Ganesha Request Handler Design

1. Introduction

Here we describe the functioning of a backend coordinator for the crud_cache, used by NFS-Ganesha for its data backup, called the **Ganesha Request Handler (GRH)**. The main goal of **GRH** is to take in requests from the crud_cache and dispatch these requests to workers, which will execute them in the appropriate context.

A general instance of execution corresponds to the following:

- Client wants to copy a file from its file system to the server, or vice-versa
- crud_cache receives the request to copy
- **GRH** takes up the request and call the backend library to complete it
- **GRH** gives back the library's return code to the crud_cache
- crud_cache stores the return code
- Client retrieves the return code

There are multiple advantages to this approach:

- Scalability, regarding the amount of data the server can hold,
- Parallelism, since the crud_cache is on the client's side, and **GRH** is on the server side, all requests can be parallelized and easily managed,
- Flexibility, as the backend libraries are on the server side, adding, removing or modifying them is easier.

2. Terminology

2.1. Key-Value Store

A key-value store is a way to hold data (called the “value”) referenced by an accessor (called the “key”). For the store to be properly usable, each key must be unique, and reference a unique set of data. For this reason, we can use it to simulate a file system, as the path of each file and the referenced content are unique.

Specifically, we can use it to hold the metadata of files and directories, which are the information surrounding a file (when was it created, what is its length, who can access it, …). This means that in a specific directory, we can duplicate an entire arborescence by retaining only the metadata information, and not the actual files.

This is useful to hold large file systems, which may correspond to multiple terabytes or petabytes of data. We can do this without holding everything directly in the filesystem, as what we need is just a way to show an arborescence to the user, and then retrieve the actual files from elsewhere when asked for it.

Regarding our tool, we use two of these stores for distinct purposes:

- The Key-Value Store NameSpace (KVSNS[1]), which holds the metadata of the simulated file system,
- an Object Store, which holds the actual data referenced by the KVSNS. There are multiple possibilities for this object store, for instance we can use the cloud (through Amazon’s S3 servers[2]) or magnetic tapes (with the Phobos software[3]).

2.2. Crud_cache

The crud_cache is a cache that allows one to hold data in a Lustre Hierarchical Storage Management way, meaning the data stored in it can be either in the cache, and directly accessible, or in the object store (which we will call archive from now on). As such, a file can be in 3 possible states:

---

[1] https://github.com/phdeniel/kvsns
[2] https://aws.amazon.com/s3/
[3] https://github.com/cea-hpc/phobos

- cached, the data is only available in the cache,
- synchronized, the data is in the cache and the archive,
- archived, the data is only available in the archive.

To attain these states, there are 4 operations available to a user:

- Create, to create a file in the cache,
- Archive, to copy a file from the cache to the archive,
- Release, to remove a file from the cache (file must be in the archive),
- Restore, to copy a file from the archive to the cache.

Finally, the crud_cache is destined to manage Object Stores that support the CRUD semantic, meaning Object Stores that can perform the basic **Get/Put/Delete** operations. However, it is not necessary for those stores to support **pread/pwrite** operations, which are random read/write operations.

2.3. KVSFS

The Key-Value Store File System is the combination of the KVSNS and crud_cache, and allows one to store data in a HSM object store. The KVSNS holds the metadata and represents the filesystem as a posix namespace, while the crud_cache will hold the file's data. With these two mechanisms, the KVSFS corresponds to an HSM filesystem, usable as a normal file system.

Regarding the KVSNS, we also define and use a specific configuration file, called the **kvsns.ini** file, which holds all relevant information regarding the backend libraries used, the server address and ports, or the external softwares used.

2.4. Queue Manager

A queue manager is a software that can manage data ordered as a queue. The main operations associated with one are “enqueue” (to put data into the queue) and “dequeue” (to retrieve data from the queue). As for normal queues, these ones are managed in a “First In First Out” way, meaning the first data enqueued is also the first dequeued, and if you want to dequeue the second batch of data, you have to dequeue the first beforehand.

For our purposes, the queue manager will be used to hold the requests, meaning it should be the intermediary between the crud_cache and **GRH**. For this reason, we need a queue manager that is both scalable and parallel. Moreover, since it should be accessible by multiple users at the same time, it is necessary to daemonize this process.

Regarding our tool, we mainly consider using Celery[4].

[4] https://docs.celeryproject.org/en/stable/

3. Requests Management

3.1. Requests types

There are 3 kinds of requests the crud_cache can push to **GRH**, and they all follow the CRUD semantic: **Put**, **Get** and **Del**. **Put** will put an object, either to the archive or the cache, **Get** will retrieve an object from the archive or the cache, and **Del** will delete an object from the cache.

3.2. Workflow

When trying to execute a request, the crud_cache will register this request. This operation will first try to enqueue the request in the request queue, called the ‘request queue’, which **GRH** will take care of. Then, it will return a **request ID**, corresponding to the patch of request enqueued.

On the side of **GRH**, when a request is enqueued, the queue manager will automatically spawn a worker to take care of it. That worker will then call up the backend library to actually execute the request.

Finally, when the client wants to retrieve the result, it uses that same **request ID** and asks **GRH** for the request’s status. Multiple statuses are possible:

- **Running**, the request is still being treated,
- **Completed**, the request has ended without any problem,
- **Failed**, if the request has ended, the error code and an error message will be returned.

3.3. Execution

When a worker is spawned alongside a request, it will call the backend library to try and execute the request. This will use the library’s context **GRH** keeps around to start the proper action, and wait for its failure or completion.

3.4. Synchronization

As the queue is managed by a daemon, and **GRH** spawn workers to manage requests, synchronous or asynchronous requests are managed on the client's side, by waiting until a request completion. Since the registration process only encapsulates the request ID generation and the enqueuing of the request, it returns immediately. As such, managing the request synchronously or asynchronously comes down to waiting on the return queue until the request's result can be dequeued.

3.5. Request cancel

Cancelling requests can be managed like the other types of requests, meaning we consider a new type of request for the **Cancel** operation. For this operation to be meaningful, it requires a **request ID** that is only obtained after the request to be cancelled is at least managed by a worker.

If a cancel request is launched, a message will be sent to the worker managing the request to stop the execution. that worker will then send a new request to the backend, asking that the soon-to-be-canceled request actually gets canceled. This new request will be passed upon the backend, which means the cancel request only makes sense if the backend can manage cancel requests.

4. Initialization

4.1. Queue Manager

As explained, we want the queue manager to be daemonized. Therefore, it is necessary to initialize it beforehand. In the case of Celery, it must define the backend and broker used, which for our case will be Redis for both of them. Moreover, this daemon

4.2. Ganesha Request Handler

Ganesha Request Handler is also used as a daemon, and must also be initialized beforehand. It must define a port and address to be used as server, alongside the Celery backend and broker.

As part of its own initialization, **GRH** will also load the backend libraries, according to a configuration defining all the libraries available. When done, it will keep the context of each library around.

Then, when a request is received, **GRH** will push the request to Celery, alongside the backend asked for, and the context for that library, allowing the worker to work independently from that point onwards.

4.3. Backend Library

When the backend library is loaded, **GRH** will also call the backend's init function. In it, the library should create its context. If necessary, the backend library can define parameters in the **kvsns.ini** file. When done with its init function, the library will send the context structure to **GRH** for safekeeping.

5. Error Management

5.1. Request errors

Each request will be treated by the backend and the associated object store. At that point, the request is given to the object store, and the worker will wait for its completion. Then, the result given back by the backend library, including the errors, will be considered as the request result.

5.2. Functioning errors

The functioning errors (meaning internal errors not directly related to the requests) are not the same as the request errors, though they overlap in the case of **GRH**.

5.2.1. crud_cache

There are two functioning errors that can occur in the crud_cache:

- during the registration process, in which case the request is not added to the request queue, and an error is given back to the caller.
- during the get result process, in which case the request result is left in Celery for later retrieval, and an error is given back to the caller.

5.2.2. Ganesha Request Handler

Since **GRH** isn't directly linked to the crud_cache, if an error occurs in it, it cannot send back the error or directly tell the client the request cannot be fulfilled. As such, the simplest way of dealing with functioning errors is to consider them as request errors, meaning if **GRH** has such an error while treating a request, the error code and message will be given back to Celery.

5.3. Crash recovery

Since the crud_cache is on the client's side, its crash recovery capacity will not be detailed here.

Similarly, since the queue manager is an external software used as a daemon, it is its job to recover the information in case of a crash, and thus its crash recovery capacity will not be detailed here.

Since the queue manager spawns a worker that will take care of a request, if the worker crashes, then the queue manager is responsible for crash recovery.

6. General workflow