Stratis Software Design: Version 2.0.0*

Last modified: 01/10/2020

Contents

Asking Questions and Making Changes to this Document

This document can be found in the stratis-docs repo, and is written using LyX 2.3.3. Please use the mailing list (stratis-devel@lists.fedorahosted.org), or open an issue on GitHub, for any questions or issues that arise.

Executive Summary

Stratis is a new tool that meets the needs of Red Hat Enterprise Linux (RHEL) users (among others) calling for an easily configured, tightly integrated solution for storage that works within the existing Linux storage management stack. To achieve this, Stratis prioritizes a straightforward command-line experience, a rich API, and a fully automated, externally-opaque approach to storage management. It builds upon elements of the existing storage stack as much as possible, to enable delivery within 1-2 years. Specifically, Stratis initially plans to use device-mapper and the XFS filesystem, but may incorporate other technology in the future.

Part I
Background

1 Problem Statement

Linux has gained many storage-related features over the years, but each of these features has required the user to manage the configuration of these features in a layered, additive manner. Genuinely new and useful features such as thin provisioning, RAID, and multipath are dependent on the user correctly configuring many different layers via different tools to achieve a complete result. Furthermore, since each layer's configuration tool only has a command-line interface (CLI), higher-level management tools must each construct input and parse the human-oriented output for each these layers' CLI. This causes a waste of effort and opportunity for bugs, as each higher-level tool builds its own internal API for the feature on top of the lower level tool's CLI.

1.1 Goal: Bring advanced features to users in a simpler form

Linux storage features are modular and stackable. This promotes flexibility and allows independent development efforts, but leads to a huge number of possible configurations. This requires the user to manage the stack because there's not enough commonality to enable effective automation.

But really, there *is* a single configuration that can work for most use cases. By assuming a fixed layering of storage features (some perhaps optional), we enable software to effectively manage these on behalf of the user.

Automated management then leads to less administrative burden placed on the user. The user still specifies resources, desired features, and results – what hardware resources to use, what features to enable, how storage should be logically presented – using a smaller number of concepts with well-defined relations. Software manages the rest, and handles most runtime issues without user involvement.

1.2 Proposal: Implement a hybrid Volume Managing Filesystem

In the past ten years, *volume-managing filesystems* (VMFs) such as ZFS and Btrfs have come into vogue and gained users, after being previously available only on other UNIX-based operating systems. These incorporate what would be handled by multiple tools under traditional Linux into a single tool. Redundancy, thin provisioning, volume management, and filesystems become features within a single comprehensive, consistent configuration system. Where a traditional Linux storage stack exposes the layers of block devices to the user to manage, VMFs hide everything in a *pool*. The user puts raw storage in the pool, the VMF manages the storage in the pool, providing the features the user wants, and allows the user to create filesystems from the pool without being concerned with the details.

Unfortunately, existing VMFs aren't easily used on enterprise Linux distributions like RHEL. ZFS isn't an option RHEL can embrace due to licensing, Ubuntu notwithstanding. Btrfs has no licensing issues, but maintaining up-to-date support for it in enterprise kernels proved difficult.

We can see from the many developer-years of effort that have gone into these two projects that writing a VMF is a tremendous, time-consuming undertaking. We also can hear our users demanding their features and ease of use.

Rather than writing a new VMF from scratch, Stratis proposes to satisfy VMF-like requirements by managing existing technologies on behalf of the user, so that users can manage their storage using high-level concepts like "pool" and "filesystem", and remain unconcerned with the more complex details under the covers.

This is also a chance to learn from the benefits and shortcomings of existing solutions. We should not just copy ZFS. ZFS is now fifteen years old and the storage landscape has changed since its design. We seek to satisfy the same needs that ZFS does, but also integrate more tightly into today's increasingly automated storage management solutions that span the data center as well as the local machine. This is made possible by a hybrid, userspace-based approach.

1.3 Requirements

1. *Make features easier to use in combination with each other*: thin provisioning, snapshots, multipath, encryption, hardware reconfiguration, monitoring, and a caching tier
2. Simple and comprehensive command-line interface
    (a) Simple
        i. Single way to do things
        ii. Do not expose internal implementation details. Gives Stratis more implementation freedom, and of little value since internals are too complex to make manual user repairs practical
        iii. User typically will not use on a daily basis
            A. Consistent commands that a user can guess at, and probably be right
            B. Require explicitness from the user for potentially data-losing operations, such as giving a "–force" option.
    (b) Comprehensive
        i. User must master only one tool
        ii. Helps user learn: if task not possible through tool, it must not be worth doing (or a good idea)
3. Programmatic language-neutral API for higher-level management tool integration
    (a) A clear next step for users after hitting the limitations of scripting the CLI
    (b) Encourages tight integration and use of all features by higher-level tools
4. Event-driven monitoring and alerts

   (a) Monitoring and alert messages expressed in terms of Stratis user-visible simple concepts, not implementation details
   (b) Low CPU/memory overhead to monitoring
   (c) Only alert when action really is needed
   (d) Fail gracefully if alerts are unheeded

5. Eliminate manual resizing of filesystems
   (a) Numerous problem reports throughout the years indicate that resizing filesystems is an area where users feel unease, due to potential data loss if a mistake is made. No real reason to require the user do this any more.
   (b) Simpler for DevOps
   (c) Makes storage "demand-allocated", similar to virtual memory. Current technology allows us to manage a filesystem's actual usage up (growfs) or down (thin provisioning).

6. Initrd-capable
   (a) Allows root fs, all other filesystems except /boot to use Stratis. Needed for ease of use
   (b) Limited environment – alternate IPC mechanism that works in the initrd is available

7. Adaptable to emerging storage technologies
   (a) Persistent memory
      i. Block-appearing pmem can be used by Stratis

8. Implementable in 1-2 years
   (a) We're already behind, waiting another 10 years isn't an option

Part II
Solution Overview

2 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. Instead of an entirely in-kernel approach like ZFS or Btrfs, Stratis uses a hybrid user/kernel approach that builds upon existing block capabilities like device-mapper, existing filesystem capabilities like XFS, and a user space daemon for monitoring and control.

The goal of Stratis is to provide the conceptual simplicity of volume-managing filesystems, and surpass them in areas such as monitoring and notification, automatic reconfiguration, and integration with higher-level storage management frameworks.

3 Stratis and the Linux storage stack

Stratis simplifies many aspects of local storage provisioning and configuration. This, along with its API, would let projects dependent on configuring local storage do so much more easily.

For example, installing the OS to a Stratis pool using Anaconda. After selecting the disks to use for the pool, the first benefit would be the complex flow around sizing of filesystems could

be omitted. Second, since Stratis has an API, Anaconda could use it, instead of needing work in Blivet to build an API on top of command line tools.

Other management tools like Cockpit, virtualization products like RHEV, or container products like Atomic would find it much simpler and less error-prone to use storage and snapshots with Stratis, for the same two reasons: don't need to worry about per-filesystem sizing (only that the pool has enough "backing store"); and the API, which allows better tool-to-tool integration than using CLI programmatically.

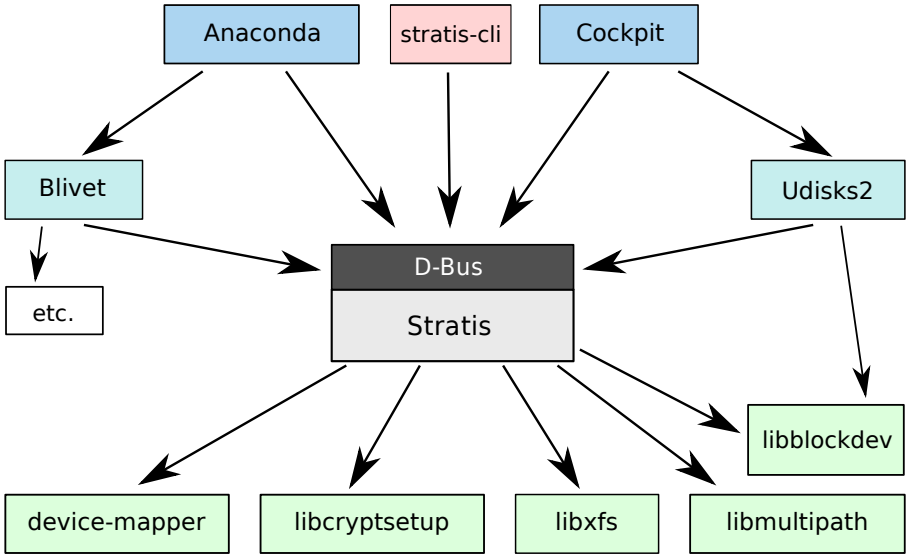

Figure 1: Future Stratis Position in the Linux Storage Management Stack

4 Conceptual Model

4.1 Blockdevs, pools, and filesystems

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

4.2 Attributes and features of a pool

A pool is created with an initial set of one or more blockdevs. Blockdevs may also be added after the pool is created. The pool's primary collection of blockdevs is called the *data tier*.

A pool also optionally has a *cache tier* that uses a separate collection of faster blockdevs to improve performance instead of increasing the pool's capacity.

Since a single system may have multiple pools, each pool has a name, as does each filesystem within a pool. These are both settable by the user. Blockdevs, pools, and filesystems also have UUIDs, which are not settable by the user.

Stratis supports large numbers of blockdevs and up to 2^{24}filesystems per pool. However, practical limits on these values may compel users to restrict themselves to smaller numbers of blockdevs and filesystems.

A new filesystem is either a new empty filesystem or a snapshot of an existing filesystem within the pool. Stratis currently does not distinguish between snapshots and filesystems.

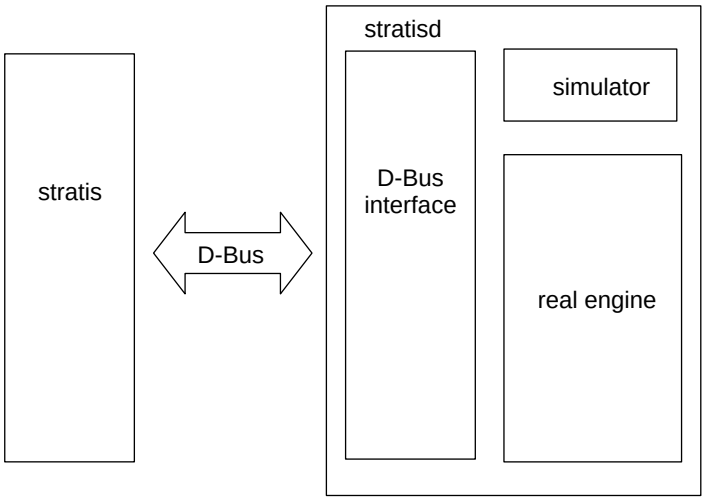

Figure 2: Stratis Architecture

5 Scalability and Performance Considerations

Stratis doesn't optimize performance within its data tier, instead focusing there on flexibility and integrity. Improved performance is the job of caching tier, or perhaps building the pool using blockdevs with higher IOPs, such as SSDs.

Part III
Implementation

6 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 the system and continues to run as long as Stratis pools or blockdevs are present in the system.

stratisd includes a simulator engine. The simulator engine is purely computational and does not affect the environment, although it does communicate over the D-Bus.

Figure 2 shows the basic Stratis architecture.

7 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.

This component is not required to be installed, in cases such as an appliance where a higher-level application such as Cockpit or Ansible uses the D-Bus API directly.

7.1 Known shortcomings

Stratis' goal is to hide the complexity of its implementation from the user, but by using a reuse/layering approach to its implementation, there will be places where Stratis' implementation details will peek through. This could cause user confusion, and also could threaten Stratis integrity if the user makes changes.

- For Stratis filesystems, 'df' will report the current used and free sizes as seen and reported by XFS. This is not useful information, because the filesystem's actual storage usage will be less due to thin provisioning, and also because Stratis will automatically grow the filesystem if it nears XFS's currently sized capacity.
- Users should not try to reformat or reconfigure XFS filesystems that are managed by Stratis. Stratis has no way to enforce this or warn the user to avoid this, other than in the documentation.
- Stratis will use many device-mapper devices, which will show up in 'dmsetup' listings and /proc/partitions. Similarly, 'lsblk' output on a Stratis system will reflect Stratis' internal workings and layers.
- Stratis requires a userspace daemon, which must remain running at all times for proper monitoring and pool maintenance.

8 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.

8.1 Overview

The D-Bus API is part of stratisd. It is a thin layer that receives messages on the D-Bus, processes them, transmits them to the Stratis engine, receives the results from the engine, and returns the result to the invoker of the API. When processing method calls, its responsibilities are confined to:

- Receiving arguments and verifying that they conform to the signature of the invoked method.
- Transforming method arguments received on the D-Bus to arguments of the appropriate type to be passed to engine methods.
- Converting tuple arguments used to represent non-mandatory arguments to values which inhabit the Rust Option type.
- Invoking the appropriate engine methods and capturing their return values.
- Marshalling the appropriate return values to place on the D-Bus along with the return code and message.
- Adding or removing objects from the D-Bus tree.

The D-Bus API is implemented using the dbus-rs library[1].

The Stratisd D-Bus API Reference Manual contains a description of the API.

[1] `https://github.com/diwic/dbus-rs`

8.2 D-Bus Access Control

8.2.1 Security Policy

Most stratisd D-Bus methods require root permissions. However, listing operations do not; these can be done by an unprivileged user. The default permissions are specified in the policy file, stratisd.conf, included in the stratisd distribution. These defaults permit all actions by root users but restrict unprivileged users to read-only actions. Systems administrators can adjust permissions by editing the stratisd D-Bus policy files.

8.3 Querying stratisd state via D-Bus

stratisd exposes a reporting interface that allows users to query the state of stratisd's internal data structures for debugging and error reporting. The D-Bus query returns a JSON string that can be parsed to detect state programmatically; however, the report interface is unstable, and consequently the names and schemas of provided reports do not follow the semantic versioning rules to which the rest of stratisd's API conforms.

9 JSON RPC IPC mechanism

9.1 Overview

Due to restrictions in the initramfs where D-Bus is not currently present, there is an alternate mechanism for IPC in the initramfs. This IPC mechanism implements a JSON RPC framework passed over Unix sockets. The motivation for using Unix sockets is that:

1. They are available in the initramfs as a form of IPC
2. They allow passing file descriptors from process to process in the same network packet as the JSON

The JSON RPC IPC mechanism uses the exact same code for the stratisd storage engine and is simply a thin layer that handles all of the network operations and input parsing to provide arguments to the engine. The initramfs IPC API is more limited than that of the D-Bus API and is not versioned for backwards compatibility given that it is expected that the corresponding CLI will be used to communicate with the minimal daemon.

10 Internals

Stratis internals aim to be 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.

10.1 Data Tier Requirements

The data tier of Stratis must manage blockdevs on behalf of the user to provide the following:

1. Managed filesystems that consume only as much space as the files they contain
2. Fast snapshots of existing filesystems
3. The ability to add individual blockdevs to grow the physical space available to filesystems
4. Encryption

10.2 Data Tier

The data tier achieves these requirements by layering device-mapper (DM) devices on top of the pool's blockdevs. The topmost layer consists of thin devices allocated from a thinpool. Stratis initializes these thin devices with a filesystem, and manages the DM devices and filesystems to meet the above requirements.

10.2.1 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, and storing pool metadata on each blockdev. The minimum blockdev size Stratis will accept is 1 GiB. Blockdevs may be encrypted or unencrypted. See 10.5 for details on the implementation of encryption.

10.2.2 Flex

Pools need to cope with the addition of block devices.

Stratis allows adding a blockdev to an existing pool, and using it to grow the pool's allocated space.

The flexibility layer contains four linear DM devices made up of segments from lower-level devices. The first two devices will be used by Layer 4 (Thin Provisioning) as metadata and data devices. The flex layer will track what lower-level devices these are allocated from, and allow the two devices to grow as needed.

The third linear DM device is a spare metadata device to be used in the case that the metadata device requires offline repair. It will not usually be instantiated on the system, but guarantees there is room if needed. This device's size tracks the size of the metadata device, both as initially allocated, and as the metadata device is extended.

The fourth and final linear DM device is used for the Metadata Volume (MDV, see 10.3.6). The MDV is used to store metadata about upper layers, layer five and above.

The initial sizes of all flex layer devices should be chosen to allow an entire pool to fit within a single blockdev of the minimum size (1 GiB).

10.2.3 Thin Provisioning

The two linear targets from L3 are used as a metadata device and data device for a DM thinpool device. The thinpool device implements a copy-on-write (CoW) algorithm, so that blocks in the data device are only allocated as needed to back the thin volumes created from the thinpool.

Stratis manages the thinpool device by extending the two subdevices in the thinpool when either runs low on available blocks. If the pool approaches a point where it no longer has empty lower-level space to extend onto, Stratis alerts the user and takes action to avoid data corruption.

10.2.4 Thin Volumes

Stratis creates thin volumes from the thin pool. It will automatically give a new volume a default size, format it with a filesystem, and make it available to the user.

Stratis also enables creating a new volume as a read/write snapshot of an existing volume. Although the underlying implementation does not require maintaining the relation between a snapshot and its origin, Stratis records this relation in its metadata. This relation may be of use to users who may, for example, use snapshots for backups and may make use of the origin information to identify a particular backup snapshot to restore from.

10.2.5 Filesystem

Stratis monitors each filesystem's usage against its capacity and automatically extends them online without user intervention. Extending involves changing the thin dev's logical size, and then using a tool such as xfs_growfs to grow the filesystem.

10.3 Data Tier Metadata

Stratis must track the blockdevs that make up the data tier of the pool, the three linear targets that span the block devices, the thinpool device and the attributes of the thin devices and filesystems created from the thinpool.

10.3.1 Requirements

1. Uniquely identify a blockdev as used by Stratis, which pool it is a member of, and parameters needed to recreate all layers
2. Detect incomplete or corrupted metadata and recover via second copy
3. Allow for blockdevs being expanded underneath Stratis
4. Redundant on each blockdev to tolerate unreadable sectors[2]
5. Redundant across blockdevs to handle missing or damaged members. Can provide metadata of missing blockdevs
6. Handle thousand+ blockdevs in a pool
7. Handle million+ filesystems in a pool and updates without writing to each blockdev
8. Extensible/upgradable metadata format

10.3.2 Conventions

Sectors are 512 bytes in length[3].

UUIDs are written as un-hyphenated ASCII encodings of their lower-case hexadecimal representation[4] except in JSON-formatted metadata where they are unhyphenated.

All checksums are calculated using an implementation of the CRC-32C (Castagnoli) algorithm.

10.3.3 Design Overview

Stratis metadata is in three places:

1. Blockdev Data Area (BDA)
   (a) Signature Block within Static Header
   (b) Metadata Area (MDA)
2. Metadata Volume (MDV)

[2]Recovery from accidental start-of-blockdev overwrite by placing a second copy at the end of the disk was also considered, but raised other issues that outweighed its benefit.

[3]Historically this is the minimum storage unit of a hard drive. Many Linux kernel APIs assume this value is constant (as does this document), and use another term such as 'block size' for dealing with cases where the minimum storage unit is different.

[4]UUIDs are 128-bit values and therefore require only 16 bytes to represent their numeric value. However, since each ASCII value requires a byte, and the hexadecimal representation of an 128-bit value requires 32 hexadecimal digits, the chosen encoding requires 32 bytes.

(Specific DM targets such as the thinpool also place their own metadata on disk.)

Information is duplicated across all blockdevs within an on-disk metadata format called the Blockdev Data Area (BDA). The BDA consists of a binary Signature Block, and the Metadata Area (MDA), which stores information in a text-based JSON format. Both the binary and text-based portions of the BDA define redundancy and integrity-checking measures.

The Metadata Volume (MDV) stores metadata on Layers 5 and up in a conventional block device and filesystem that is part of the Flex layer. Choosing to split overall metadata storage into two schemes allows upper layers' metadata to be free of limitations that would apply if a single scheme was used. For example, on-disk metadata formats find it hard to support runtime size extension, may keep redundant copies to ensure reliability, and aggressively check for corruption. This can work well with small amounts of data that is infrequently changed, but has trouble as data grows, or we wish to do updates in-place.

Upper-level metadata can achieve redundancy and integrity by building on the pre-existing lower layers, and work under looser restrictions around updating in place, and the total size to which it may grow. It can reuse an existing, well-tested solution for solving data organization and storage issues – a general-purpose filesystem.

10.3.4 BlockDev Data Area (BDA)

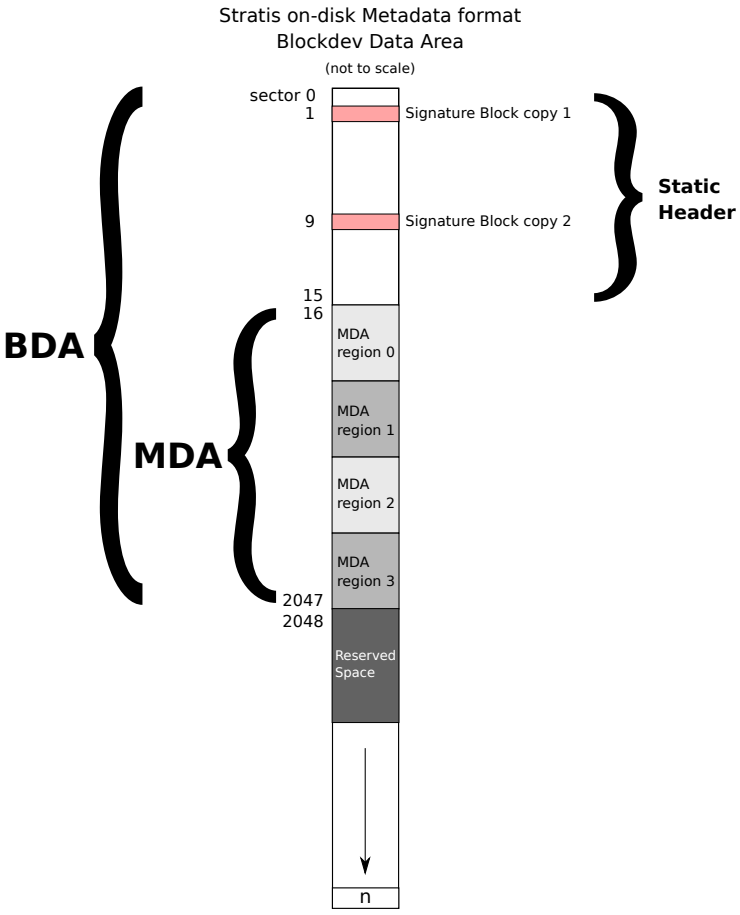

Figure 3: BDA format

The BDA consists of a fixed-length Static Header of sixteen sectors, which contains two copies of the Signature Block; and the metadata area (MDA), whose length is specified in the Signature Block. These are written to the beginning of the blockdev as described below.

Stratis reserves the first 16 sectors of each blockdev for the Static Header. When initializing or modifying the Signature Block, identical data is written to locations 1 and 2.

Static Header

sector offset	length (sectors)	contents
0	1	unused
1	1	Signature Block location 1
2	7	unused
9	1	Signature Block location 2
10	6	unused

Signature Block

byte offset	length (bytes)	description
0	4	checksum 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	1	Signature Block version (u8) (value = 1)
29	3	unused
32	32	UUID of the Stratis pool
64	32	UUID of the blockdev
96	8	sector length of blockdev metadata area (MDA) (u64)
104	8	sector length of reserved space (u64)
112	8	flags (u64)
120	8	initialization time: UNIX timestamp (seconds since Jan 1 1970) using UTC (u64)
128	384	unused

- No flags are yet defined, so 'flags' field is zeroed.
- All 'unused' fields are zeroed, and are reserved for future use.
- If not zero, blockdev metadata area length (offset 96) must be a number divisible by four of at least 2032.
- The BDA is followed immediately by *reserved space*, whose size is specified in the signature block (offset 104).
- Minimum length of BDA (static header and MDA) plus Reserved Space is 2048 sectors (1 MiB).
- When a blockdev is removed from a pool, or is part of a pool that is destroyed, Stratis wipes the Static Header.
- The purpose of the unused sectors is twofold. First, placing the Signature Block copy locations in two separate 4K blocks helps to prevent a single bad write operation on 4K-block disks from corrupting both copies. Second, using a single sector for the Signature Block helps to minimize the likelihood of corruption on disks with 512 byte blocks.
- Each time that Stratis writes one or both Signature Block locations, it also zeroes the unused sectors that share the same 4K block.

The MDA is divided into four equal-size regions, numbered 0-3. When updating metadata, identical data is written to either the odd (1 and 3) or even (0 and 2) regions, chosen by examining the timestamps and overwriting the older of two pairs.

Each MDA region's update consists of a fixed-length MDA Region Header, followed by variable-length JSON data.

MDA Region Header

byte offset	length (bytes)	description
0	4	checksum covering remainder of MDA header
4	4	checksum covering JSON data
8	8	length of JSON data in bytes (u64)
16	8	UNIX timestamp (seconds since Jan 1 1970) using UTC (u64)
24	4	nanoseconds (u32)
28	1	Region Header version (u8) (value = 1)
29	1	Variable-length metadata version (u8) (value = 1)
30	2	unused
32	variable	JSON data

- Metadata updates write to the older of the odd or even MDA regions. This is determined by lowest timestamp, and then lowest nanoseconds if timestamps are equal.
- MDA updates include the MDA Header, which includes the current time. However, if using the current time would not result in the update having the latest time across all MDA regions on all blockdevs in the pool, instead use a time of one nanosecond later than the latest MDA region time across all blockdevs.
- The procedure for updating metadata is:
  1. Determine which regions in the MDA to use (odd or even) as described above.
  2. Write MDA header and JSON data to the first MDA region (0 or 1)
  3. Perform a Flush/FUA
  4. Write MDA header and JSON data to the second MDA region (2 or 3)
  5. Perform a Flush/FUA
  6. Repeat for additional blockdevs. Also see 10.3.7
- Multiple blockdevs being updated with the same metadata must write identical data to each MDA region, but which regions (odd or even) is used may vary, if the blockdevs have received differing numbers of metadata updates over time.

10.3.5 Metadata Area (MDA)

The MDA contains a JSON object that represents the pool's overall configuration from L0 to L4.

Top level objects:

key	JSON type	required	description
name	string	y	the name of the pool
backstore	object	y	the block devices in the pool
flex_devs	object	y	layout of the data and metadata linear devices
thinpool_dev	object	y	parameters of the thinpool device
started	boolean	as of stratisd 3.1.0	indicates whether a pool is started or stopped

backstore: An object describing the data tier and the cache tier.

key	JSON type	required	description
data_tier	object	y	the block devices in the data tier
cap	object	y	the cap device, from which segments are allocated to the flex layer
cache_tier	object	n	the block devices in the cache tier

data_tier: An object describing the data tier.

key	JSON type	required	description
blockdev	object	y	Settings and mappings describing block devices that make up the tier

blockdev: An object describing physical block devices that make up the tier.

key	JSON type	required	description
devs	array	y	an array of **base_block_dev** objects
allocs	array	y	an array of arrays of **base_dev** objects

base_dev: An object describing an allocation from a block device.

key	JSON type	required	description
parent	string	y	UUID of the device the segment is created from
start	integer	y	the starting sector offset within the parent device
length	integer	y	the length in sectors of the segment

base_block_dev: An object describing a block device in the lowest layer.

key	JSON type	required	description
uuid	string	y	The UUID of the block device, as recorded in its Signature Block
user_info	string	n	user-provided information for tracking the device
hardware_info	string	n	uniquely identifying information for the blockdev, such as SCSI VPD83 or serial number

cap: An object describing a view of the top-level linear device provided by the backstore to the flex layer.

key	JSON type	required	description
allocs	array	y	an array of pairs of integers representing the starting offset and length of an allocation in sectors

cache_tier: An object describing the cache tier. Identical format to **data_tier** except VDO layer is not supported.

flex_devs: An object with four keys that define the linear segments that make up each device in the Flex layer.

key	JSON type	required	description
meta_dev	array	y	an array of pairs of integers representing the starting offset and length of an allocation in sectors that make up the metadata volume (MDV)
thin_meta_dev	array	y	an array of pairs of integers representing the starting offset and length of an allocation in sectors that make up the thin metadata device
thin_meta_dev_spare	array	y	an array of pairs of integers representing the starting offset and length of an allocation in sectors that make up the thin metadata spare device
thin_data_dev	array	y	an array of pairs of integers representing the starting offset and length of an allocation in sectors that make up the thin data device

thinpool_dev: An object that defines properties of the thinpool device.

key	JSON type	required	description
data_block_size	integer	y	the size in sectors of the thinpool data block
feature_args	array	since stratisd 3.1	the feature args passed to the thin pool on setup
fs_limit	integer	since stratisd 3.1	the maximum number of filesystems able to be created in this pool
enable_overprov	boolean	since stratisd 3.1	indicates whether overprovisioning is allowed on this pool

10.3.6 Metadata Volume (MDV)

The Metadata Volume is formatted with an XFS filesystem that is used by Stratis to store information on user-created thin filesystems (L5-L7). This information is stored in the filesystem in a TBD format, maybe either an individual file-based scheme, or SQLite database.

10.3.7 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 MDA region 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 that they can be used for MDA writes.

10.3.8 Metadata and Recovery

Bad things happen.

In order to recover from disk errors, Stratis uses checksums over the critical metadata, and writes duplicate copies to a single blockdev, as well as across multiple blockdevs, when possible. It takes this approach – copies – rather than a mechanism that might make it possible to partially repair corrupted metadata for three reasons:

1. This metadata is relatively small.
2. Partially reconstructed information has limited value. This is due to the layered nature of Stratis. It's not sufficient to know some subset of the device mapping levels. Since they are layered, recovering only some layouts allows no data to be recovered without also knowing how others are mapped on top, and vice versa.
3. Stratis metadata on the block devices should require relatively few updates per day, since the changes it would reflect are blockdevs being added to the pool, or thinpool data device expansions. Infrequent updates reduces the likelihood of corruption[5].

[5] citation needed?

Filesystem metadata is stored on the Metadata Volume on an XFS filesystem. Partial data recovery of that information is possible.

In addition to Stratis-specific metadata, device-mapper layers such as thin, as well as XFS filesystems, all have their own metadata. Stratis would rely on running each of their specific repair/fsck tools in case they reported errors.

10.4 Cache Tier

The Cache Tier is a secondary optional stack that, if present, serves as a cache for the DataTier. If present, the Cache Tier sits directly underneath the Flex Layer. Its structure is similar to the lower levels of the Data Tier.

10.4.1 Requirements

1. Caching may be configured for write-back and write-through modes.
2. Stratis concatenates all cache blockdevs and uses the resulting device to cache the thinpool device. This lets all filesystems benefit from the cache.
3. Cache blocksize should match thinpool datablock size.
4. Removing cache tier comes with performance hit and "rewarming" phase

10.4.2 Cache Tier Metadata

Cache Tier Metadata Requirements

1. Identify all blockdevs that are part of the pool's cache tier and other cache-specific configuration parameters (e.g. WT/WB, block size, cache policy)
2. Cache tier supports up to 8 devices.

10.5 Encryption

stratisd encrypts devices at the blockdev level. If Stratis devices are encrypted, the following conditions will hold:

- Each blockdev will be encrypted with a distinct and randomly generated MEK (Media Encryption Key).
- All blockdevs in a pool will be encrypted, or all blockdevs in a pool will be unencrypted. Stratis will support mixed usage for pools, where some pools are encrypted and others are not.
- A distinct passphrase (used to generate the KEK or Key Encryption Key) will be supported for each pool, although it will be possible to use the same passphrase for every pool, if desired.
- The user will be required to choose whether or not a pool is encrypted at pool creation time and must identify their choice by an additional argument in the CLI's "pool create" command. Encryption of an already existing pool will not be supported.
- Re-encryption will not be supported initially; this functionality will be considered and added in a later step.
- If a pool was encrypted on creation, then all blockdevs added to the data tier will be automatically encrypted with a randomly generated MEK and the pool's designated KEK.

- On any encrypted blockdev, the Stratis metadata will itself be encrypted; it will be inaccessible until the encrypted blockdev is opened.
- The use of a cache tier and of encryption will be mutually exclusive. If a pool is encrypted, an attempt to add a blockdev to the cache tier will be rejected.
- The default cipher specified by cryptsetup 2.1, `aes-xts-plain64`, is used to encrypt all encrypted devices. The MEK is 512 bits.

The above conditions require an implementation which makes use of the LUKS2 header. In particular, for a blockdev to be decrypted with its pool-specific KEK, it will be necessary to include information within the device header which allows the device to be identified as belonging to a particular Stratis pool. The LUKS2 header includes support for LUKS tokens, which will allow stratisd to identify the pool to which a device belongs; the LUKS1 header does not.

11 Implementation Details

11.1 'stratis' command-line tool

Stratis' command-line tool is written in Python. Since it is only used after the system is booted by the adminstrator, Python's interpreted nature and overhead is not a concern.

11.2 stratisd

Stratisd needs to be implemented in a compiled language, in order to meet the requirement that it operate in a preboot environment. A small runtime memory footprint is also important.

stratisd is written in Rust. The key features of Rust that make it a good choice for stratisd are:

- Compiled with minimal runtime (no GC)
- Memory safety, speed, and concurrency
- Strong stdlib, including collections
- Error handling
- Libraries available for D-Bus, devicemapper, and JSON serialization
- FFI to C libs if needed
- Will be available on RHEL 8 in delivery timeframe; currently packaged in Fedora

Other alternatives considered were C and C++. Rust was preferred over them for increased memory safety and productivity reasons.

11.3 devicemapper names

If stratisd terminates unexpectedly and is restarted, it needs to rebuild its knowledge of the running system. This includes not only re-enumerating blockdevs to find Stratis pool members, but also determining the current state of the devicemapper targets that make up pools. A restarting stratisd needs to handle if none, some, or all of the expected DM devices are present, and if present DM devices are working correctly, or in an error state.

To these ends, Stratis uses consistent naming for devicemapper targets. This lets stratisd more easily determine if DM devices already exist, and avoids leaking old DM mappings.

11.3.1 Naming convention Requirements

- Globally unique
- Maximum 127 characters
- Differentiate between Stratis and other DM devices
- Forward-compatible to allow Stratisd updates
- Human-readable
- Easily parsable

11.3.2 Naming Convention

Stratis DM names consist of five required and two optional parts, separated by a '-'.

Part	Name	Max length	Required	Description
1	stratis-id	7	y	Universal DM differentiator: 'stratis'
2	format-version	1	y	Naming convention version: '1'
3	private	7	n	Optional indicator 'private'
4	pool-id or dev-id	32	y	ASCII hex UUID of the associated pool or, in the case of encryption, the block device
5	layer-name	16	y	Name of the Stratis layer this device is in
6	layer-role	16	y	Name of the role of the device within the layer
7	role-unique-id	40	n	Role-specific unique differentiator between multiple devices within the layer with the same role

- The maximum length (adding 6 '-'s as separator) is 125, to stay within the DM name limit of 127 characters.
- "private" is included in names for DM devices that are internal and that should be excluded from content scanning by tools such as blkid.
- Characters for each part are drawn solely from the character classes '[a-z]' and '[0-9]' except that part 7 may also use the '-' character. These restrictions meet D-Bus and udev requirements[6].)
- Encryption uses device UUIDs because there may be multiple encrypted devices in one pool so using the pool UUID would result in naming collisions

11.4 devicemapper minimum version

Stratisd devicemapper minor version 37 or greater, for DM event poll() support and support for event_nr in list_devices ioctl.

11.5 OS Integration: Boot and initrd

Since we want to allow Stratis to be used for system files, Stratis needs to run in the initrd preboot environment. This allows it to activate pools and filesystems so that they can be mounted and accessible during the transition to the main phase of operation.

The use of D-Bus is not possible in the preboot environment. Therefore, Stratis has an alternate IPC mechanism to be used in the initramfs. This can be accessed through the stratis-min and stratisd-min exectutables.

Stratis packages distribute a dracut module, systemd generator, and service file to automate set up of all needed pools, encrypted or unencrypted, during the boot process and `/etc/fstab` mount operations.

[6]See libdm/libdm-common.c _is_whitelisted_char() in the lvm2 code for more.

11.6 OS Integration: udev

The udev library "libudev" enables access to the udev device database. This allows library users to enumerate block devices on the system, and includes attributes describing their contents, such as what filesystem or volume manager signature was detected. (libudev uses libblkid for this, which recently had Stratis signature support added.) The primary benefit of this is to perform the time-consuming block device scan only once, and to alleviate library users from interpreting block device contents.

On boot, Stratis uses libudev to enumerate Stratis block devices on the system, reads the Stratis metadata from each, and activates pools that are complete. Later, during the main running phase, Stratis monitors udev events for newly-added block devices, so that if missing Stratis pool members are connected to complete a pool, the pool can be activated and used.

11.7 OS Integration: /dev entries

Stratis allows the user to create filesystems, which then can be mounted and used via `mount(8)` and the `fstab(5)`. Stratisd provides a udev rule that generates a `/dev/stratis` directory. It creates `/dev/stratis/<poolname>` for each pool present on the system, and `/dev/stratis/<poolname>/<filesystemname>` for each filesystem within the pool. Changes such as creations, removals, and renames are reflected in the entries under `/dev/stratis`. These entries give the user a well-known path to a device to use for mounting the Stratis filesystem. Filesystems may also be listed in `/etc/fstab` using XFS UUID. However, if the device is encrypted, it is recommended that users take advantage of the `stratis-fstab-setup@.service` systemd unit file to automate pool unlock prior to mounting.

11.8 Snapshots

Stratis's current snapshot implementation is characterized by a few traits:

- A snapshot and its origin are not linked in lifetime. i.e. a snapshotted filesystem may live longer than the filesystem it was created from.
- A snapshot of a filesystem is another filesystem.
- A filesystem may be snapshotted while it is mounted or unmounted.
- Each snapshot uses around half a gigabyte of actual backing storage, which is needed for the XFS filesystem's log.

These may change in the future.

11.9 Backstore Internals

The backstore is divided into two tiers: the data tier, and an optional cache tier. Each tier has its own set of physical block devices. The goal of each tier is to provide a single linear device that the flex layer (or another tier) can easily build on top of.

A tier is created with a certain feature set, which results in an internal layering of devices as needed to support those features. The features a tier supports are fixed at tier creation time. However, the block devices that make up the tier may change. New blockdevs may be added.

This requires each tier to support:

- add_blockdev (add a new blockdev to the tier)
- blockdevs (list/iterate)

The tier includes optional internal support for multiple features, which also are implemented using DM devices.

At the "bottom" of the tier are blockdevs. These blockdevs are mapped through layers that add value, such as encryption.

Each layer takes a list of blockdevs and converts it to a list of "better" blockdevs, whose total size is likely different.

While each intermediate layer may provide an array of blockdevs, the "cap" layer of the tier presents a single linear blockdev that maintains the location of each presented block and never shrinks, and hides the interior complexity of the tier from upper users.

The ordering of layers (from bottom to top) within a tier is:

1. blockdev
   Blockdevs supply available space to the tier.

2. encryption
   This layer provides optional encryption for all block devices available in the pool.

3. cap
   If presented with more than one blockdev, or the blockdev has a nonzero offset, the "cap" layer will ensure the Tier presents a single blockdev with consistent block mapping for use above the tier by creating a Linear device that never relocates previously-mapped block ranges.

11.9.1 Demand-based allocations

Layers should not consume the entire space available to them when constructing devices, but instead grow existing mapped allocations (or create new ones) as the total demands of upper layers grow larger. This is preferred over a "greedy" strategy because it provides a better user experience to allow the amount of space allocated to both data and metadata to be calculated dynamically if requirements change.

11.10 Operation States

When encountering errors, Stratis must handle them if possible, but there are also errors that are severe enough to hamper Stratis's ability to function. When these occur, instead of terminating, Stratis continues by transitioning to a less-capable operation state. This allows some measure of continued monitoring and enables its condition to be visible to the user through the API.

Action availability state	Description
fully_operational	All pool operations are available in this state. No restrictions are imposed.
no_ipc_requests	Pool operations triggered by an IPC request are disabled. This is often due to a failed IPC command that could not be fully rolled back. Manual resolution of the bad roll back state will allow the pool to resume fully operational state again.
no_pool_changes	Any operations that modify the state of the pool will be disabled. This includes IPC requests and background operations such as extending the thin pool and filesystem.

11.11 Licensing

Currently, stratisd and all Rust libraries included in the stratis-storage organization meant for stratisd are licensed as MPLv2. This conclusion was reached after careful consideration of

compatibility with other licenses in the open source ecosystem. MPLv2 allows some major benefits:

1. It is compatible with more permissive licenses in the open source ecosystem such as BSD and MIT licenses. Given the prevalence of more permissive licenses in the Rust ecosystem, this is an important consideration.

2. It is compatible with GPL code. This permits stratisd to incorporate GPL code and become effectively GPL without ever changing the MPLv2 license.

3. If GPL code is incompatible with the license of a dependency added later, MPLv2 allows the removal of GPL code and migration of the GPL code into a separate service to permit the addition of the conflicting dependency. This can all be done without relicensing.

After evaluating the options, the MPLv2 seemed to be the most flexible license that still fulfilled the requirements for this project.

References

[1] Anne Mulhern. justbytes documentation. `http://pythonhosted.org/justbytes/`.

[2] Anne Mulhern. The computation and representation of address ranges: With an introduction to the python `justbytes` library. `https://mulhern.fedorapeople.org/justbytes_presentation.pdf`, August 2016.

Appendices

A Initial options for adapting existing solutions

As part of early requirements-gathering, the team looked at existing projects in this space, both as candidates for building upon to create a solution, as well as if an existing project could be extended to meet the requirements.

A.1 Extending an existing project

A.1.1 SSM

System Storage Manager (SSM) provides a command line interface to manage storage in existing technologies. Our interest in SSM was to determine if it would be an existing project we could extend to meet our requirements.

SSM provides a unified interface for three different "backends": LVM, Btrfs, and crypto. However, if we wish to provide a simple, unified experience, the first step would likely be to pick one of the backends and build around its capabilities. This eliminates complexity from the CLI -- no need for the user to pick a backend or encounter commands that happen to not work based upon the chosen backend, but obviates much of the point of SSM.

SSM does not provide a programmatic API. It internally contains "ssmlib", which could be enhanced and exposed, but would be Python-only. ssmlib is also built around executing command-line tools, which can cause issues.

SSM is not a daemon. We'd need to modify SSM to operate on a daemon model. An ongoing presence is needed for fault monitoring but also automatic filesystem and thinpool extensions.

SSM doesn't currently support RAID5/6, thin provisioning, or configuring a cache tier.

SSM is written in Python, which would limit its ability to be used in an early-boot environment.

SSM does not provide functionality for error recovery. If the storage stack encounters an error the user has to use the individual tools in the stack to correct. Thus greatly diminishing the ease of use aspect and value proposition of SSM.

Analysis

Extending SSM does not meet the requirements.

A.1.2 LVM2

Logical Volume Manager (LVM2) is the nearly universally-used volume manager on Linux. It provides the "policy" that controls device-mapper. It adds:

- On-disk metadata format to save and restore configuration across boot
- Usage model built on Physical Volume, Volume Group, and Logical Volume (PV, VG, LV) concepts.
- A comprehensive set of command line tools for configuring linear, raid, thinpool, cache, and other device-mapper capabilities
- Monitoring, error handling, and recovery
- LV resize; PVs may be added or removed from a VG
- Snapshots and thin snapshots
- Choice of user-guided or automatic allocation/layout within the VG

Analysis

Adding the capability to manage filesystems to LVM isn't something that has been much considered. Extending LVM2 would make it very hard to achieve simplicity of interface, given the conflicting requirement to maintain backwards compatibility with what LVM provides now.

A.2 Building upon existing projects

A.2.1 XFS

XFS is a highly respected non-volume-managing filesystem. To meet the goal of eliminating manual filesystem resizing by the user, Stratis requires the filesystem used have online resize (or at least online grow) capabilities, which XFS does. In the absence of online shrink, Stratis would rely on trim to reclaim space from an enlarged but mostly empty filesystem, and return it to the thin pool for use by other filesystems.

Use of XFS on top of thin provisioning also makes proper initial sizing important, as well as choosing sizes for XFS on-disk allocations that match those used by the underlying thin-provisioning layer, to ensure behavior with the two layers is optimal.

Analysis

XFS meets the requirements and currently seems like the best choice.

A.2.2 devicemapper

devicemapper is a framework provided by the Linux kernel for creating enhanced-functionality block devices on top of physical block devices. These new devices can add support for RAID, thin provisioning, encryption, multipath devices, caching devices, and more. The framework provides the ability to configure and layer these capabilities, but no facilities for saving or restoring a configuration. devicemapper provides mechanism, but no policy.

Analysis

Using devicemapper directly would require that an upper layer implement its own on-disk metadata format and handle some tasks in a similar manner to LVM2.

A.2.3 LVM2

See A.1.2 for a description of LVM2 capabilities.

LVM is a mature software project that implements volume management. For Stratis, the question is whether the benefits of internally using LVM2 outweigh the costs.

Issues with Building on LVM2

Note: This assumes the implementation described in Part III.
Note: lvm-team has raised objections to items on this list.

- Policy+mechanism vs policy+policy+mechanism: LVM2 is configurable but has limitations. e.g. we might wish to let the user define a block device as only to be used to replace a failed disk in a raidset. However LVM raid_fault_policy="allocate" will use *any* free PV, not just one explicitly reserved.
- A good API needs the ability to convey meaningful and consistent errors for other applications to interpret. The lvm command line employs a simple exit code strategy. The error reason is embedded in stderr in free form text that changes without notice. Thus it is virtually impossible for any lvm command line wrapper to provide meaningful and consistent error codes other than success or failure. Note: Lvm has recently added JSON output which contains the ability to add more meaningful and useful error codes, but this functionality is not implemented and non-trivial in scope to complete.
- lvm-dbus cannot be used because it requires Python and D-Bus, neither of which are available in initrd
- Stratis-managed LVM2 devices would show up in LVM2 device & volume listings, which would cause user confusion
- Using LVM2 for metadata tracking is good, but only if upper layer has no metadata storage needs of its own. What about tags? Tags can't store JSON objects since '[]{},"' are not allowed in tags.
- LVM2 metadata format prevents new metadata schemes, such as tracking thin volumes separately from PV metadata, or metadata backup copy not also at the tail of the blockdev.
- Use of new device-mapper features delayed by LVM2 implementation and release cycle.
- One big argument by LVM2 proponents is that LVM2 is a large, long-lived project that has learned many things the hard way, and it would be foolish to abandon all that value by starting over.
  - Must we use the code, or can we take lessons from LVM2 devs and incorporate them independently? Maybe fix some things that backwards-compatibility makes impossible to fix in LVM2?
  - Large parts of the codebase don't benefit Stratis:
    * File-based & configurable Locking: not needed since everything is serialized through stratisd
    * daemons/* including clvmd
    * Udev: stratisd assumes udev & listens for udev events
    * Filter/global_filter

    * Caching: not needed, daemon is authoritative
    * profiles
    * preferred names ordering
    * lvm.conf display settings: not needed, up to API client
    * dev_manager: Stratis layers are predefined, much simpler
    * config_tree
    * report: beyond Stratis scope
    * Command-line tools, option parsing: handled in cli, reduced in scope
    * lib/misc/*: not needed or handled via libraries
    * Multi-metadata-format support
  - What would Stratis benefit from?
    * on-disk metadata format
    * Best policy for duplicate/absent/corrupted block devices
    * fault tolerance/recovery
    * pool/snapshot monitoring

Analysis

While we cannot dismiss using LVM as an option for the future, currently there are some areas that it does not meet Stratis requirements. There are also questions about the best way to interface with LVM that need to be resolved prior to its adoption.

A.3 Conclusions

Based on looking at the existing available building blocks, the best option is to build Stratis as a new project that initially makes use of XFS and device-mapper in its implementation. In parallel, request enhancements to LVM2 to enable its substitution for device-mapper when the enhancements are implemented. This lets Stratis proceed without delay to a point where it can be placed in prospective users' hands to start getting feedback, and will allow Stratis to eventually use LVM2, and avoid duplicating functionality that LVM2 already provides.

B Encryption implementation details

Stratis uses libcryptsetup to manage FDE (full disk encryption) for each block device added to an encrypted pool.

B.1 On-disk format

Stratis uses the LUKS2 encryption format as the basis for the encryption implementation. The LUKS2 header contains information that can be used to identify a device as a Stratis encrypted device.

B.1.1 LUKS2 token format

LUKS2 tokens contain the necessary information to describe how to unlock the device. No private information is contained in the tokens. Stratis uses two tokens, one standard cryptsetup token which allows activating the device using a passphrase stored in the kernel keyring and another Stratis-defined token that contains activation information required by Stratis. Token IDs are statically assigned for each token for a number of reasons:

- Using static IDs, stratisd does not have to do a scan of all of the tokens followed by heuristics to recognize a token. Instead, it chooses the static ID, and if the format does not match what was expected, it is not treated as a Stratis encrypted device.

- As other activation methods are added, activation of a pool can specify one token ID to use for activation of all devices in the pool and expect that they will all be unlocked using the same mechanism.
- New tokens may be added without breaking changes to the assignment of existing token IDs.

The Stratis token has an ID of 0 while the LUKS2 keyring token has an ID of 1.

LUKS2 keyring token

The kernel keyring token contains information for LUKS2 integration with keys in the kernel keyring. This token is a standard token supported by cryptsetup, and details for the format can be found in the cryptsetup documentation. Interaction with this token is handled by standard libcryptsetup API methods.

Stratis token

The Stratis token is used for device identification and activation, and the values are set directly by Stratis. The format specification is as follows:

```
{
        "type": "stratis",
        "keyslots": [],
        "activation_name": <DEVICEMAPPER_NAME>,
        "pool_uuid": <POOL_UUID>,
        "device_uuid": <DEVICE_UUID>
}
```

See below for an explanation of each JSON value:

<DEVICEMAPPER_NAME>

This is the managed name that will show up in devicemapper when the device is activated using libcryptsetup. While the name can be seen on the system by querying devicemapper, managing encrypted Stratis devices outside of Stratis is not supported so this is considered an internal name.

<POOL_UUID>

This UUID corresponds to the Stratis pool that owns this block device. The pool UUID in the token matches the pool UUID in the encrypted Stratis metadata.

<DEVICE_UUID>

This UUID corresponds to the device UUID for this specific block device. The device UUID in the token matches the device UUID in the encrypted Stratis metadata.

B.1.2 Encrypted Stratis metadata format

Once the LUKS2 volume is unlocked, the encrypted Stratis metadata should be accessible through the unencrypted logical devicemapper device in exactly the same way as an unencrypted Stratis device.

B.2 Encrypted device creation

Stratis does not wipe the device prior to initializing it with a LUKS2 header. This means that an attacker can deduce the following from examining a disk that has been initialized using Stratis without any prior preparation:

- The amount of data currently stored on disk that is encrypted. For example, if a disk has sectors that were zeroed or contain unencrypted data previously stored on the disk at the end of an encrypted segment, this can give insight into how many encrypted blocks are stored on the device. To prevent this, the device should be wiped with random data prior to providing it to a Stratis pool.
- The contents of unencrypted data that was previously stored on the disk prior to initialization by Stratis. cryptsetup's LUKS2 implementation does not overwrite all of the data on the disk by default, so without prior preparation, any sectors that have not been overwritten with encrypted data will still contain the data previously stored on the disk. To prevent this, the disk should be zeroed or wiped with random data.

B.3 Encrypted device discovery

Stratis identifies encrypted devices belonging to Stratis by their Stratis token. It does not attempt to activate any device until it receives a D-Bus command to activate all devices. When the command is received, stratisd attempts to unlock the devices using the LUKS2 keyring token previously set by stratisd. If activation of devices yields a set of devices that can form a complete pool, the pool is set up.

B.4 Encrypted device activation

Devices are activated with the activation name in the Stratis token. See B.1.1 for an explanation of the Stratis token. Device activation results in a new, activated device path with the canonical path name

```
/dev/mapper/<activation_name>
```

where the unencrypted data can be accessed. Because of the design and use of libcryptsetup, encrypted devices can also be activated outside of Stratis using libcryptsetup or the cryptsetup CLI as long as the required passphrase is in the kernel keyring.

B.5 Encrypted device destruction

Destroying an encrypted Stratis device does not wipe the entire device as the cost of this operation is linearly proportional with respect to disk size. Instead, Stratis uses libcryptsetup to destroy all of the keyslots, wiping all of the encrypted MEK (media encryption key) data in each keyslot. It then obtains the size of the LUKS2 metadata on this device and does an additional wipe of the remaining LUKS2 metadata. This provides the following device properties on pool destruction:

- The MEK data is destroyed so the data stored on the device will be unrecoverable.
- If this device is used when creating another pool later, Stratis will be able to reinitialize it and will consider this device unowned due to the wipe of the LUKS2 metadata.
- The encrypted data will be left on the device but will be inaccessible due to the destruction of the key.

Additional security may be achieved by zeroing the disk or overwriting it with random data using external tools.

B.6 Key consistency for encrypted pools

Stratis takes steps to ensure that the same passphrase used during initial creation of the pool is the passphrase that is used for adding encrypted block devices at a later time. stratisd runs a check prior to adding block devices to an encrypted pool to verify that the passphrase in the

kernel keyring with the key description recorded in the LUKS2 metadata can unlock the block devices that have already been encrypted and added to the pool. This prevents the passphrase data associated with a key description from being accidentally or intentionally changed between pool creation and addition of encrypted block devices which would result in an unusable pool and loss of data on the next occasion when the pool must be set up.

B.7 Key management

Stratis provides facilities for key management. The kernel keyring is used as the backing store for the keys provided through the D-Bus API to allow proper access controls for passphrases. The architecture of key input has been carefully designed to avoid leaving keys in memory after they are no longer in use. The key is not directly sent over D-Bus; D-Bus traffic is not encrypted so this would leak the plaintext passphrase. Instead, the client side provides the server side with a file descriptor from which to read the passphrase so that no data is ever exposed in the D-Bus method call. When a user provides a key using the interactive mode in the CLI or D-Bus API, input buffering is turned off on the client side so the key is never stored in userspace memory. On the server side, stratisd reads the key into a memory block managed by libcryptsetup so that as soon as the memory is no longer in use, libcryptsetup will wipe the memory using a method that will not be optimized out by the compiler. All of this guarantees that the key data will only be available in one memory location while it is in use, and that the memory will be securely wiped after use.