Stratis Software Design: Version 0.4*

Andy Grover <agrover@redhat.com>

Last modified: 10/24/2016

Contents

Introduction

Stratis is a local storage solution that lets multiple logical filesystems share a pool of storage that is allocated from one or more block devices. This is roughly equivalent to what volume-managing filesystems like ZFS and Btrfs do, but differs in implementation. Instead of an entirely in-kernel approach, Stratis uses a hybrid user/kernel approach that builds upon existing block capabilities like devicemapper, existing filesystem capabilities like XFS, and a user space daemon for monitoring and control.

The goal of Stratis is to provide the conceptual simplicity that volume-managing filesystems offer, many of their features, but also surpass them in areas like monitoring and notification, automatic reconfiguration, and integration with higher-level storage management frameworks.

Asking Questions and Making Changes to this Document

This document can be found in the stratis-docs repo, and is written using LyX 2.2.1. Please ask any questions by opening an issue, and propose changes as pull requests.

1 Conceptual Model

Stratis's conceptual model consists of *blockdevs*, *pools*, and *filesystems*. A pool is created from one or more blockdevs (block devices), after which filesystems may be created from the pool. Filesystems are mountable hierarchical collections of files that allocate backing storage from the pool on an as-needed basis. The key difference between a Stratis filesystem and a conventional Unix filesystem is that Stratis filesystem sizes are not managed by the user, but by Stratis.

Since a single system may have multiple pools, each pool has a name, as does each filesystem within a pool. Each pool has a data redundancy level, which may be none, raid1, raid5, raid6, or raid10. The default is none. The pool's

cache also has a redundancy level that also defaults to none. This redundancy level applies to all storage allocated within the pool.

Stratis supports effectively unlimited blockdevs in a pool, and additional blockdevs may be added after the pool is created. Blockdevs may also be removed from a pool, if certain preparations are performed and conditions are met.

Stratis supports up to 2^{20} filesystems. New filesystems may be created from an existing filesystem (i.e. a read/write snapshot), or created from scratch. After creation, Stratis treats all filesystems identically; a filesystem and its snapshot have no dependencies on each other, but simply take up less space in the pool due to each initially referencing the same set of data blocks. However, to aid the user informationally, Stratis tracks and reports snapshot parents.

2 User Experience

Stratis has a command-line tool that enables the administrator to create a Stratis pool from one or more blockdevs, and then allocate filesystems from the pool.

See reference implementation at `https://github.com/stratis-storage/stratis-cli` for the most up-to-date status of the CLI design.

3 D-Bus Programmatic API

The Stratis service process exposes a D-Bus interface, for other programs to integrate support for Stratis. This is considered the primary Stratis interface. The command-line tool uses the D-Bus API.

TODO: Create a separate D-Bus API Reference Doc and link to it here.

3.1 D-Bus Access Control

3.1.1 Security Policy

3.1.2 Prevent Spoofing

4 Scalability and Performance Considerations

Stratis assumes three usage models:

1. Disk only
2. Disk with SSD caching
3. SSD only

The use cases that might do(1) are not terribly concerned with IOPs, but are concerned about resiliency, integrity, and ease of management. This might include large media files that are accessed mostly sequentially, or other“cold” data.

If random IO performance is needed, adding a caching tier based on SSDs(2) is going to be better than any optimization of a disk-only solution. Stratis therefore doesn't optimize case(1) for IOPs, but supports setting up faster devices in a caching tier, for either read caching or read/write caching (for which users likely want redundancy in the cache tier).

In case(3), the cache tier isn't needed; the pool should have high IOPs. Stratis adds value through resiliency, integrity, ease of use, and ease of management.

5 Software Components

Stratis consists of a command-line tool, *stratis*, and a service, *stratisd*.

stratis implements the command-line interface, and converts commands into D-Bus API calls to stratisd.

stratisd implements the D-Bus interface, and manages and monitors Stratis internal pool blockdevs, as described below. It is started by (udev? systemd?) and continues to run as long as Stratis pools or blockdevs are present in the system.

6 Internals

Stratis internals are opaque to the user. This allows its implementation maximum flexibility to do whatever it needs in Stratis version 1, as well as to be extended in later versions without violating any user-visible expectations.

6.1 Cache Tier Requirements

Caching may be configured for redundancy, or no redundancy.

Caching may be configured for write-back and write-through modes.

Stratis concatenates all cache blockdevs and uses the resulting device to cache the thinpool device (L4). This lets all filesystems benefit from the cache.

Cache blocksize should match thinpool datablock size.

The cache tier can be entirely reconfigured without data loss, once flushed and in write-through mode, because it's just a cache. To handle configuration changes, we can just drop the cache and recreate it with its new configuration.

(more details to fill in here.)

6.2 Cache Tier Metadata

6.2.1 Cache Tier Metadata Requirements

1. Identify all blockdevs part of the pool's cache tier, and the configured redundancy level

2. Cache tier supports up to 8 devices.

3. TBD

6.3 Data Tier

6.3.1 Layer 0: blockdevs

This layer is responsible for discovering existing blockdevs in a pool, initializing and labeling new blockdevs unambiguously as part of the pool, setting up any disk-specific parameters (TLER?), and storing pool metadata on each blockdev. Blockdevs may be in the following states: good-unused, good-used, bad, spare.

6.3.2 Layer 1: Integrity (not in version 1)

Identity-mapped immediately above each blockdev in the pool, Stratis uses a to-be-developed DM target that allows the detection of incorrect data as it is read, by using extra space to record the results of checksum/hash functions on the data blocks, and then compare the results with what the blockdev actually returned. This will enable Stratis to detect data corruption when the pool is non-redundant, and to repair the corruption when the pool is redundant. Such a DM target could use DIF information if present, but would also need to work with non-DIF blockdevs.

6.3.3 Layer 2: Redundancy

A Stratis pool may optionally be configured to spread data across multiple physical disks, so that the loss of any one disk does not cause data loss. Stratis uses conventional RAID technology (1, 5, 6, 10) as specified, and converts Layer 0 blockdevs into a smaller-sized amount of storage with the specified raid properties.

Since Stratis supports more blockdevs than are RAID-able (generally 8 or fewer is best for a raidset IIRC), and differently-sized blockdevs, a redundant Stratis pool will likely contain multiple raid sets (all of the same global type) Depending on layout, there may be some amount of space in a pool's blockdevs that cannot be used becauseit cannot be used in a RAID set. Stratis will intensively manage raidsets,extending them across newly added blockdevs (version 1) or creating new raidsets; and handling the removal of blockdevs (version X). Stratis may use dm-raid's reshape capabilities when possible.

A pool may also be configured with redundancy level of 'none'. In this case, Stratis may opt to create a striped blockdev ('raid0') across all or some part of the blockdevs, if there are more than one.

Stratis cannot support redundancy with a single disk, but we may wish to reserve the small space for raid metadata and other uses even on one-disk Stratis pools. This will allow the pool to be made redundant (in the version when we support this) without encountering some ugly edge cases.

6.3.4 Layer 3: Flexibility

Whether blockdevs are part of raidsets or used directly, pools need to cope with the addition (version X) or removal (version X+1) of them.

Version X: Stratis needs to add a blockdev to a pool, and have the space available to its filesystems grow accordingly. This may involve reshaping one or more raidsets to be wider, or just knowing to extend the thinpool onto the blockdev when previous blockdevs are full.

Version X+1: Stratis needs to be able assign states to each blockdev that reflect whether a) it can be removed from the pool with no effect, b) removed with loss of redundancy, or c) not removed without data loss. It should also be able to move active data off a perhaps-failing disk (or refuse, if space on other disks in the pool is not sufficient) so that the disk then can be removed from the pool. blockdevs may also be marked as 'spare', so that they do not contribute to the total available space of the pool, but can immediately replace a failed disk.

This layer will support two linear devices made up of segments from L2 devices. It will track what L2 devices they are allocated from, allow the two devices to grow (and shrink?), and handle the online movement of segments in these devices when L2 devices come and go(v2).

6.3.5 Layer 4: Thin Provisioning

The two linear targets from L3 are used as metadata device and data device for a thin pool. Stratis manages the thinpool device by extending the two L3 subdevices when either runs low on available blocks. If we approach a point where we no longer have empty L2 space to extend onto, this is when we would need to start setting off alarm bells, remounting read-only, throttling writes, that sort of thing.

6.3.6 Layer 5: Filesystem

From the thin pool Stratis can create one or more thin volumes. It will automatically give a new volume a default size, and format it with a filesystem, and make it available to the user. Stratis monitors each filesystem's fullness vs its capacity and automatically extends (using e.g. xfs_growfs) them without user intervention. Stratis also periodically runs fstrim on each one as well.

Stratis keeps snapshot origin info in its metadata for user informational purposes.

Encryption layer (in version 3) could conceivably go on top of thin device, making it per-fs.

6.4 Data Tier Metadata

Stratis must track the blockdevs that make up the data tier of the pool (L0), integrity parameters (L1), the raidsets that are created from the data blockdevs (L2), the two linear targets that span the L2 devices (L3), the thinpool device

(L4) and the attributes of the thin devices and filesystems created from the thinpool (L5).

6.4.1 Requirements

1. Uniquely identify a blockdev as used by Stratis, which pool it is a member of, and parameters needed to recreate upper layers 1-5
2. Detect incomplete or corrupted metadata
3. Allow for blockdevs being expanded underneath Stratis
4. Redundant on each blockdev to tolerate e.g. accidental dd
5. Redundant across blockdevs to handle missing, damaged etc. members. Can provide number & characteristics of missing blockdevs
6. Handle thousand+ blockdevs in a pool
7. Handle million+ filesystems in a pool and updates without writing to each blockdev (idea: use a system filesystem to store user filesystem info?)
8. Extensible/upgradeable metadata format

6.4.2 Design Overview

Stratis metadata is in three places:

1. Blockdev Data Area
   (a) Signature Block
   (b) Metadata Area (MDA)
2. Thin Device 0

6.4.3 BlockDev Data Area (BDA)

The BDA consists of a fixed-length header of eight sectors, which contains the Signature Block, followed by the metadata area (MDA), whose length is specified in the Signature Block.

Stratis reserves the first 8 sectors of each blockdev for a static header:

sector offset	length (sectors)	contents
0	1	unused
1	1	Signature Block
2	6	unused

The Signature Block consists of:

byte offset	length (bytes)	description
0	4	CRC32 of signature block (bytes at offset 4 length 508)
4	16	Stratis signature: '!Stra0tis\x86\xff\x02^\x41rh'
20	8	Device size in 512-byte sectors (u64)
28	4	flags
32	32	ASCII hex UUID for the Stratis pool
64	32	ASCII hex UUID for the blockdev
96	8	MDAA UNIX timestamp (seconds since Jan 1 1970) (u64)
104	4	MDAA nanoseconds (u32)
108	4	MDAA used length in bytes (u32)
112	4	MDAA CRC32 over used length
116	12	unused
128	8	MDAB UNIX timestamp (seconds since Jan 1 1970) (u64)
136	4	MDAB nanoseconds (u32)
140	4	MDAB used length in bytes (u32)
144	4	MDAB CRC32 over used length
148	12	unused
160	4	sector length of blockdev metadata area (MDA) (u32)
164	4	sector length of reserved space (u32)
168	344	unused

- All 'flags' or 'unused' fields are zeroed.
- Blockdev metadata area (offset 160) must be an even number of at least 2040 – combined minimum length of overall BDA (static header and MDA) is 2048 sectors (1 MiB).
- The BDA is written to the beginning and end of the blockdev.
- The BDA at the start of the blockdev is followed immediately by *reserved space*, whose size is specified in the signature block (offset 164).
- The MDA is evenly divided into two areas: MDAA and MDAB.
- Data within the metadata areas is stored in JSON format.
- Metadata updates write to the older of the MDAA and MDAB areas. This is determined by lowest timestamp, and then lowest nanoseconds if timestamps are equal.
- The procedure for updating metadata is:
  1. The JSON metadata is written to either MDAA or MDAB in the start-of-blockdev BDA, as determined above.
  2. The chosen MDA's timestamp, nanosecond, length, and CRC fields in the signature block are updated: timestamp gets the UNIX timestamp. Nanoseconds is the current nanoseconds within the second. Length is set to the length of the metadata, and the CRC32 is calculated on the data up to that length.

3. The signature block's CRC32 is calculated.
4. The signature block is written and flushed to disk. The implementation may choose to write the entire 8 sectors of the static header.

- The identical MDAx and signature block is then written to the end-of-blockdev BDA, followed by a flush/FUA.

6.4.4 Metadata Area (MDA)

The MDA contains a JSON structure that represents the pool's overall configuration of blockdevs, from L0 to L4.

6.4.5 Thin Device 0

Thin device 0 is an XFS-formatted thin filesystem that is used by Stratis to store information on user-created filesystems (L5). These are stored in the filesystem in TBD format.

6.4.6 The MDA and Very Large Pools

Stratis pools with very large numbers of blockdevs will encounter two issues. First, updating the metadata on all blockdevs in the pool may become a performance bottleneck. Second, the default MDA size may become inadequate to contain the information required.

To solve the first issue, Stratis caps the number of blockdevs that receive updated metadata information. A reasonable value for this cap might be in the range of 6 to 10, and should try to spread metadata updates across path-independent blockdevs, if this can be discerned, or randomly. This limits excessive I/O when blockdevs are added or removed from the pool, but maximizes the likelihood that up-to-date pool metadata is retrievable in case of failure.

To solve the second issue, Stratis monitors how large its most recent serialized metadata updates are, and increases the size of MDA areas on newly added devices when a fairly low threshold – %50 – is reached in comparison to the available MDAx size. This ensures that by the time sufficient blockdevs have been added to the pool to be in danger of serialized JSON data being too large, there are enough blockdevs with enlarged MDA space so that they can be used for MDA writes.

7 Delivery of Features

7.1 Stratis version 0.1

Simplest thing that does something useful

1. Create a pool
2. Destroy a pool

3. Create a filesystem
4. Create a filesystem from existing filesystem (a r/w snapshot)
5. Destroy a filesystem
6. List filesystems
7. Rename filesystems
8. List pools
9. Rename pools
10. List blockdevs in a pool
11. Redundancy level: none
12. D-Bus API
13. Command-line tool
14. Save/restore configuration across reboot
15. Initial disk labeling and on-disk metadata format
16. Error handling for missing, corrupted, or duplicate blockdevs in a pool

7.2 Stratis version 0.5

Add cache tier

1. List cache blockdevs in a pool
2. Add cache blockdevs
3. Remove cache blockdevs
4. Write-through caching only

7.3 Stratis version 1.0

Minimum Viable Product

1. Snapshot management: auto snaps, date-based culling, "promotion" from snap to "primary"
2. Monitor pool(s) for getting close to capacity, and do something (remount ro?) if dangerously full
3. Some sort of notification method to the user if pool is approaching capacity
4. Maintain filesystems: Grow a filesystem as it nears capacity
5. Maintain filesystems: Run fstrim periodically to release unused areas back to thinpool
6. Add and use an additional blockdev

7.4 Stratis version 2.0

Add Redundancy Support

1. Remove an existing blockdev
2. Redundancy level: raid1
3. Redundancy level: raid5
4. Redundancy level: raid6
5. Redundancy level: raid10
6. Cache redundancy level: raid1
7. Write-through caching enabled
8. thin/cache metadata validation/check (i.e. call thin_check & cache_check)
9. Quotas
10. Blockdev resize (larger)
11. Spares

7.5 Stratis version 3.0

Rough ZFS feature parity. New DM features needed.

1. Send/Receive Integrity checking (w/ self-healing only if on raid)
2. Raid scrub
3. Compression
4. Encryption
5. Dedupe[aq]
6. Raid write log (on ssd? To eliminate raid write hole)

7.6 Stratis version 4.0

Future features and evolution

1. Change a pool's redundancy level
2. Boot from a filesystem
3. Libstoragemgmt integration
4. Multipath integration
5. Tag-based blockdev and filesystem classification/grouping
6. Mirroring across partitions within a pool, for multi-site or across hw failure domains (shelves/racks)

8 Open Questions

Initial filesystem sizing. Mkfs does different things depending on the size of the blockdev. If it's small then things will be suboptimal if we grow it substantially. Weigh this against too large, which would waste thinpool space (mkfs touches/allocates more thin blocks).

Alignment and tuning of sizes across layers. It would be great if the fs happens to write to a new location that allocates a thin block that it uses that entire block. Also, look at XFS allocation groups, they may work cross-purposes to thinpool by spreading files across the blkdev. Align as much as possible.

Behavior when thin pool is exhausted. Slow down? Switch all fs to read-only?

Should we separate cache into read devs and write devs? Write-back cache we may want to support redundancy (cache contains only copy of data until flush) whereas read cache redundancy serves little purpose, just reduces the total space available.

Snapshot semantics & management. Snapshots should be thought of as "secondary" and read-only until promoted to "primary" and read-write[1].

Quotas. If the default behavior is to grow a vol that nears its notional capacity, how do we limit it for vols (e.g. /var/log or time machine) where being limited in size is part of how they expect to operate?

Should we have a 'monitor' subcommand? What would it do?

Being able to identify and differentiate blockdevs within a pool is very important, given how many there can be. We should provide the user with system-generated info, and also allow the user to add their own descriptions of blockdevs.

Journald interactions

Nfs interactions

[1] See GitHub issue:`https://github.com/stratis-storage/stratisd/issues/19`