Stratis Software Design: Version 0.8.2*

Andy Grover <agrover@redhat.com>

Last modified: 04/04/2017

Contents

Asking Questions and Making Changes to this Document

This document can be found in the stratis-docs repo, and is written using LyX 2.2.2. Please ask any questions by opening an issue, and propose changes as pull requests.

Executive Summary

Stratis is a new tool that meets the needs of Red Hat Enterprise Linux (RHEL) users calling for an easily configured, tightly integrated solution for storage that works within the existing Red Hat 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. Extending or building on SSM 2.1.1 or LVM 2.1.2 was carefully considered. SSM did not meet the design requirements, but building upon LVM may be possible with some development effort.

Part I
Background

1 Problem Statement

RHEL 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 tools to achieve a complete result. Furthermore, since storage administration tools are focused on the command-line, higher-level management tools must each individually build upon the inherently risky proposition of treating the human-focused syntax and output of command-line interface (CLI) tools as a stable programmatic API. This causes a waste of effort and opportunity for bugs, as each administration 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 but leads to a huge number of possible configurations. This requires the user manage the stack because there's not enough commonality to enable automation.

However, there is generally a single design that can work for most use cases. By assuming a fixed layering of storage features (some of which may be optional), we enable software to effectively manage these on behalf of the user.

Once this is done, the details of how the solution is implemented are less of a concern to the user. The user can express what remains – what hardware resources to use, what features are desired, how storage should be logically presented – using a small number of object types with well-defined relations. This leaves software in a good position to manage the stack and handle most 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 RHEL. ZFS isn't an option RHEL can embrace due to licensing (Ubuntu notwithstanding.) Btrfs has no licensing issues, but after many years of work it still has significant technical issues that may never be resolved.

We can see from the tremendous resources that have gone into these two projects that writing a VMF is a tremendous, time-consuming effort. 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 sizable stack of layered blockdevs doing all the work 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, integrity, redundancy, 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
    (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 – no Python or DBus – but can use device-mapper

7. Bootable (planned – see 12.6)

   (a) Feels like a "real" filesystem if no secondary filesystem is needed for boot

   (b) Enables Stratis features to be used by system image, e.g. booting from a snapshot, and allowing /boot to grow

   (c) Requires explicit support in bootloader (Grub2)

   (d) device-mapper not available

8. Adaptable to emerging storage technologies

   (a) Persistent memory

      i. Block-appearing pmem can be used by Stratis

      ii. byte-addressible pmem see 12.6

9. Implementable in 1-2 years

   (a) We're already behind, waiting another 10 years isn't an option

2 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.

2.1 Extending an existing project

2.1.1 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.

2.1.2 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.

2.2 Building upon existing projects

2.2.1 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. 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.

2.2.2 device-mapper

device-mapper 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. device-mapper provides mechanism, but no policy.

Analysis

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

2.2.3 LVM2

See 2.1.2 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.

2.3 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.

Part II
Solution Overview

3 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.

4 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 directly, 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.

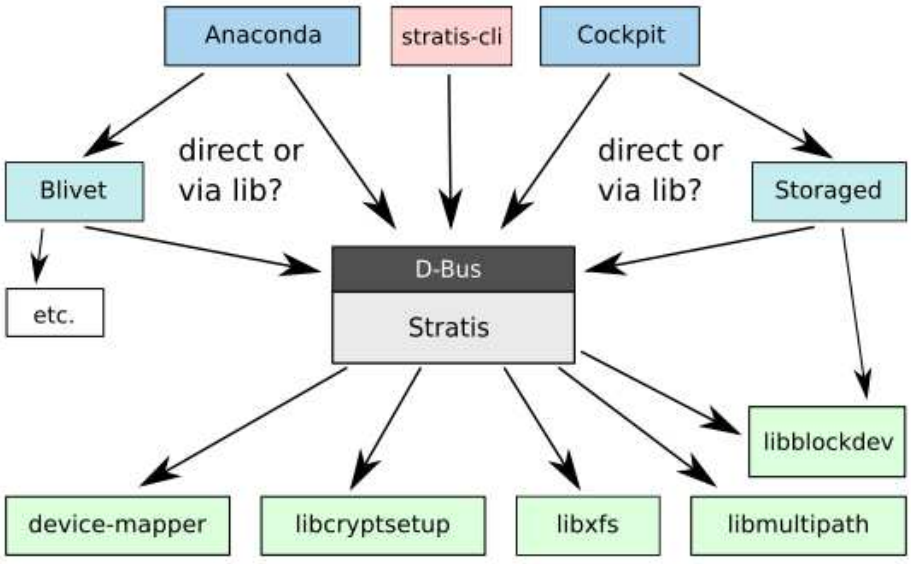

Figure 1: Stratis in the Linux Storage Management Stack

There are existing libraries that handle CLI-to-API for both Anaconda and Cockpit. These could be extended to support Stratis, or not.

5 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 unlimited blockdevs and 2^{24}filesystems per pool. However, practical limits on these values may compell users to restrict themselves to significantly smaller numbers of blockdevs and filesystems.

A pool must be created with an initial set of blockdevs in order to be usable. 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.

A new filesystem is either a new empty filesystem or a snapshot of an existing filesystem within the pool. Immediately after creation a snapshot and its origin will reference the same set of data blocks. Although Stratis does not distinguish between snapshots and filesystems, treating both as first-class filesystems, Stratis will maintain the relation between the snapshot and its origin to aid users.

6 Scalability and Performance Considerations

Stratis doesn't optimize performance within its data tier. The justification is that adding an SSD caching tier, or even moving to all-SSD storage, is likely to be a better solution if performance is a consideration. Stratis therefore focuses only on resiliency, integrity, and ease of management in the data tier.

Part III
Implementation

7 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.

8 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.

8.1 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, which ideally would be hidden from the user.
- Stratis requires a userspace daemon, which must remain running at all times for proper monitoring and pool maintenance.

9 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.

9.1 Overview

The D-Bus API is a thin layer which receives message on the D-Bus, processes them, transmits them to the Stratis engine, receives the results from the engine, and transmits a response on the D-Bus. It does the minimum amount of processing necessary to invoke the methods of the underlying engine and to place the results on the D-Bus. 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.

When generating property values, it need not process arguments, since there are none, nor does it add or remove objects from the D-Bus tree. It may still invoke engine methods in order to obtain the property values.

The D-Bus API is implemented using the dbus-rs library[4].

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

9.2 D-Bus Access Control

9.2.1 Security Policy

9.2.2 Prevent Spoofing

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.

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

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 (and eventually remove) individual blockdevs to grow the available space available to filesystems
4. User-selectable redundancy level (per-pool granularity)[5]
5. Integrity checking[6]
6. Encryption[7]

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 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, and storing pool metadata on each blockdev. Blockdevs may be in the following states: good-unused, good-used, bad, spare. Minimum blockdev size Stratis will use is 1 GiB.

10.2.2 Layer 1: Integrity (optional)[8]

This layer 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.

10.2.3 Layer 2: Redundancy (optional)[9]

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, 1E[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[11]), and differently-sized blockdevs, a redundant Stratis pool may 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 because it cannot be used in a RAID set. Stratis will intensively manage raidsets, extending them across newly added blockdevs or creating new

[5] in a future version
[6] in a future version
[7] in a future version
[8] in a future version. This will require a new device-mapper target be developed.
[9] in a future version
[10] See 'linux/Documentation/device-mapper/dm-raid.txt' for more info
[11] More are supported by DM, but too many increase the likelihood of individual failures.

raidsets; and handling the removal of blockdevs[12]. Stratis may use dm-raid's reshape capabilities when possible, although this changes the stripe size and could cause issues.

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 ugly edge cases.

10.2.4 Layer 3: Flexibility

Whether blockdevs are part of raidsets or used directly, pools need to cope with the addition or removal[13] of them.

Stratis must allow adding a blockdev to an existing pool, and use it to grow the pool's allocated space. (This may involve reshaping one or more raidsets to be wider, or just knowing to extend the thinpool onto the new blockdev when other blockdevs are full.)

In a future version, Stratis will calculate if a blockdev can be removed from the pool with no effect, removed with loss of redundancy, or not removed without data loss, and allow, allow-and-warn, or disallow removal, as appropriate.

It should also 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.

The flexibility layer will support two linear DM devices made up of segments from lower-level devices, which will be used by Layer 4 (Thin Provisioning) as metadata and data devices. It will track what lower-level devices they are allocated from, allow the two devices to grow, and handle the online movement of segments in these devices when lower-level devices come and go.

Furthermore, the flexibility layer will support a third linear DM device called the Metadata Device that will be used to store metadata about upper layers.

All three devices in this layer may be built on L0, L1, or L2 devices, depending on configuration. The first two will share a configuration, but the Metadata Device may have a separate configuration (i.e. Metadata Device could be redundant, even while the other two are not).

10.2.5 Layer 4: Thin Provisioning

The two linear targets from L3 are used as metadata device and data device for a DM thinpool device. Stratis manages the thinpool device by extending the two L3 subdevices when either runs low on available blocks. If the pool approaches a point where the pool no longer has empty lower-level space to extend onto, Stratis alerts the user and takes action to avoid data corruption. Actions could include switching filesystems to read-only mode, running fstrim, or progressively throttling writes.

10.2.6 Layer 5: 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.

[12] in a future version
[13] in a future version

10.2.7 Layer 6: Encryption (optional)[14]

Stratis could enable per-filesystem encryption between the thin device and the filesystem.

10.2.8 Layer 7: Filesystem

Stratis monitors each filesystem's usage against its capacity and automatically extends (changing the thin dev's logical size and then using e.g. xfs_growfs) them online without user intervention. Stratis also periodically will run fstrim to reclaim unused space during idle periods. Idle periods will be found based upon current and historical system I/O activity levels.

10.3 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 three linear targets that span the L2 devices (L3), the thinpool device (L4) and the attributes of the thin devices (L5) and filesystems created from the thinpool (L7).

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[15]
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[16].

All UUIDs are written as un-hyphenated ASCII encodings of their lower-case hexadecimal representation[17].

[14] in a future version

[15] 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.

[16] 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.

[17] 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.

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 Device (Flex Layer)

(Specific DM targets such as raid, integrity, and thinpool also place their own metadata on disk.)

Information on levels 0-4 is duplicated across all blockdevs within a 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 Device stores metadata on Layers 5 and up in a conventional block device and filesystem that is built on top of the layers beneath it. Choosing to split 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 preexisting lower layers, and work under looser restrictions around updating in place, and the total size to which it may grow. It can apply existing, well-tested solutions for solving data organization and storage issues, such as a general-purpose filesystem. This is the approach Stratis takes[18].

[18] It would also be possible to store lower-level metadata on a /boot partition. This would still enable Stratis to be used for the root and other core filesystems, but would prevent /boot from being *in* Stratis. This has to be weighed against the work required to define a new on-disk metadata format. Our hope is that the split-metadata design allows the on-disk metadata format's implementation to be easy enough that we can avoid the dependency on a non-Stratis /boot partition.

10.3.4 BlockDev Data Area (BDA)

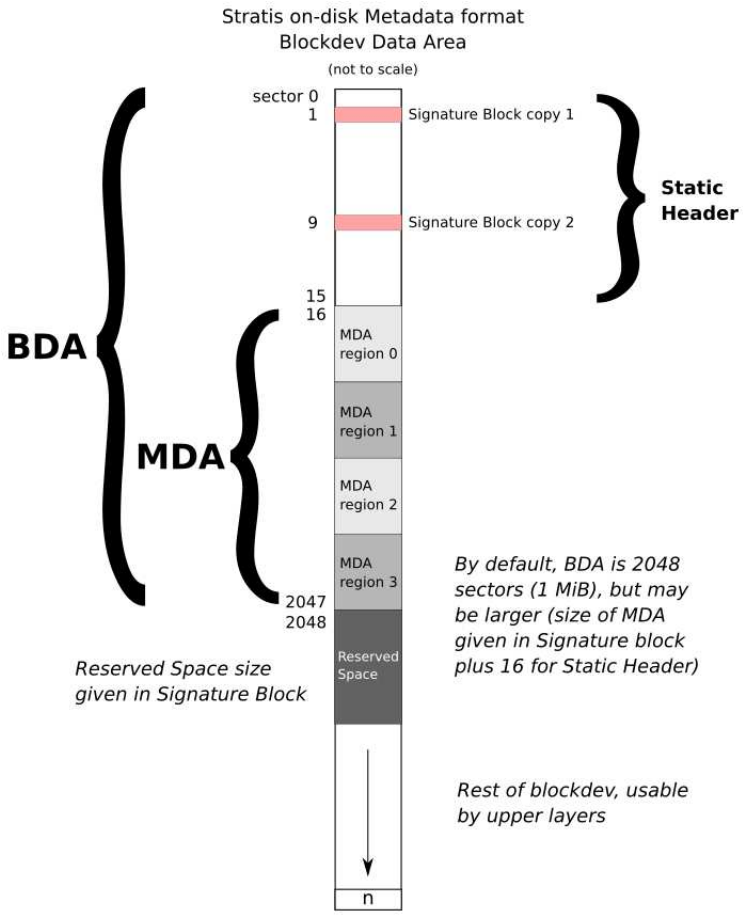

Figure 2: BDA format

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.

Static Header

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

Signature Block

byte offset	length (bytes)	description
0	4	IEEE 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	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)
124	392	unused

- No flags are yet defined, so 'flags' field is zeroed.
- All 'unused' fields are zeroed, and are reserved for future use.
- Blockdev metadata area length (offset 96) must be a number divisible by four of at least 2032 – combined minimum length of overall BDA (static header and MDA) is 2048 sectors (1 MiB).
- The BDA is followed immediately by *reserved space*, whose size is specified in the signature block (offset 104). Reserved space is used by Stratis for temporary storage.
- When a blockdev is removed from a pool, or is part of a pool that is destroyed, Stratis wipes the Static Header.

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	IEEE CRC32 covering remainder of MDA header
4	4	IEEE CRC32 covering JSON data
8	8	length of JSON data in bytes (u64)
16	8	UNIX timestamp (seconds since Jan 1 1970) (u64)
24	4	nanoseconds (u32)
28	4	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.
- Software will generally read Signature Block copy 1 and MDA regions 0 and 1, and only reference additional copies if the CRC check fails.

10.3.5 Metadata Area (MDA)

The MDA contains a JSON object that represents the pool's overall configuration of blockdevs, from L0 to L4. See Appendix A for the formal JSON schema.

Top level objects:

key	JSON type	required	description
name	string	y	the name of the pool
block_devs	object	y	the block devices in the pool
integrity_devs	object	n	info on integrity devices & layout
raid_devs	object	n	layout of the raid devices
flex_devs	object	y	layout of the data and metadata linear devices
thinpool_dev	object	y	parameters of the thinpool device

block_devs: an object with UUID keys and object values with the following structure:

key	JSON type	required	description
dev	string	n	the blockdev's most recently recorded device node
size	integer	y	the blockdev's size in sectors
location	string	n	user-provided information for tracking the device
disk_id	string	n	uniquely identifying information for the blockdev, such as SCSI VPD83 or serial number

integrity_devs: TBD

raid_devs: TBD

flex_devs: an object with three keys: "meta_dev", "thin_meta_dev", and "thin_data_dev". Each corresponding value is an array of objects. These objects define the linear segments that make up each device. Each segment object has the following structure:

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

parent: may be a valid blockdev, integrity dev, or raid dev UUID.

thinpool_dev: an object with the following structure:

key	JSON type	required	description
data_block_size	integer	y	the size in sectors of the thinpool data block

10.3.6 Metadata Device

The Metadata Device 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 CRCs over the critical L0-L4 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 – it concerns disks and raidsets, of which the pool will have only a small number, so having multiple entire copies isn't terribly wasteful.

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 e.g. L0-L2 layouts allows no data to be recovered without also knowing how L3 and L4 are mapped on top, and vice versa.

3. L0-L4 metadata should require relatively few updates per day, since the changes it would reflect are blockdevs being added/removed from the pool, or thinpool data device expansions. Infrequent updates reduces the likelihood of corruption[19].

L5-L7 is stored on the Metadata Device on an XFS filesystem, so partial data recovery of that information is possible.

In addition to Stratis-specific metadata, device-mapper layers such as cache, raid, 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 Requirements

1. Caching may be configured for redundancy, or no redundancy.

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

[19] citation needed?

3. Stratis concatenates all cache blockdevs and uses the resulting device to cache the thinpool device (L4). This lets all filesystems benefit from the cache.
4. Cache blocksize should match thinpool datablock size.
5. Removing cache tier comes with performance hit and "rewarming" phase
6. For write-back caching, Cache tier must be redundant if data tier is redundant.
7. (more details to fill in here.)

10.5 Cache Tier Metadata

10.5.1 Cache Tier Metadata Requirements

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

Part IV
Development Plan

11 Implementation Choices

11.1 'stratis' command-line tool

Stratis' command-line tool is currently written in Python. Since this 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 DBus, device-mapper, JSON serialization, and CRC
- FFI to C libs if needed
- Will be available on RHEL 7 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.

12 Delivery of Features

12.1 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
17. thin/cache metadata validation/check (i.e. call thin_check & cache_check)

12.2 Stratis version 0.5

Add cache tier & basic snapshot support

1. List cache blockdevs in a pool
2. Add cache blockdevs
3. Remove cache blockdevs
4. Write-through caching only
5. Create/destroy primary snapshots

12.3 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. Notification method to the user if pool is approaching user or system-defined 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

12.4 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. Quotas
9. Blockdev resize (larger)
10. Spares

12.5 Stratis version 3.0

Rough ZFS feature parity. New DM features needed.

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

12.6 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)
7. Support for byte-addressible persistent memory

13 Schedule

Very tentative: 0.1 1H 2017, 0.5 2H 2017, 1.0 1H 2018.

14 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.

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

Appendices

A JSON Schema for MDA

```
{
    "$schema": "http://json-schema.org/schema#",
    "title": "Stratis Variable Length On-Disk Metadata",
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
            "description": "User selected name"
        },
        "block_devs": {
            "type": "object",
            "description": "Block devices in pool",
            "patternProperties": {
                "^[0-9a-f]{32}$": {
                    "type": "string",
                    "description": "Device",
                    "required": ["size"],
                    "properties": {
                        "dev": {
                            "type": "string",
                            "pattern": "^/dev/.*$",
                            "description" : "Most recently recorded device node"
                        },
                        "size": {
                            "type": "integer",
                            "description": "Size in sectors",
                            "minimum": 0
                        },
                        "location": {
                            "type": "string",
                            "description": "User provided tracking information"
                        },
                        "disk_id": {
                            "type": "string",
                            "description": "Uniquely identifying id"
                        }
                    }
                }
            },
            "additionalProperties": false
        },
        "flex_devs": {
            "type": "object",
            "properties": {
                "meta_dev": { "$ref": "#/definitions/flexDevice"},
                "thin_meta_dev": { "$ref": "#/definitions/flexDevice"},
                "thin_data_dev": { "$ref": "#/definitions/flexDevice"}
            },
            "required": ["meta_dev", "thin_meta_dev", "thin_data_dev"],
            "additionalProperties": false
        },
        "thin_pool_dev": {
            "type": "object",
            "properties": {
                "data_block_size": {
                    "type": "integer",
                    "description": "Size of thinpool data blocks in sectors",
                    "minimum": 1
                }
            },
            "required": ["data_block_size"],
            "additionalProperties": false
        }
    },
    "required": ["name", "block_devs", "flex_devs", "thin_pool_dev"],
    "additionalProperties": false,
    "definitions": {
        "flexDevice": {
            "properties": {
                "parent": {
                    "type": "string",
                    "pattern": "^[0-9a-f]{32}$",
                    "description": "UUID of the segment's device"
                },
                "start": {
                    "type": "integer",
                    "description": "the starting sector offset",
                    "minimum": 0
                },
                "length": {
                    "type": "integer",
                    "description": "the length in sectors",
                    "minimum": 0
                }
            },
            "required": ["parent", "start", "length"],
            "additionalProperties": false
        }
    }
}
```