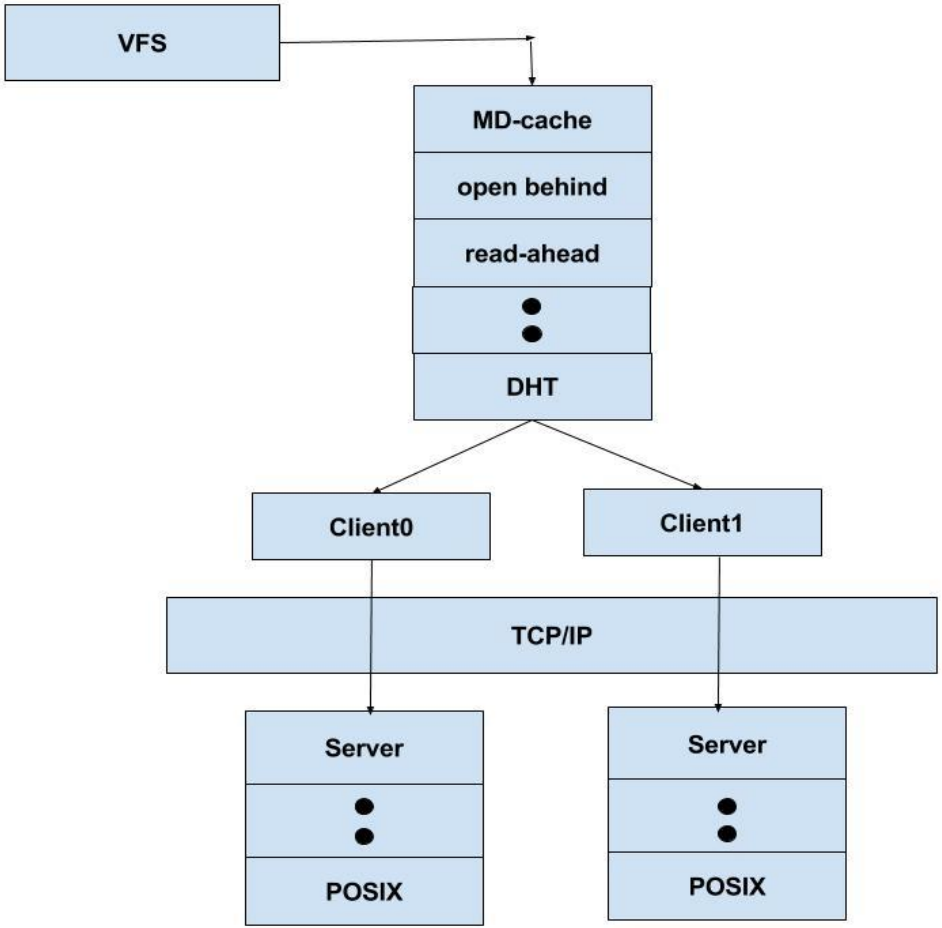
Introduction to GlusterFS

- GlusterFS is a distributed file system capable of scaling up to several petabytes and can handle thousands of clients. GlusterFS clusters together storage building blocks over RDMA or TCP/IP
- Aggregates disk and memory resources for managing data in a single global namespace.
- Based on a stackable user space design.
- GlusterFS and FUSE
    - GlusterFS is a user space file system and hence uses FUSE (Filesystem in user space) to hook itself with VFS layer.
    - FUSE is a loadable kernel module that lets non-privileged users create their own file systems without editing kernel code.
    - Gluster uses already tried and tested disk file systems like ext3, ext4, xfs, etc as the underlying filesystem to store the data.

- Create and start volume
  - Trusted storage pool consisting of storage servers that provide bricks to create the volume.
  - Every node has a daemon - glusterd which can interact with the glusterd on other nodes
  - We can also remove nodes from the trusted storage pool.
  - To create a volume we can use bricks from any number of nodes provided they are in the same trusted storage pool.
  - As a part of volume creation glusterd will generate client and server volfiles.
  - A volfile is a configuration file that has a collection of translators, where each translator has a specific function to do and together implement a desired functionality.
  - They are stacked in a hierarchical structure.
  - Translator receives data from its parent translator, performs necessary operations and then passes the data down to its child translator in hierarchy.

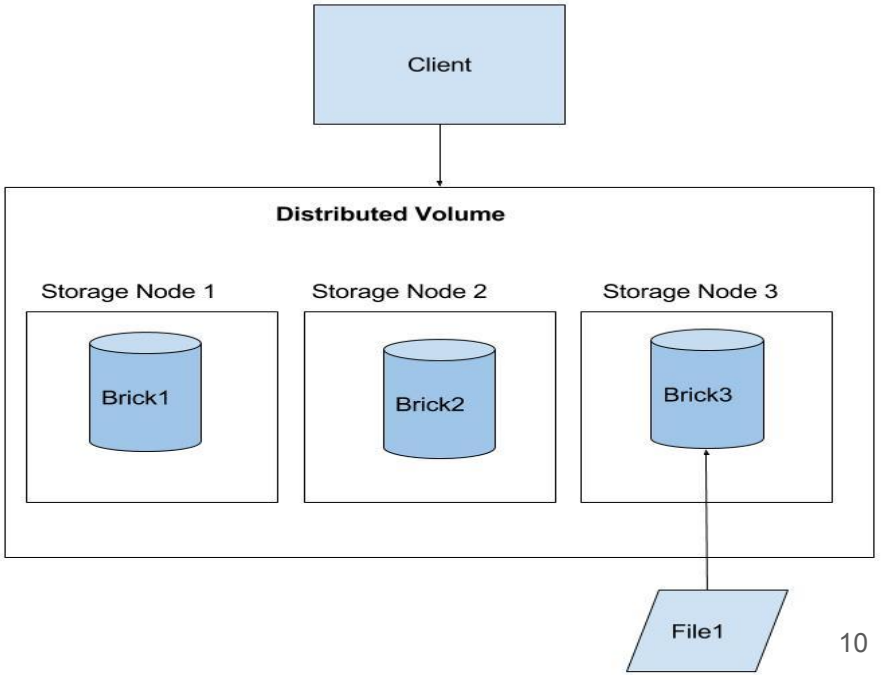
- Winding file operation of fop
  - Volume mount starts glusterfs client process.
  - This interacts with the glusterd process and gets a configuration file or the client vol file.
  - Glusterd also sends a list of ports on which the client can talk to each brick.
  - When we issue a file operation or fop, VFS hands the request to the FUSE module which in turn hands the request to the GlusterFS client process.
  - GlusterFS client processes the request by winding the call to the first translator in the client volfile,

- The last translator in the client volfile is the protocol client which has the information regarding the server side bricks and is responsible for sending the request to the server (each brick) over the network.
- On the server stack the first translator loaded in the protocol server which is responsible for getting the package.
- Finally the request reaches the POSIX filesystem.

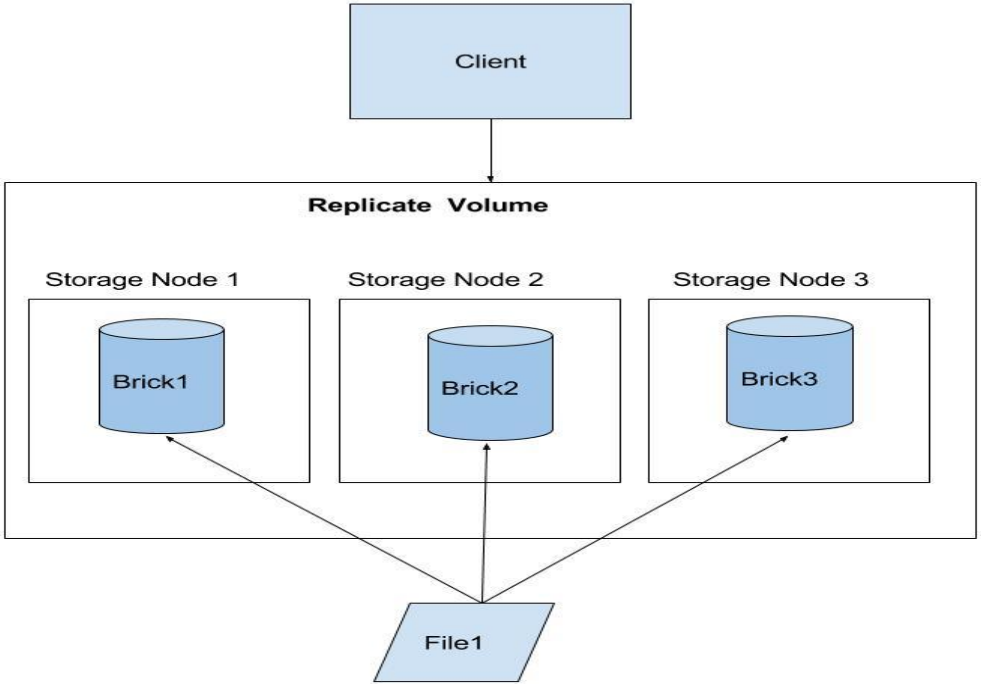


Types of volume

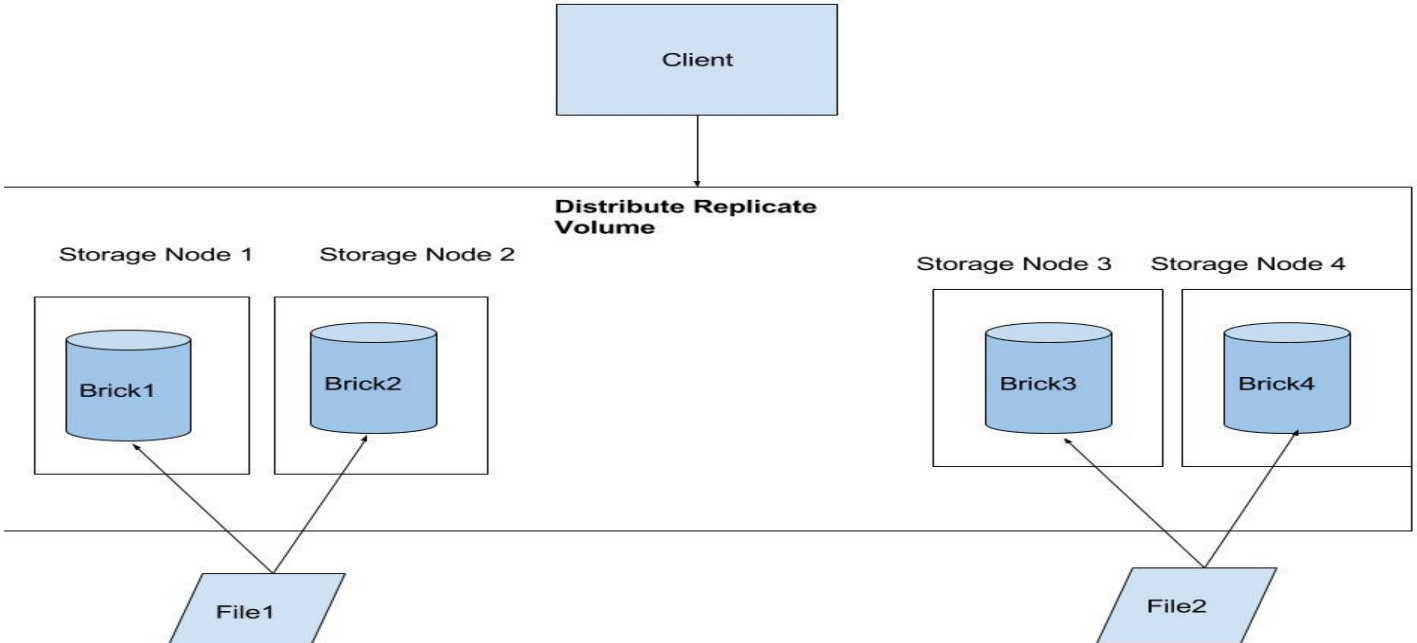
- Distributed Glusterfs Volume
  - This is the default glusterfs volume.
  - Files are randomly stored across the bricks in the volume.
  - Purpose is to easily scale the volume size.
  - There is no data redundancy
  - Loss of data.



- Replicated Glusterfs Volume
  - Overcome the data loss problem faced in the distributed volume.
  - Exact copy of the data is maintained on all bricks.
  - Number of replica pairs in the volume can be decided by client while creating the volume.
  - If one brick fails the data can still be accessed from its replica pair.



- Distributed Replicated Glusterfs Volume
  - Combines the advantages of both a distribute and a replicate volume.
  - Files are distributed across replicated sets of bricks.
  - Used when high availability of data due to redundancy and scaling storage is required.
  - If there are four bricks and replica count 2 then the first two bricks become replicas of each other then the next two and so on.



Introduction to DHT

- DHt is loaded on the client stack.
- All operations are driven by the clients which are all equal.
- No metadata servers or special nodes

- Additional information about file or directories are stored in the extended attributes or xattrs.
- Xattrs are filesystem features that enable users to associate files/dirs with metadata. T
- hey store information as key-value pairs.
- There are mainly two DHT related xattrs- linkto and layout.

- Directories
  - DHT creates directories on all the bricks in the volume.
  - Layout range is assigned and stored in the extended attribute called ***trusted.glusterfs.dht*** .
  - Range varies from 00000000 to 0xffffffff and each brick is assigned a specific subset of this range.
  - Layout is complete and healthy when the range 00000000 to 0xffffffff is distributed across the volume without any gaps or overlap.
  - The directory creation and the setting of layout is part of mkdir operation.

Here is an example of how layout is stored for a directory:

```
# file: ../export/t11/b1/a
trusted.glusterfs.dht=0x0000000100000000000000055554a1d

# file: ../export/t11/b2/a
trusted.glusterfs.dht=0x000000010000000055554a1eaaaa943b

# file: ../export/t11/b3/a
trusted.glusterfs.dht=0x0000000100000000aaaa943cffffffff
```

- While setting the layout we can either do a
    - **Homogeneous** setting of layout, where each brick gets the equal range
    - **Heterogeneous** weight assigning where based on the size of the bricks we assign the layout range to a brick. This means that larger bricks have larger layout, increasing the probability of data on these bricks.

- There are mainly two types of anomalies that can be seen w.r.t layout:
    - Holes - on a brick if a directory does not have a layout it is called a hole. If there is no layout on a directory no files can be stored on that brick.
    - Overlaps - all brick must have exclusive layout ranges, if the layout ranges overlap it is an overlap.

Files

- Unlike directories file have to be present on only one subvol.
- Find hash value of file and the brick on which the hash value falls, this is hashed brick.
- The brick on which the data file actually exists is the cached brick.
- For a newly created file the hashed and the cached brick will usually be the same.

- For example, if we create a file say ‘f1’ and its hash value is 0x5556ab12 then based on the previous layout the hash falls on brick3 and the data file would be created on brick3.

- While renaming a file the destination file's hashed brick may be different from the source file's hashed brick.
- In this case instead of actually moving the entire data file to the new hashed brick we create a linkto file.
- This is a 0 byte file which is created on the new hashed brick and has mode equal to _____T (no permissions except for the sticky bit 'T').
- The purpose of the linkto file is to act as a pointer to the brick where the data file actually exists (which is still located on the old hashed brick).
- They have an xattr called the *trusted.dht.linkto* xattr which stores the name of the brick on which the data file actually exists.
- Now the brick on which the linkto file exists is the hashed brick and the file on which the actual data file exists is the cached brick.
- All fops on the file will first land to the hashed brick and will be redirected to the cached brick by reading the linkto file's xattr.

FOP’s in DHT

- Mkdir
  - DHT first performs lookup to ensure that this entry does not exist .
  - DHT will first create the entry on the hashed brick, then proceed to creating entry on the other brick.
  - Once it has created the entries it will set the layout on all brick.

- Lookup
  - The objective of lookup is to get the layout and check the health of the layout of the directory.
  - Lookup on directory -
    - Lookup call is wound to all the bricks which will read the layout for this entry from the disk.
    - Merge the layout and check for any anomaly (hole/overlap).

        - If we find any of these anomalies in the layout lookup to heal the layout.
        - A selfheal is triggered which will find the brick which have anomaly and set proper layout.

- Lookup-selfheal
    - Creating the entry - On some bricks may the entry might be missing. If so it must create an entry on that brick and set the layout range on that brick.
    - Setting the layout - entry exists but layout is not set properly. If so, we need to set an appropriate layout range on that brick

- Read
    - DHT will start reading entries brick-by-brick.
    - Will discard linkto files and only read the data file.
    - Setting option ‘use-readdirp’ on DHT will display the file and the stats of the file.

- Rmdir
  - Entry must first be removed from all the non-hashed brick.
  - If removal from all the non-hashed brick was successful rmdir proceeds to the hashed brick of the directory.
- Rename
  - Renaming a directory always happens on the destinations hashed brick first and then proceeds to all the other brick.

- Lookup on file
  - If the entry is a data file we find the hashed brick and perform lookup only on the hashed brick.
  - If we do not find the file on the hashed brick we must send a lookup on all the bricks
  - If we find a file on some non-hashed brick,we will create a proper linkto file on the hashed brick that points to the the brick where the data file exists.
  - If lookup for the file on returns linkto file find name of the brick from xattr and perform lookup on the brick pointed by the linkto file.

- Create file
  - Like directories before creating files also, DHT sends a lookup to ensure that the file does not exist.
  - If no, find the hashed brick and create the file

- Unlink file
  - First send unlink to the data file, if it is removed successfully only then send unlink to linkto file.

Managing Scalability in GlusterFS

- Expand volumes
  - Increase volume size by adding more bricks.
  - Glusterd will generate volfiles corresponding to all the newly added bricks
  - Notify all clients to update their client volfiles to include these bricks.

- Rebalance
  - After adding new bricks we need to rebalance the volume.
  - Rebalance consists of two parts.
    - Fix-layout: Recalculate the layout of all directories so that the layout now spans to the newly added bricks as well. This is the fix-layout phase of rebalance.

    - Migrate-data: Once the layout of the brick changes, the hashed brick of many of the files may change as well. Hence we may need to migrate or move a file to its new hashed brick. This is the migrate-data phase of rebalance. However unlike rename command rebalance will actually migrate the file and not just create linkto files.

How rebalance works:

- Starts from the root directory to read the layout to check for anomalies. It will recalculate the layout to include the new bricks and assign this new layout to the directory. Once it has healed a directory it will read all its children.

- If the first child read is a directory it will repeat step 1. Keeps healing directories in a depth-first-search manner.

- When rebalance reads a file, it adds the entry to a queue for the file migration process to take care of it.

- On starting rebalance ‘n’ number of threads are spawned and are in a sleeping state. Once we add an entry to the queue it will wake up the thread. Each thread will read one entry from the queue and is responsible for migrating that file. It calculates the hash of the file and finds the brick on

which the hash falls. If this brick is different from the current brick on which the file resides, the file will be migrated to its new hashed brick. Once a thread has migrated a file it will read the next entry from the queue. If the queue is empty (i.e, no more files left to be migrated) the threads will go back to sleep state.

- A rebalance will only read those files that are present on its local bricks, this way multiple rebalance daemons will not migrate same file.

- If we do not want the files to be migrated we can also just run fix-layout option which will only heal the layout.

- Shrink volumes
  - Shrink the volume by removing bricks.
  - Once we issue remove-brick a rebalance is automatically triggered.
  - Once rebalance has successfully completed information regarding the bricks will be removed from the volfiles.
  - We can remove the brick in various ways:
    - Start - will trigger rebalance and once the rebalance completes and all migrates all files.
    - If we do not need the data stored on that brick we can force remove the brick.

- Replace brick
  - If a brick goes bad or we want to use another brick we can also replace a brick in the volume. However we will loose the data stored on that brick.

Reliability

To maintain availability and ensure data safety there are various options:

- Replicate volumes:
  - Replica pairs on different nodes.
  - Data is immediately replicated on all bricks.

- Geo-rep
  - Provides asynchronous replication of data across geographically distinct locations.
  - Replicates entire volume unlike AFR which is intra-cluster replication .
  - Useful for backup of entire data for disaster recovery.

Features

DHT provides various features that can be set or unset by the user. It can be set with the following command:

gluster volume set <volname> <key> <value>

- lookup-optimize: With this option set we can optimize these negative lookups for files as a lookup for file will not wind to non-hashed bricks if hashed brick did not return any result.
- Weighted-rebalance: with this option, the layout will be assigned to each brick based on the size of the brick.
- We can also change client and server log levels. Increasing the log levels will help to get better logs and debug in case there is a failure. For example setting option ‘rebalance-stats’ will log the time taken to migrate each file.

- min-free-disk: This is the percentage/size of disk after which the no more file/directory creation will happen on that brick. In case a brick directory creation will fail with ENOSPC and for file a link to file will we created and the actual data file will be stored in the next available brick. The default value is 10%, but can be modified by the user.

- Rebal-throttle: Sets the maximum number of parallel file migration allowed on a node during rebalance. The default value is normal and allows a max of [($(processing units) - 4) / 2), 2] files to be migrated at a time. Setting the option to lazy will allow only one file to be migrated at a time and aggressive will allow max of [($(processing units) - 4) / 2), 4]

- Quota: GlusterFS allows you to set limit on usage of disk space. Quota can be set on a volume to limit usage at the volume level or specifically to each directory to limit usage at the directory level, ie, you can just set limit to a single directory in the volume.

Performance Translators

There are some translators that help in increasing the performance in GlusterFS. They are loaded in the client volfile.

- Readdir-ahead: Once a readdir request is completed, preemptively issue subsequent readdir requests to the server in anticipation of those requests from the user. If the sequential readdir request was received, the directory content is ready and be immediately served. If such a request was not received the data is simple dropped.

- IO-cache: used to cache the data that is read maintained as an LRU list. When io-cache receives a write request it will flush the file from the cache. It also periodically verifies the consistency of the cached files by checking the modification time on the file.

- Write-behind: Improves latency of write operation by relegating the write operation to the background and returning to the application even as the write is in progress. Successive write requests can be pipelined which helps reducing the network calls.

Demo

A demo to see some basic functionality:

- How volumes are created
- How files get hashed and directory layouts are set
- Execution of fops
- Expanding and shrinking of volumes
- Replication

Q&A

Thank you!