vReaper 2.0 proposal

Proposal v1.0
28.03.2019
Contributing authors: Martin Barisits, Cedric Serfon, Mario Lassnig

The Reaper is the component in Rucio responsible for the physical deletion of files. While the component itself is delivering high deletion performance, for it to work efficiently it needs considerable manual effort to distribute the workload intelligently. The lack of this effort leads to considerable impacts on the deletion performance, mostly leading to the fact that the deletion is not happening on RSEs where it should happen. The re-design of the Reaper should address this shortcomings.

Current issues

1. The deletion workload needs continuous manual redistribution to achieve proper deletion performance. This is done by assigning reaper instances to operate on subsets of RSEs. Not continuously doing this results in reaper instances being completely idle, while others have millions of files in their backlog.
2. There is no failover strategy. This is due to the manual splitting of the reaper daemon instances and the assignment of a subset of RSEs (RSE expression) to each one. For example, ATLAS runs ~10 different instances of the deletion daemon, each being responsible for a specified subset of the RSEs. Should one of these daemons crash, there is no failover to a different daemon due to the static split of RSEs. Moreover, the manual splitting can lead to different reapers running on the same RSEs in case of overlapping RSE expressions.

Objectives

To address the current issues, the following three objectives have been specified:

1. **Global performance**: All reaper daemon instances should have a steady deletion performance. A situation where some of them are busy and some are idle should not occur, if there is a deletion backlog to operate on.
2. **Protection of storage**: Measures need to be in place to protect the storage from deletion overload. It should be possible to specify the maximum number of threads concurrently interacting with a storage element.
3. **Intelligent distribution of workload**: The global deletion workload needs to be distributed intelligently. RSEs having a large deletion backlog need to be handled with a larger priority than RSEs with a very small backlog. At the same time, starvation of RSEs cannot happen either.

Proposals

Currently there are two proposals which are in large parts similar. The general idea is that all daemon instances work on the entire deletion workload, thus there is no fixed assignment of RSEs to specific daemon instances. The splitting of the workload is based on the heartbeat concept also used in other Rucio daemons. Thus each daemon is assigned a specific partition of the workload, based on the hashing of the replica names. This addresses **objective 1**.
The second objective requires that not too many daemon threads work on the same storage element. This is done to protect the storage from overload. The technical design of this is described later in this document, but is also similar in both approaches (See Heartbeat extension)
Objective 3 is where both proposals diverge: The first proposal relies that all daemons are responsible to make the selection decisions on their own, while the second proposal relies on a centralized daemon to prepare the list of replicas which are eligible for deletion.

Proposal 1 (Shared)

This section explains the workflow of each thread in the reaper daemon.

1. Report heartbeat (without `payload`). This heartbeat is just for information to be able to see how many threads are running in total.
2. Based on the RSEUsage, prepare a list of RSEs needing deletion. These are the RSEs which are over the deletion watermark OR RSEs which have epoche tombstone replicas OR RSEs which are in greedy deletion mode → Create `rses_deletion_needed` list. Based on this list, a prioritized list should be created. Some RSEs might need significant deletion workload, while others only need very little. Doing this intelligently is key to fulfill **objective 3** and **objective 1**.
   Later on the thread will iterate this list and execute one deletion cycle (deletion of ~500 replicas) on each RSE in the list. It might be necessary to give some RSEs additional cycles since more deletion needs to be done. It is important to have each RSE in the list at least once, so not to starve any RSE. The list should be randomized in order to re-distribute the load on startup of all daemons.
3. For each `RSE` in the `rses_deletion_needed` list
   a. With `list_payloads` check if there are already sufficient threads working on this RSE. (For example, N < 40) This fulfills **objective 2**
      If there are sufficient threads, continue with the next `RSE` in the list.
   b. If not, report heartbeat with `payload=RSE` which gives the `assign_payload_thread` and `payload_nr_threads` for exactly this `RSE`.
   c. The rest of the deletion cycle is similar to the current reaper workflow with the difference that `list_unlocked_replicas` is called with `assign_payload_thread` and `payload_nr_threads`. It is foreseen to

change `list_unlocked_replicas` to do `SKIP LOCKED` queries and immediately "reserve" the replicas by changing them to `BEING_DELETED` immediately. By skipping locked rows, no two threads will access the same replicas.

d. Reminder: Inside the deletion cycle the behaviour is similar to now.

Comments

- Doing step 2 intelligently is key in achieving the right deletion performance. Thus RSEs in need of massive deletion, should always have the maximum number of threads possible.
- Due to timing issues multiple threads might call `list_payloads` in step 3a at the same time, potentially resulting in too many threads for a given cycle. However, these events are statistically unlikely due to the randomization.
- Due to timing issues in step 3b different threads might not see each other and essentially declare the same replicas as `BEING_DELETED`. This would result in multiple deletion attempts for the same replicas. Other than wasted resources this is not an issue though.

Proposal 2 (Preparer)

Some information can be found here
The reaper is split into 2 different daemons :

- A preparer that get the list of files to delete
- The actual reaper that does the physical deletion and remove the replicas from the catalog

Preparer :

1. The preparer runs over all the RSEs and get the `needed_free_space`
2. For each RSE it calls `list_unlocked_replicas` with option `bytes=needed_free_space` and the replicas obtained are marked as `BEING_DELETED`. The number of files in `BEING_DELETED` state should scale with the number of "actual" reapers to prevent starvation from the latter.

Actual reaper :

1. Each worker queries a list of `BEING_DELETED` files. The partitioning is based on a hash of the replica name using the same heartbeat mechanism without payload that is described in the first proposal.
2. Once the reaper got its partition, it chooses the order in which the RSEs will be processed based on a share depending on how many replicas need to be deleted (e.g. if there are 4 RSEs A, B, C, D with resp. 200, 100, 50 and 50 files to delete, A will be selected 50% of the time, B 25 % and 12.5% for C and D). Once the RSE is chosen, the reaper checks how many reapers already run on the same storage (based on heartbeat with payload) and if below the threshold, it sends an heartbeat for this storage. Otherwise, it goes to next storage. A cycle timeout is defined. If after this timeout not all

the replicas are deleted it goes back to 1) (this is to prevent a bad storage to completely clog the system.
3. The rest is similar to the first proposal

Heartbeat extension

The heartbeat extension allows each running thread to additionally report a parameter to the central database. This parameter can be used as information on what data the thread is working on.

The `live` method of the heartbeat module needs to be expanded with a payload field. If a payload is given, the return dictionary is expanded with a `assign_payload_thread` and `payload_nr_threads`. This is essentially a sub-partition based on all executables with exactly this payload.

```
def live(executable, hostname, pid, thread, older_than=600, hash_executable=None,
         payload=None, session=None):
```

Additionally a new method is needed which returns a count of active threads for each payload for a given executable.

```
def list_payloads(executable, payload, older_than=600, hash_executable=None, session=None):
```

Decision

In the development meeting on the 04.04.2019 it was decided to follow the implementation of the first proposal.