GLUSTER NSR
(New Style Replication)
Design Document

Jeff Darcy
(jdarcy@redhat.com)

Avra Sengupta
(asengupt@redhat.com)

Venky Shankar
(vshankar@redhat.com)

TABLE OF CONTENTS

1. Feature Summary
2. Introduction
3. Architecture
    Overview
    1. Leader Election
    2. NSR Client
    3. Journaling
    4. In-Memory Journal
    5. NSR Server
    5. Reconciliation
4. Appendix

1. Feature Summary

NSR or New Style Replication is a server side replication mechanism, which provides better fault tolerance and improved performance.

This feature will reduce, the client side latency, owing to NSR's replication mechanism, which leverages server side bandwidth. This design principle, will not only improves performance, but will also make NSR split brain resistant.

2. Introduction

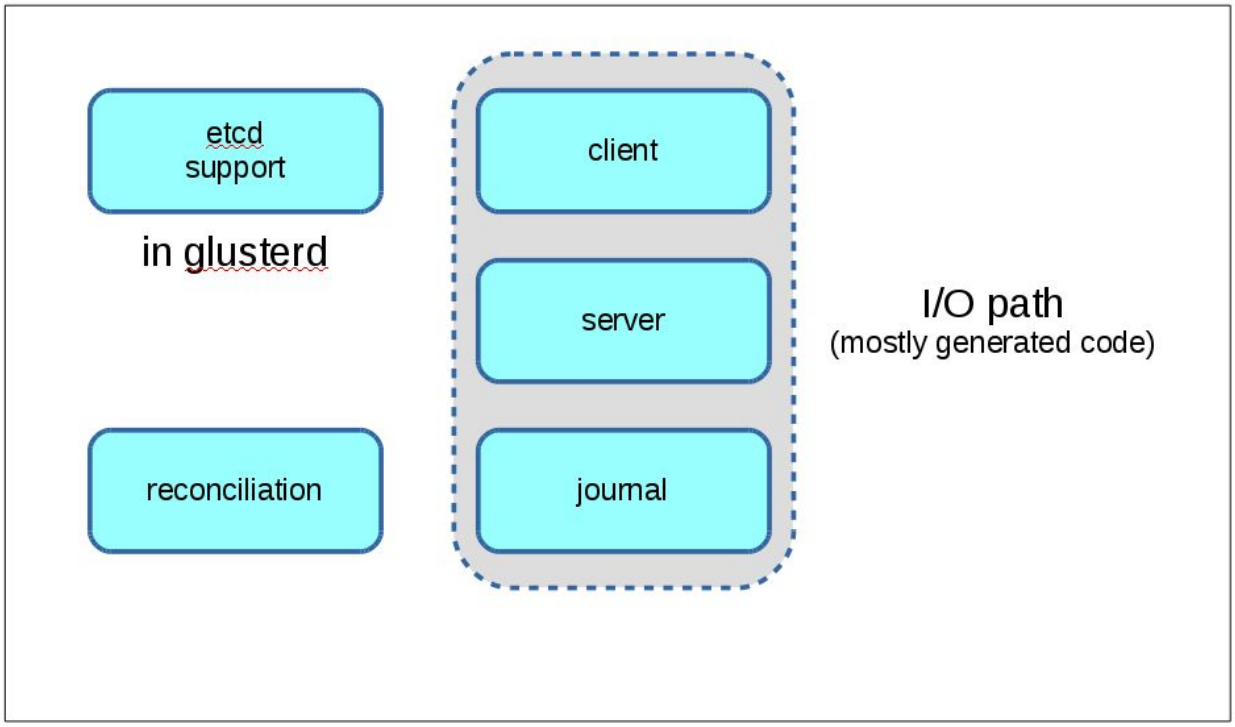
New Style Replication relies on two basic virtues: Journalling, and leader based coordination. Instead of sending data to all the servers of a replica, the client will send data to one server(the leader, which will be elected via etcd) in every replica subvolume. The leader will then forward the data, to other servers in the replica subvolume, and propagate the replies back to the client. The reads will also be sent to, and processed by the current leader.

This approach along with a precise operation based journalling, will enable NSR to perform data recovery, and eliminate split-brain scenarios. It will also help increase client's throughput, as it will now be able to use it's full bandwidth to perform ops(reads/writes) on only one server.

3. Architecture

Overview

In this section we will cover NSR architecture, identify and explore each of the modules that will act as building blocks for the feature.



1. Leader Election

We will be using etcd for leader election. etcd provides TTL(Time To Leave) on objects, which enables us to lease leadership to servers for terms.

A monitor cluster(etcd cluster) is created, and is used to save the term value, and store the leader key. The leader key is watched by all servers in the replica group. If the key is empty, any server can write its id on it and thus get elected as the leader of the replica group. The

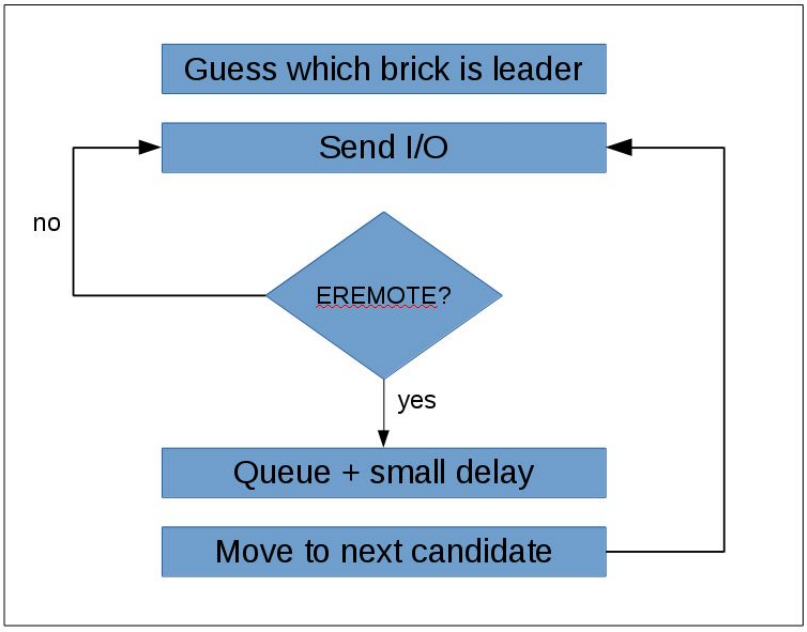
written value is given a TTL that removes it after a set interval, and the elected leader must rewrite it periodically to remain elected and renew it's lease. By the use of etcd's atomic compare and swap operation, there is no risk of a clash between two instances being undetected.

The etcd cluster will also be used to store information about the journal's terms.

Note: The above exercise needs to be done for every replica group in the subvolume, as every replica group will have it's own leader.

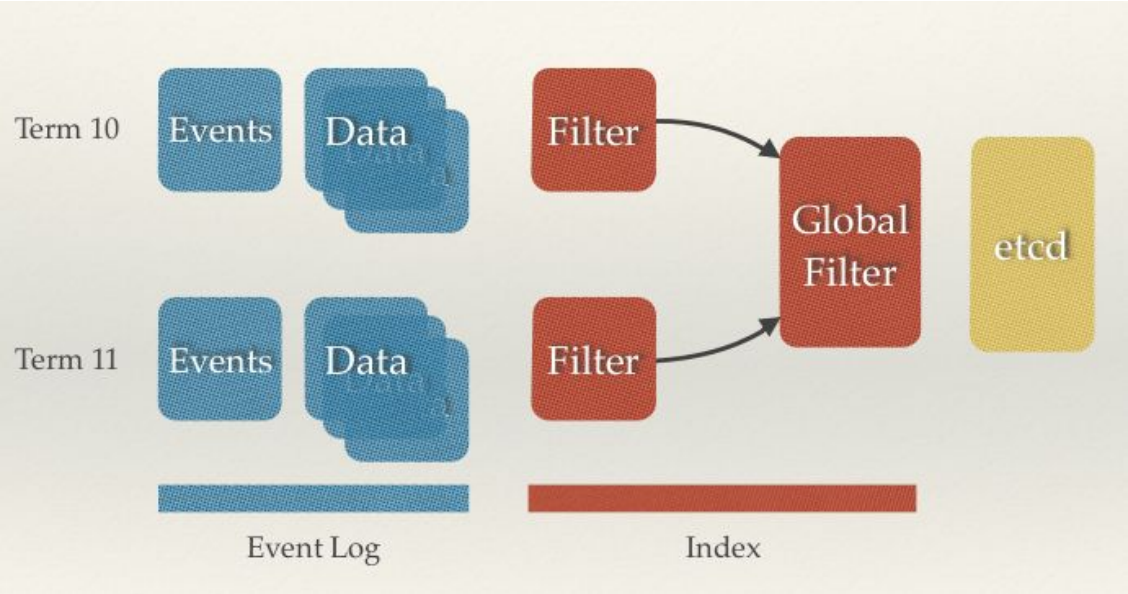
2. NSR Client

The nsr-client translator will help the client figure out which server in the replica group is the leader, and send the write and read operations to the same. It needs to check the return code of the operation, and accordingly retry on a different brick if it receives an error code stating that the brick is not the current leader.

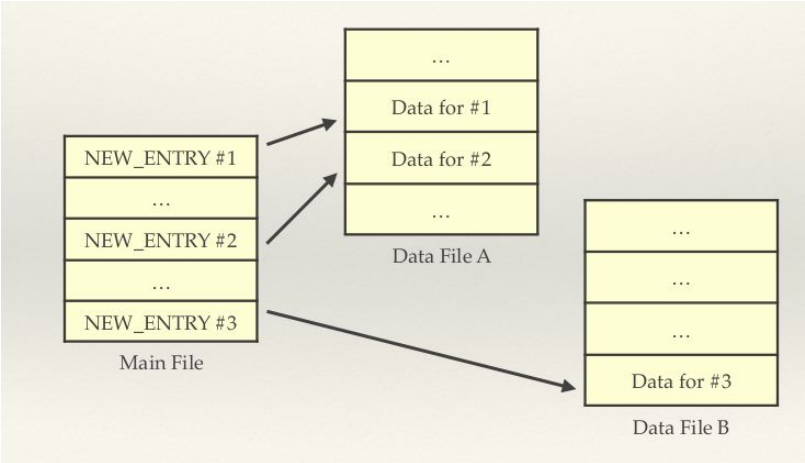


3. Journaling

NSR will rely on a full data journaling model, which will make its reconciliation much more robust. The journal comprises of two parts, the event log and the index.



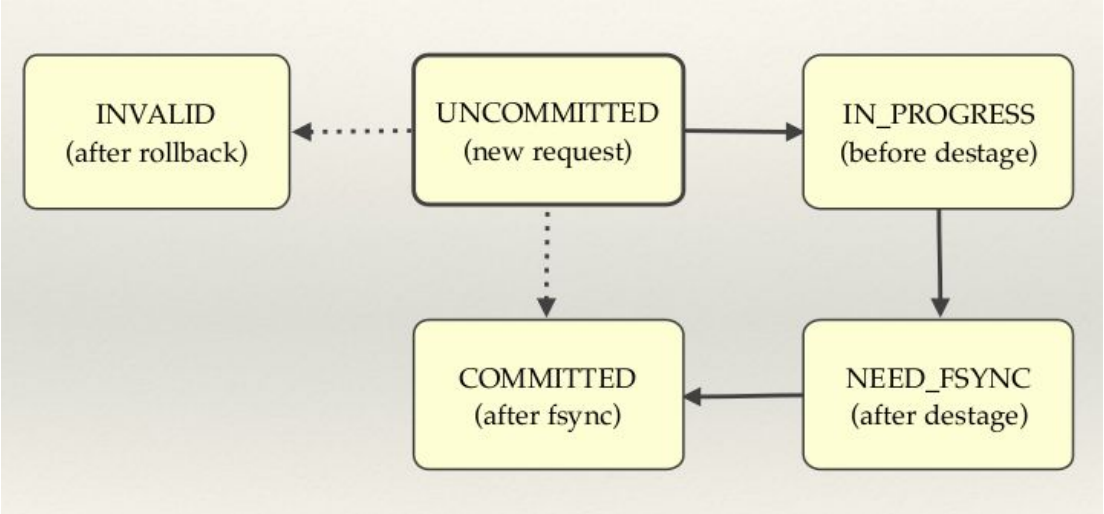
a. **The Event Log :** The event log is divided into terms, and within each term a series of events corresponding to user I/O requests is stored. The events themselves are stored in one file, while bulk data is stored in one or more supplementary files.



Each term's main event-log file contains short event records, which may contain pointers to one or more supplemental data

files. Each such pointer consists of an ID, identifying a particular data file, plus an offset into that file.

An event once it enters the state machine, can go through various state changes throughout it's lifecycle.



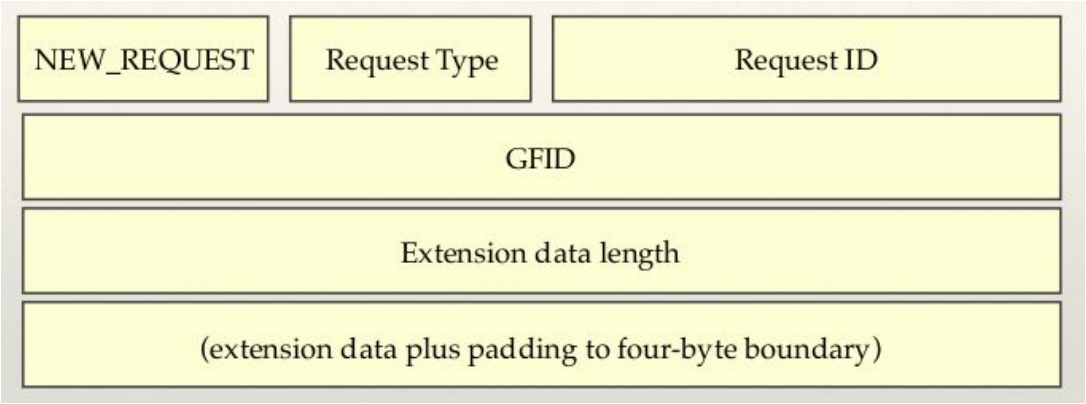
i) *Uncommitted* : This is the first state every Journal Entry is going to be in, when it's first introduced in the "state machine". This also means that this particular Journal Entry has not yet been acted upon and the actual fop is still pending.

ii) *In Progress* : This is the state that the Journal Entry is moved into, right before the actual fop is performed in the Data Store. This enables us to differentiate between a Journal Entry that has not yet been worked upon, from one that might be in any state of modification as part of the fop.

iii) *Waiting For Sync* : This is the state where the Journal Entry will be moved to, once the actual fop is performed, but a fsync is still pending. This means that the data might or might not be in the disk right now, but the fop is successfully complete.

iv) *Committed* : When a sync comes, all journals till that point, who were in “Waiting For Sync” state, are moved to “Committed” state. This completes the lifecycle of the Journal Entry.

v) *Invalid* : When a Journal is in Uncommitted state, and has not yet been acted upon, and a rollback request for the same comes, that particular entry is marked as “Invalid”, suggesting that this particular Journal Entry will not be acted upon.

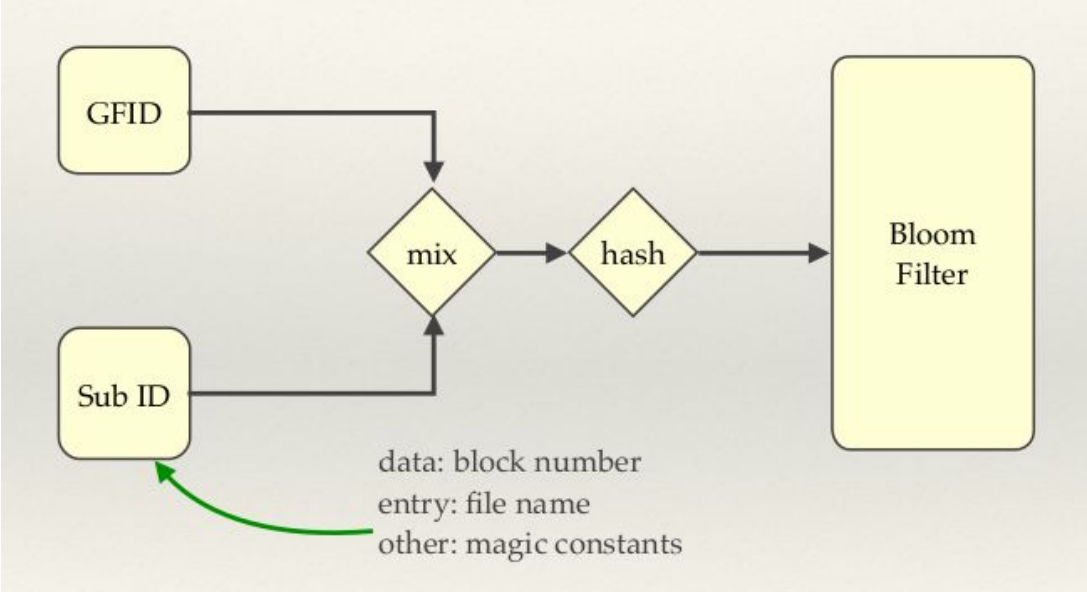


All entries in the event log, start with a four-byte header, starting with an event type. The other contents of the header can vary depending on the the event type. A Request ID in the above header will be used to associate all related events with one another.

There will usually be multiple terms “in play”, at once, but only one term will be the current term. Any term that ceases to be the current term, will stop collecting new request events, but might collect other internal events related to destaging or reconciliation. Once those events are finished, the term is considered as completed, and is kept around only for reconciliation.

A term can be deleted, when two conditions are met - it’s not needed locally, and it’s not needed for reconciliation.

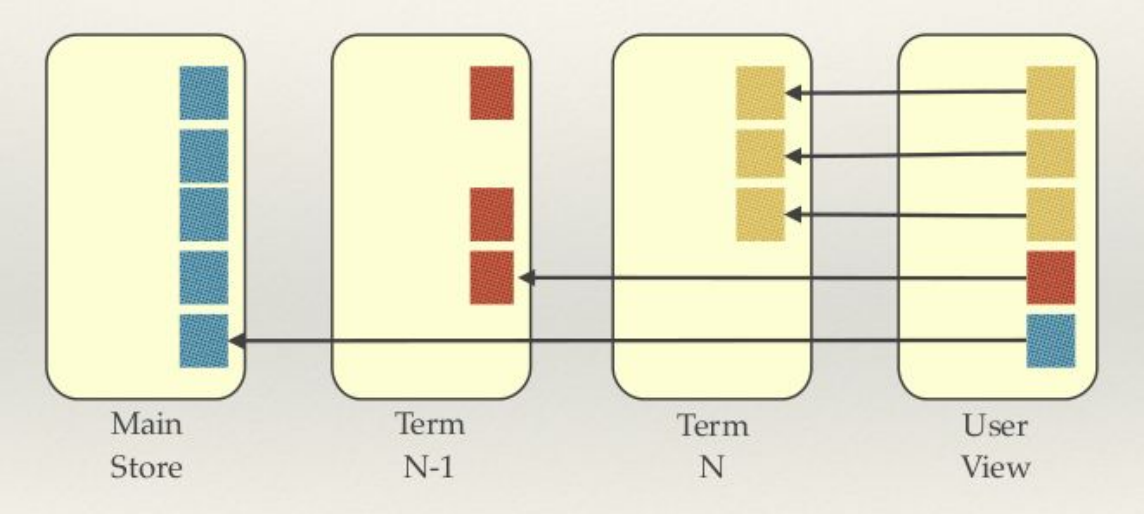
b. **The Index** :We maintain and store a Bloom filter for each term, to enable efficient determination of, whether a term contains anything at all relevant to a subsequent read (or reconciliation). Each data block, directory entry, or metadata field is treated as a separate entity for filter purposes, even if a single request affects multiple entities. For each entity, we calculate a hash and use that to populate the particular term's Bloom filter.



We will also maintain a global filter, to quickly determine, whether any term still contains relevant data for a read or a reconciliation. It will be kept at a coarser granularity, than per-term filters by using only GFIDs, without further refinement by sub-id.

4. In-Memory Journal

Because there might be an indefinite amount of time before writes are destaged from the journal to the main store, we need to account for in-journal data on reads.



Each term here needs to act, as an overlay on anything previous, “hiding” any older content from view. Here only the blocks with the arrows pointing to them, represent data that the user can still see. All other blocks are hidden.

5. NSR Server

The leader plays a crucial role at the core of NSR. Every replica group (not a volume or a cluster), will have one leader at any given point in time. It will perform the following responsibilities:

A. It will be the only member of the group, which would be accepting writes. Non-leader servers of the replica group, on receiving a write request, will let the client know that it’s not the leader.
B. The leader, on receiving the write, forwards it to the followers without performing any operation on it.
C. When any non-leader node receives a request, it writes the fop in the journal. The entry in the journal will be in “Uncommitted” state at this point in time.
D. Once the fop is written in the Journal, the non-leader sends an acknowledgement back to the leader.
E. The leader, on receiving acknowledgements from the followers, decides (based on the current quorum), if quorum will not meet even if the leader’s successful.

F. If Quorum will meet, the leader will try toadd the fop in it's own journal as an "Uncommitted Entry", else it will proceed to step I.
G. After the leader successfully writes the journal entry, it checks if quorum is now met. If the leader fails to perform the write, a leadership change is initiated in the background.
H. If Quorum is met(irrespective of the leader's success or failure), then the leader sends a +ve ack to the client, else if quorum is not met it will proceed to step I.
I. The leader send a -ve ack to the client, and then issues rollback for the same to the followers. If the rollback reaches the followers before they have begun destaging, then the journal entry is marked as "Invalid". If not, then either a roll-back or a roll-forward of the same, will happen during reconciliation, based on the state of the leader and other followers.

5. Reconciliation

Reconciliation in NSR will work hand in hand with Journaling, which enables precise recovery. In the event that a leader has gone down, and there aren't enough servers to form quorum, then all writes will be rejected, until quorum is regained.

Reconciliation, will be driven by the current leader, and will act upon the term information present in etcd, as well as the term states in each replica brick.

a. The reconciliation process will run through all the entries in order, starting from the oldest term, yet to be reconciled.
b. It will check for overlaps, and discard any part that's no longer relevant.
c. By figuring out which replicas are in which state, for any given term, it will propagate from more current to less current.
d. Finally it will mark the entry as completed.

4. Appendix

- http://review.gluster.org/#/c/8915/3/
- http://www.gluster.org/community/documentation/index.php/Features/new-style-replication
- http://blog.gluster.org/2014/04/new-style-replication/
- http://www.projectcalico.org/using-etcd-for-elections/
- https://github.com/jdarcy/etcd-api
- https://github.com/coreos/etcd/blob/master/Documentation/clustering.md