GlusterFS Transaction Framework

Introduction

The transaction framework offers atomic semantics to translators so that they will be able to execute distributed and composed operations without needing explicit locking nor ordering management.

Motivation

Current solution for managing atomicity in GlusterFS is the use of inodelk/entrylk requests. While this approach has worked fine for some time, now it's beginning to show some limitations:

- **There isn't a common implementation**
  Each translator that wants to use atomic semantics needs to implement all the required logic itself. Most of the translators choose to use inodelk/entrylk locks because this was the first method implemented in Gluster and it has been well tested. However reimplementing it makes it prone to errors. Even there are some times where not all optimizations are implemented in all translators to avoid much of the complexity.

  This makes it difficult to replace the inodelk/entrylk logic by other options that could perform better, like leases, because it would require a lot of changes in many translators and would make it complex to test.

- **Translators use independent namespaces**
  Each translator uses a separate namespace where inodelk/entrylk is acquired, so a lock acquired by one translator cannot be reused by any other translator below it. This means that a single request could need multiple inodelk/entrylk to be acquired, one for each translator. This has a big impact on latency on some configurations and doesn't offer a real benefit on most of the cases.

- **More and more translators need atomicity**
  As new translators are implemented, more and more atomicity requirements emerge to be able to handle all kinds of workloads. With the current approach this is severely affecting performance of new features.

  Even already existing translators need to implement locking or other complex logic in places where previously there wasn't any synchronization to solve some newly discovered races.

Proposal

Definitions

- **Resource**
  A *resource* is any object that can be acquired in an exclusive or shared way to operate with it. Currently the only defined resource is an inode, but others can be defined if needed.

- **Subvolume pool**
  A *subvolume pool* is a subset of the subvolumes directly attached to an xlator. It can include all of them, or even a single one.

- **Subvolume picker**
  A *subvolume picker* is a policy used to select subvolumes from a *subvolume pool* in a specific order. There are some predefined policies to select subvolumes in a sequential way in the same order they are defined in the subvolume list, or in a round-robin way, or in a random way, or based on first available.

- **Task**
  A *task* is a simple operation that is executed by the transaction framework when some conditions are met. Each task has an assigned *subvolume pool* and *subvolume picker* from which subvolumes are selected to send fops. A quorum value and the number of subvolumes that must process the request are defined for each task.

- **Transaction**
  A *transaction* is an object used to encapsulate all the logic needed to guarantee atomic execution of one or more requests sent to one or more subvolumes. A *transaction* is composed by a list of *tasks* that are sequentially processed once the assigned *resources* have been acquired. Transactions can be nested if necessary.

- **Transaction library**
  The *transaction library* offers support to easily create and use transactions. It coordinates all required *resources* and manages *transactions* at all xlator levels.

- **Transaction user**
  The *transaction user* represents any xlator that makes use of the API provided by the *transaction library* to create and use *transactions*.

- **Answer**
  An *answer* is an abstract object that represents the result of one of the fops sent through the *transaction library*. It's a black box from the point of view of the transaction framework.

- **Answer group**
  An *answer group* is a set of *answers* that share a common result, so that they can be considered consistent. The degree of allowed differences is defined by the *transaction user* who is responsible for determining to which group belongs each *answer*.

Transactions

What we consider a transaction ?

For the purposes of the transaction framework implementation for Gluster, we consider that transactions are local to each translator, meaning that what one translator can consider as a transaction, another translator could execute it as a normal request. For example a *write* request for AFR will be considered a transaction, while the same write won't probably be a transaction for DHT.

A transaction is formed by two or more requests to be executed as one big operation without allowing other requests coming from other clients, or even from other users from the same client, to be executed in the middle. As a special case, a single request could be sent inside a transaction if, in some circumstances, an additional request will be needed depending on the result of the first one.

The requests composing a transaction can be sent in parallel or sequentially one after the other. But they should be carefully created to contain as few requests as possible to avoid unnecessary delays for other operations. Special care needs to be taken to only use transactions when we really need atomicity between two or more operations.

We don't consider a single request accessing only a single object as a valid transaction because in Gluster we already treat each single request as atomic. Only translators that have multiple subvolumes and/or split each request into multiple subrequests are candidates to use transactions.

Overview

As expected, the transaction framework will implement all necessary logic to guarantee atomicity along the whole lifecycle of a transaction. However it will also have some extra features that will simplify other operations related with the management of operations:

- **Dirty control**
  A dirty flag will be maintained by the transaction framework for each file to identify partial updates in case of failure. This flag will be updated automatically when needed depending on the tasks associated with each operation and the error code returned by the requests.

The flag will be independently tracked for the following groups:

- ➔ File data/Directory contents
- ➔ Entry metadata (basically iatt contents: mode, times, size, …)
- ➔ Gluster extended attributes
- ➔ User extended attributes

- **Versioning**
  A version number will be assigned to each file to track the changes made to it. This version can be requested by the xlators to compare it between bricks and help determine if bricks contain consistent data or not.

  The version will be tracked for the same groups than the dirty flag.

- **State tracking**
  Once a file has been detected as bad by an xlator, it will be remembered so that future accesses to the same file still keep this information. A heal procedure will be needed to clear this state.

  The state will be tracked for the same groups than the dirty flag.

- **Healing support**
  The information provided by the xlators about the completion of requests for each task will allow the transaction framework to determine which bricks are healthy and which ones not. This will be used to provide some callbacks to notify the xlators about unhealthy data.

  The transaction framework will also track special requests to heal bad bricks and control operations on bricks being healed.

- **Subvolume ordering**
  The transaction framework will provide an easy way to apply specific ordering to send the requests inside a task to the subvolumes. This way it will be easy to implement different policies depending on xlator needs.

- **Lightweight cache**
  Taking advantage of the control about locked inodes that the transaction framework will have, an small cache, only valid during the time period on which the inode is locked, will be implemented to allow easy optimizations to the xlators.

- **Special xattr tracking**

Since each xlator may need specific attributes to control its own integrity or special data, the transaction framework will provide a way to define these xattrs so that they can be retrieved all at once in a single request, reducing network round-trips.

Namespaces

Each transaction will be processed inside a namespace, in a similar way that we are currently processing inodelk and entrylk requests. This means that we could have different transactions running concurrently, even if they use the same resource, as long as they are created in different namespaces.

Translators are responsible to consistently use the namespaces to avoid conflicts. By default all transactions will be created in a common namespace shared by all xlators. Only those xlators that need special behavior for some requests will need to define a specific namespace (that should be a very marginal case, if used at all. Otherwise the benefits from the transaction framework will be lost).

State

Each transaction will have some space reserved to track its life cycle. This space can be freely used by the xlator to store whatever it wants. It's an efficient way of having access to the most important data through the callbacks used by the transaction framework.

This state is defined at the time of transaction creation.

Custom data

The owner of the transaction can also attach custom data to it. Data assigned to a transaction will be valid only during the lifetime of the transaction. Once it finishes, the data will be automatically destroyed.

This may be used for any data that doesn't fit well into the state area for any reason. Accessing it is not so efficient as using the state, but might be a better option for data that cannot be predicted at the time of transaction creation.

Branches

Once a transaction is created, requests are sent inside the transaction context. These requests will be routed through one or more subvolumes of the owner xlator. Each of the paths used for executing a transaction is called a branch.

A translator will have as many branches as subvolumes, however it can select which ones will be used for every transaction using a subvolume pool. It's not required that all transactions use all the branches.

Each branch will keep a queue of tasks per resource, created by all transactions, that will be processed sequentially.

Initialization

Each xlator will need to perform some initialization jobs in its init() function to be able to correctly use the transaction framework:

- **Define a heal notification function**
  Every time that the transaction framework detects a problem in a file, it will call this function to allow the xlator to initiate any required healing steps.

- **Create subvolume pickers**
  If the xlator needs to send the requests to its subvolumes in a specific order not already available by the predefined pickers, it can define new functions that will be used by the tasks to use the appropriate order.

- **Create subvolume pools**
  Some xlators may need to send some requests only to some subset of the total number of subvolumes available. In these cases, it can define these groups at the initialization time to avoid having to define them for each request (though it's also allowed).

  There are two special cases that do not need to be defined: a pool composed by all subvolumes, and a pool composed only by one subvolume. These are special pools that can be used without creating them.

- **Define required special xattrs**
  Since most requests that require transactions will need some special attributes to track special state, the transaction framework will allow xlators to define them at initialization time. Once defined, each time all transaction resources are acquired, these attributes, including the ones from all other xlators, will be requested automatically in a single request.

Resources

Each transaction will have resources assigned. Resources are "*things*" that are used by the operations issued inside a transaction and that need to be handled with care to avoid concurrent modifications by other clients. The most common resource will be an inode but other types of resources can be defined.

Types of resources:

- **Inode**

An inode represents a file, directory or any other file system entry. To simultaneously access them on multiple bricks or by multiple operations, an inode resource must be assigned to the transaction.

- **Other**
  For the current needs, it seems we have enough with inode resources. However the framework will allow the definition of new kinds of resources easily to accommodate future needs.

Each resource type will have a specific handling logic. More details about how inode resources work will be explained later.

States

Each resource will be in one of these states:

- **Idle**
  A resource has been assigned to one or more transactions but no action has started on it. Once we need to do some work on it, the resource will enter the *acquiring* state.

- **Acquiring**
  The resource is being acquired. In the case of an inode resource this means that an inodelk is being sent to all appropriate bricks.

  Once acquired, the resource will go to the *ready* state. If for some reason the resource cannot be acquired, it will return to the *idle* state and some notification will be sent to the owner of the transaction to let it know about the problem.

- **Ready**
  A resource is considered to be ready when it has been successfully acquired but no operation depending on it is being processed.

  When an operation depending on this resource starts, it automatically enters the *busy* state. If the resource needs to be released, it will go to the *releasing* state.

- **Busy**
  It means that some operations depending on the resource are being processed.

  When all operations related to the resource finish, the resource will return to the *ready* state.

- **Releasing**
  When the resource is not needed anymore, the required actions to release it will be executed.

  Once the release process finishes, the resource goes to the *idle* state.

Flags

All resources can have some flags. Some of them will be common to all types and others can be defined per resource type to modify some behavior.

Currently there is only one common flag that can be assigned to a resource:

- **Immediate release**
  This means that the resource will be released as soon as it gets to the *ready* state. No more new transactions using this resource will be allowed to continue until the resource has been released. Once released, if there were newer transactions using that resource, it will go to the *acquiring* state to start a new cycle.

  This flag is useful when some contention is detected and the current owner of the resource decides to release it to allow progress from another client.

A possible flag for inode resources could be the namespace, though I'm not sure if separate namespaces are really necessary or even problematic for the transaction framework.

Dependencies

In some contexts, a resource may be dependant or subordinate to another resource. A clear example of this relationship is the sharding translator. From the point of view of sharding, a high level file is, in fact, a set of multiple independent low level files that are managed together. At DHT or AFR level, these files are indistinguishable and are managed exactly equal.

In this case we have one inode that can be considered the master one, and all other inodes are slaves of the master. For sharding, the master inode corresponds to shard 0. All other shards are slaves.

In these cases, when we assign an inode resource to a transaction, we can specify a master inode bound to it. This implicitly converts the resource into an slave. All transactions must include at least one master resource. It's not required to have any slave resources on a transaction, but if they are present, its master resource must already be assigned to the transaction.

Note that what one translator will consider a master resource, another can consider it only a slave. For example sharding will consider the inode of any shard other than 0 as a slave, while AFR will consider it as master. In this case the transaction framework will treat this inode as slave, but in a transparent way for xlators. Only resources defined as master that have not been previously declared as slave will be really considered as master.

The benefit of having this relationship is that slave will almost always be acquired once the master one is acquired (the only possible exceptions will be self-healing and rebalance that can work at a deeper level where the master resource is not known). This means that when acquiring those resources we won't have contention on them, improving acquisition speed.

Additionally, we can use the same locking request to also retrieve the required metadata from the inode, minimizing the number of network round trips needed to have the inode prepared for operation.

When using transactions, all master resources need to be assigned before starting the transaction. If possible, it's recommended to also assign any slave resource at the same time. This allows earlier acquisition and may avoid some delays, but it's not a mandatory restriction. Translators are free to add new slave resources at any point inside the transaction lifetime.

Dependency considerations

Each xlator can define its own resources for each transaction and declare them as master or slave. The framework doesn't impose any explicit restriction here, but special care needs to be taken if one xlator assigns a master resource to a transaction as part of an operation that is already a member of a transaction created by a parent xlator, but has not defined the resource either as master nor slave.

This means that the transaction framework doesn't have any knowledge about this resource until an operation on another transaction has already been started after having acquired the needed resources. The new resource will need to be acquired while the parent one is already acquired. If not done carefully, this may lead to reverse order acquisition in some cases, causing deadlocks. This is something to be considered when writing code that uses the transaction framework, but due to the hierarchical structure of the gluster's translator stack, it seems really hard (if not impossible) to generate reverse order lock requests to cause a deadlock.

Transactions created on different namespaces are not affected reverse ordering because they are completely independent.

Custom data

Each resource can have generic data attached to it. This data is private to the translator that has defined the resource and can be used for any purpose the translator wants. This data is special in the sense that it will be valid only while the resource is acquired. When the transaction framework decides that the resource must be released, this data will be automatically destroyed.

Translators can only define data once the resource is acquired. Trying to do it before is an error. However this data can be queried at any time. If the resource is not acquired or the data has not been defined yet, no data will be returned. Otherwise the data will be returned. This way translators can optimize calls by first checking if some special data is present instead of always computing it (this can avoid some network requests to gather xattr data).

There are two kinds of data that can be attached to each resource:

- **Per branch data**

This data represents the physical values obtained from each branch (see later for a definition of a branch) of the transaction for a given resource. For inodes this means the extended attributes read from each of the lower translators.

This data is implicitly collected by the execution of some fops. Translators cannot directly request it, but getxattr requests will be intercepted by the txn framework and the cached data will be returned instead of sending the request to the bricks.

- **Computed data**
  This data is explicitly set by the translator. Normally this contains global values obtained from the combination of multiple branch data, but can be anything that the translator wants, even some state information.

  Here translators can store brick healthiness, virtual file sizes, marks, etc. This data is valid while the resource is not released, so it can be reused between fops if present.

  Translators can update this information, and even delete it, as many times as they want. Only restriction is that this can only be done while the resource is acquired.

Querying resource data is considered as a request inside the transaction. For branch data this is obvious because a fop will be explicitly sent to one or more of the subvolumes to retrieve the data. For computed data this is needed because the queried resource will be moved to the *Busy* state if it's already in the *Ready* state. This guarantees that the returned data is really valid and won't become outdated before sending real requests. This has two implications:

1. All resources must have been assigned to the transaction before requesting data from any of them.
2. All resources from a transaction must be acquired at the time that some data is requested from one of them. Otherwise the resource cannot go to the *Busy* state even if it was *Ready*. Not doing so could cause deadlocks between clients since the queried resource cannot be released and some contention may be detected which would require to release it.

Tasks

Once a transaction and its resources have been defined, xlators can assign tasks to it. A task is a simple procedure that will be executed sequentially with respect to other tasks inside the transaction and other transactions.

Requisites

Each task will have a set of associated requisites in terms of the number of subvolumes needed for some actions:

- **wait**
  This value represents the minimum number of subvolumes that must be ready (have completed previous tasks) before starting this task.

- **quorum**
  This value represents the minimum number of consistent answers that the task needs to be considered a success.

- **notify**
  This value represents the desired number of consistent answers that the task will wait before considering it a success. If this number is not reached, the task is still considered a success after having processed all answers if quorum is met.

- **requests**
  This value represents the number of subvolumes that will be initially used to process the request. If none of them fails, no more requests will be sent. However, if there are failures and the notify value cannot be reached, additional requests will be sent until we reach the notify value or there are no more subvolumes to send the request.

Flags

Tasks can be created with some attributes that modify its behavior:

- **Early abort**
  This flag means that the current task can be aborted at any time. This is particularly useful when a fatal error is detected while other requests are still being processed. This flag allows the transaction framework to immediately notify the failure without waiting for the pending requests to finish.

  Generally this can be done for read-only operations to improve performance in failure cases. If the request makes changes, most probably the xlator will want to wait until the pending requests complete to determine if it can go to an stable state or undo the changes.

- **Modify data**
  This flag indicates that the task will cause data modifications. The transaction framework uses this flag to control dirty and versioning of the file.

- **Modify metadata**
  This is equivalent to the *modify data* flag, but for metadata.

- **Modify Gluster xattrs**
  This is equivalent to the *modify data* flag, but for Gluster's private xattrs.

- **Modify User xattrs**
  This is equivalent to the *modify data* flag, but for user's xattrs.

- **Urgent**
  This flag marks the task as urgent, meaning that it will be processed as soon as possible (most probably just after the current task finishes). Normal tasks are always added at the end of the queue.

- **Nested**
  This flag forces the the task to be created inside the current one. This means that the task will be executed after the current one but before notifying the completion. Once the nested task, and any other task it might have created, finishes, the current one will be notified for completion.

  A failure in a nested task is propagated to the current task.

- **On healing**
  This flag is used to force the task to be processed on subvolumes that are being healed. This is especially important for modifying requests once a self-heal operation has been initiated on them. If there are no subvolumes being healed, this flag does nothing.

- **Healer**
  This flag means that the task is part of a healing operation, so it will be processed in all subvolumes, even if they are marked as damaged. The task is considered responsible of determining which subvolumes are really healthy and which ones not.

Lifecycle

The life cycle of a task is controlled by some callbacks that will be processed by the transaction framework depending on the events that happen in the task itself or even on the transaction.

These are the callbacks available to manage tasks:

- **prepare**
  This callback is executed once the resources assigned to the transaction have been successfully acquired and there are enough available subvolumes to process it.

  This callback can return an error if it's unable to do the needed preparations. In this case the task and the transaction containing it are aborted with an error. It can also return a value to indicate that the task is not really needed. In this case, the task is considered completed and the next one will be started. Otherwise the task is processed normally by picking some subvolumes and executing the *dispatch* callback for them.

In this callback it's allowed to create other tasks and even subtransactions.

- **dispatch**
  This callback is used by the transaction framework to send the associated operation to one of the subvolumes. This can be a simple fop or more than one fop that will be processed in parallel. The xlator must be sure that the fops won't interfere between them if executed in parallel. The transaction framework won't control them.

- **succeeded**
  Once the size of a group of answers have the notify value, or at least the quorum value when all requests have been sent, the transaction framework will execute this callback to handle the result of the operation. Note that a successful task only means that nothing failed while it was being managed, but the operation itself may have failed with an error.

  In this callback it's allowed to create other tasks and even subtransactions.

- **failed**
  This callback is executed when something prevents the task to succeed, or even if some problem happens during the *succeeded* callback. In this callback the xlator may try to recover a stable state or undo the changes.

  In this callback it's allowed to create other tasks and even subtransactions.

- **done**
  Once all requests have completed and all callbacks executed, this callback will be processed to do any cleanup or other operation the xlator may need.

Operations (fops)

Inside a task, the *transaction user* can send multiple fops. These fops need to be sent through the transaction framework, not directly to the lower translators using standard STACK_WIND macros, using supplied functions.

There are two main reasons to do so:

- **Take advantage of transaction framework's internal cache**
  Some requests can be optimized by returning data directly from the internal cache instead of issuing a real request to the bricks.

  Some special xattrs will already be requested by the transaction framework as soon as the resource has been acquired, so all this data can be directly taken from the cache. In other cases, the first request will need to physically go to the bricks, but subsequent requests from other tasks or transactions will be served from the cache until the resource is released.

- **Intercept request answers**

When a request gets answered, the transaction framework will be able to do an initial handling and forward it to the appropriate notification callback.

An interesting thing to consider here is that, in some cases, a *lookup* fop will be processed inside a transaction context created by an upper translator (basically for inode revalidations and pre-ops). Even if the translator doesn't need to use transactions, this can have a big benefit since it can trust returned data from multiple subvolumes because it can be sure that no other operation has been partially executed in some bricks.

API

One of the main objectives is to create an easy to use framework for all translators. We also want to remove most of the error handling burden from the translators, so almost all functions do not fail in a fatal way: the returned values for most functions are always valid to be passed to other functions, so no constant error checking is needed.

[Full proposed API is being worked on]