

Boot-to-snapshot design v0.6

Bryn M. Reeves <bmr@redhat.com>,
Marion Contos <mcsontos@redhat.com>,
Zdenek Kabelac <zkabelac@redhat.com>.

Last modified: 08/06/17

Contents

I	Background	4
1	Problem Statement	4
1.1	Bootting snapshots	5
1.1.1	Bootting snapshots of LVM2 volumes	6
1.1.2	Bootting snapshots of BTRFS volumes	6
1.2	Core Goals	7
1.2.1	Goal: Enable easy selection of bootable snapshots	7
1.2.2	Goal: Automate the management of boot entries for snapshots	7
1.3	Additional Goals	7
1.3.1	Goal: allow substitution of selected VFS mounts for snapshot versions	7
1.3.2	Goal: Simplify creation of point-in-time snapshots of multiple file systems	7
1.4	Future extensions	8
1.4.1	Goal: Simplify management of large numbers of snapshots	8
1.4.2	Goal: Simplify file operations on inactive snapshot volumes	8
1.4.3	Goal: Automatic rollback in the event of a failed boot	8
2	Linux boot process management	9
2.1	Early user space interaction	10
2.2	Boot loader entries	11
2.3	Bootloader abstractions	11
2.3.1	Grubby	11
2.3.2	perl-Bootloader	11
2.4	BootLoader Specification	12
2.4.1	BLS Implementations	13
2.4.2	BLS configuration snippets	13
2.4.3	Proposed BLS enhancements (i)	14
2.4.4	Proposed BLS enhancements (ii)	14
2.5	Other proposals	15
2.5.1	Directly booting LVM2 volumes and supporting “/boot” on LVM2 devices	15
2.5.2	Adopting an existing bootloader abstraction	16
3	Existing solutions	16
3.1	SNAPPER	16
3.2	SOLARIS ZFS and beadm	17
3.3	STRATIS STORAGE	18

II	Solution overview	18
4	User stories	18
4.1	Snapshot Manager	18
4.1.1	Consistency checking	18
4.1.2	Point-in-time backups	19
4.1.3	Frequent snapshots	19
4.1.4	Before-and-after snapshots	19
4.1.5	Testing updates (success)	20
4.1.6	Testing updates (failure)	20
4.1.7	Creating stable testing images	21
4.2	File system substitution	21
4.3	Boot Manager	21
4.3.1	Creating a boot entry for an existing snapshot (LVM2)	21
4.3.2	Finding information on an existing entry	22
4.3.3	Checking the currently selected defaults	22
4.3.4	Overriding the selected entry for subsequent boots	22
4.3.5	Manually removing an existing boot entry	23
4.3.6	Removing all boot entries corresponding to a specified kernel version	23
5	Requirements	23
5.1	Boot Manager requirements	23
5.1.1	Ability to create BLS compliant boot loader entries	23
5.1.2	Ability to remove BLS boot loader entries	24
5.1.3	Ability to list the available BLS boot loader entries and their properties	24
5.1.4	Ability to display and modify the selected default boot entry	24
5.2	File system substitution requirements	24
5.2.1	Accept file system substitutions on the kernel command line	24
5.2.2	Perform substitutions before general system initialisation	24
5.2.3	Support clean shut-down of substituted mount points	24
5.3	Snapshot Manager requirements	25
5.3.1	Allow creation and removal of snapshots using LVM2, BTRFS or Stratis.	25
5.3.2	Automatically create boot configuration for snapshots of the root file system	25
5.3.3	Support expiry of snapshots by user specified policy	25
5.3.4	Simultaneous creation of snapshots of multiple volumes	26
5.3.5	Snapshot file operations	26
6	Proposals	26
6.1	Make snapshots accessible from the boot menu	26
6.2	Develop an integrated snapshot management tool	26
6.3	introduce a mechanism for boot time file system substitution.	27
6.4	Create multiple snapshots simultaneously as a coherent set	27
6.5	Future extensions	27
6.5.1	Provide policy controlled tools for snapshot life cycle management	27
6.5.2	Implement familiar file operations for inactive snapshots	27
III	Implementation	27
7	Introduction	28
8	Development languages	28

9	Software Components	29
9.1	Boot Manager	29
9.2	File System Substitution	29
9.2.1	Direct modification of <code>fstab</code>	30
9.2.2	Substituting <code>fstab</code> via bind mounts	30
9.2.3	Substituting individual mounts via bind mounts	31
9.3	Snapshot Manager	31
10	Library Interfaces	31

About this document

This document is written using LyX 2.2.2.

Please contact the development team using the dm-devel <dm-devel@redhat.com> mailing list with any questions, corrections or suggestions.

Executive Summary

Existing boot loader management tools are not well adapted to the problem of supporting multiple, bootable snapshots of the root file system, and have no support for managing snapshots of auxiliary file system mounts. We propose the creation of a new *Boot Manager* tool to address the problem of maintaining boot loader entries corresponding to snapshots of the system state, and a high-level *Snapshot Manager* to both automate and abstract the low-level detail of snapshot creation and boot entry management, and to provide additional value-added functionality such as snapshot meta data and life cycle management, and high-level operations on snapshot content.

Part I

Background

1 Problem Statement

Linux systems today have the ability to take a snapshot of file system state, including that of the root file system, using a number of different technologies:

- Classic device-mapper *copy-on-write* snapshots.
- Device-mapper thin-provisioning snapshots (*thin snapshots*).
- BTRFS *subvolume* snapshots.

A snapshot preserves the state of the file system at the time it is created. A *read-only* snapshot provides a persistent record of the file system state at the time of creation while *read-write* snapshots allow the history of the file systems to diverge over time, as different patterns of writes occur in the origin and snapshot volumes.

Currently the most widely used mechanism for creating snapshots using the Linux device-mapper is the LVM2 suite of volume management tools. Although it is possible to create both classical and thinly provisioned device-mapper snapshots manually using the `dmsetup` program, this type of configuration is not widely used in practice and is not further considered here.

An alternative implementation of device-mapper snapshots and other volume and file system management techniques for Linux, STRATIS[1], has been proposed and may in time provide another means to implement volume snapshots.

Snapshots may be used to effect a number of useful administrative tasks, for example:

1. Creation of a snapshot (read-only or read-write) for consistency checking, and to determine if action is needed on the live data.
2. Point-in-time snapshots for backup and recovery.
 - (a) Creation of a stable disk image to be backed up to other media.

- (b) Regular scheduled snapshots on a rolling or persistent basis for “live” backup, reference, and recovery.
- 3. Very frequent snapshots to catch brief events or quickly changing content.
- 4. Before-and-after snapshots of system state for testing and roll back.
- 5. Creation of temporary environments for testing updates and changes, to be either discarded or merged into the main volume at a later time.
- 6. Creation of stable, re-usable test configurations for quality assurance and continuous integration.
- 7. Preparation of standardised *base images* for further customisation.

When used in this manner snapshots avoid the expense involved in creating, or retrieving, a backup of the entire file system state or selected parts of it.

For example, the administrator may create one snapshot prior to applying software updates, and a second once the updates have been successfully applied. If problems emerge during testing or later, the prior system state may be immediately inspected via the snapshot, and rolled back to if necessary. Alternately, a single snapshot may be taken before the update operation and rolled back to in the event of failure.

Additionally, it is possible to interrogate snapshots to perform operations such as retrieving a file's content at an earlier moment in time, performing a “diff” operation between different versions of a file, or even to perform actions within a snapshot image, such as running an arbitrary command and adding or removing files or directories.

1.1 Booting snapshots

In many situations it is useful to be able to boot the system into the state represented by one or more snapshots of the root file system and other system volumes.

This allows inspection of the state of the system at the time the snapshot was created, as well as isolated testing of potentially destructive software updates, configuration changes, and other administrative actions:

- 1. Booting into a snapshot to inspect prior system state
 - (a) To reference an earlier configuration, or software update level while troubleshooting
 - (b) To retrieve data files, or information from past system state¹
- 2. Boot into a snapshot to test updates and other changes
 - (a) For distributions that reboot to apply updates
 - (b) To test updates before accepting (merging) into the main image
 - (c) Booting a system to safely test configuration changes, scripts, and other potentially destructive modifications on a snapshot of live data.
- 3. To recover from failed updates and other system changes
- 4. To create a new virtual or physical machine image from an existing image or template
- 5. Repeatable software testing of boot-time components from a consistent, reproducible image

¹For example data stored within encrypted volumes in a nested storage configuration, or information from a database that cannot be retrieved from the files offline.

It has been technically possible for a Linux system to boot into a previously created snapshot for a number of years, but currently it is not possible for users of common distributions to boot into an arbitrary snapshot image of the system without modifying system managed files, and providing specific kernel command line parameters to select and enable the snapshot volumes.

Although the options themselves are relatively straightforward for one-off use cases, they require knowledge of the snapshot name or identifier in advance, as well as correlation of the root device specification and snapshot specification, and no tooling currently exists to either detect suitable snapshots of the system, and create appropriate boot entries for them, or to manage those entries at the time of snapshot creation or removal.

The syntax required to boot a system using different snapshot implementations (BTRFS, LVM2, Stratis) necessarily differs due to differences in the underlying snapshot mechanisms: for example, all snapshots in LVM2 have a unique name used to identify them (qualified by the volume group where required), while snapshots in a BTRFS volume are identified by a unique numeric identifier

1.1.1 Booting snapshots of LVM2 volumes

To boot a system that uses the dracut[3] tool to generate `initramfs` images using LVM2 snapshots, the user must specify the root device to be used via the `root=` kernel command line parameter, and the snapshot logical volume to be activated via the `rd.lvm.lv=` dracut parameter.

For example:

```
root=/dev/mapper/vg00-lvol0 rd.lvm.lv=vg00/lvol0
```

Note that current syntax requires that the device be specified twice: firstly as a device node or symbolic link path in `"/dev"` to set the system root device, and then a second time as a direction to early user space to activate the appropriate device (this time using LVM2 "VG/LV" notation).

A bug currently exists in dracut (dracut-044) that prevents this syntax from activating LVM2 thin snapshots: this is because these snapshots have the *activation skip*² flag set by default. A pull request has been submitted to make this behaviour consistent with classic LVM2 snapshots when the `rd.lvm.lv` syntax is used (since this requests activation of a specific, named logical volume)[4].

1.1.2 Booting snapshots of BTRFS volumes

Similarly for BTRFS, the root device is specified as before, and the snapshot's subvolumeid is given using the `rootflags=` command line parameter:

```
root=/dev/sda2 rootflags=subvolid=262
```

The identifier passed for `subvolid` is the BTRFS subvolume identifier: an opaque, numeric identifier assigned by the file system that uniquely identifies a particular subvolume within a larger BTRFS volume.

BTRFS snapshot volumes can also be specified using the `subvol` path, a path in the volume namespace that resolves to a snapshot subvolume:

```
root=/dev/sda2 rootflags=subvol=/snapshots/20170528-1
```

When either flag is present the value overrides the superblock *default subvolume* setting, forcing the specified subvolume to be mounted instead.

One important consideration in booting snapshots of BTRFS volumes is that, by default, these snapshots are created *read-only*. A snapshot subvolume may be made writable using the `btrfs-property` [5] command, or a new writable subvolume may be created from the snapshot.

²The ability to skip activation of certain logical volumes (marked with the *activation skip* flag) was introduced in LVM2.02.99.

1.2 Core Goals

A number of goals and concrete proposals for the current work are suggested. Core aims are labelled separately from other features that are of additional benefit, but do not form a necessary part of the basic snapshot booting proposal.

It is expected that the core and additional goals can be delivered as a number of discrete milestones over a time-scale of one to two years. This is an aggressive and ambitious schedule and progress towards milestones will need to be carefully tracked to ensure delivery of core functionality on time.

A minor prerequisite to this work is to document the usage of the existing options that enable booting of LVM2 snapshots using dracut that the tooling will build upon.

1.2.1 Goal: Enable easy selection of bootable snapshots

Although it is possible to boot a Linux system into various snapshot configurations, few administrators are aware of, or make use of this functionality since it is poorly documented and relies on a detailed knowledge of kernel command line parameters, system devices, and VFS configuration.

With appropriate tools these powerful features would be accessible to a much wider audience of system administrators, users and developers, as well as to groups developing projects using Linux as a base operating system.

1.2.2 Goal: Automate the management of boot entries for snapshots

Once tooling has been established to allow bootloader entries for snapshots to be created and managed, it is possible to further simplify the administrator's task by automating the process of adding and removing the required bootloader configuration when snapshots are created or removed.

This may be done either by integrating management of boot configuration with the various tools responsible for snapshot management (`btrfs(8)`, `lvcreate(8)`, `lvremove(8)` etc.), or by creating a new tool that will perform the snapshot operation itself, using the appropriate tool for the type of volume being operated upon, before calling a tool to update bootloader configuration.

1.3 Additional Goals

Additional goals, beyond the core functionality required to boot into a snapshot of the root file system, are presented in this section.

These items are not essential to the aim of allowing a user to boot into a selected snapshot of the root file system but provide considerable added value, allowing the user to more precisely recreate system state by including auxiliary file systems in the snapshot boot configuration.

1.3.1 Goal: allow substitution of selected VFS mounts for snapshot versions

In order to assemble a coherent view of the system corresponding to an earlier point in time, it is necessary to be able to substitute other VFS mounts with their snapshot equivalents during boot.

Addressing this goal exceeds the capabilities of Snapper and brings the solution closer to feature parity with ORACLE Solaris/ZFS in terms of the ability to quickly recover an earlier system state, including auxiliary file system content.

1.3.2 Goal: Simplify creation of point-in-time snapshots of multiple file systems

Currently no mechanism exists to allow an administrator to simultaneously snapshot multiple devices or file system mount points, and to combine these snapshots together into a coherent view of prior system state.

It should be possible for the administrator to specify mount points to be included or excluded from the configuration (for example to allow sharing of data between snapshot instances, or to allow logs to be written to a common location).

This is necessary in order to be able to boot the system into an actual prior state, as recorded in the snapshots. This is the case because as well as application state, modifications to the root file system may necessitate changes to data hosted on other volumes that together comprise the overall operating environment of the system (for example: an update to a database package installed to the root file system, that then requires changes to the format of files stored in other volumes, or an update of a system to some new internal baseline image, with corresponding changes to data deployed in other system volumes).

1.4 Future extensions

Once the snapshot boot mechanism and snapshot management tools are established it becomes possible to deliver additional snapshot driven features beyond the basic capacity to boot the system into a prior state: although unrelated to the boot capability, these represent potentially significant added functionality and value.

Further analysis and development of these features will take place in a separate document.

1.4.1 Goal: Simplify management of large numbers of snapshots

When regular snapshots of the system are taken, or when snapshots are preserved over a long period of time, large numbers of snapshot volumes may accumulate on the system. This is a problem for several reasons: large numbers of entries clutter system menus and tool output, making selections more difficult to navigate while increasing the risk of error, as well as lengthening the time taken for the user to make a correct selection,

These effects are especially acute in restricted terminal environments used for system recovery, such as `systemd`'s emergency mode or the shell spawned by `dracut` when an early user space error occurs.

Additionally, in the case of classic device-mapper snapshots, I/O performance degrades when many snapshots of the same origin exist due to the non-shared copy-on-write store design.

By not addressing these issues in generic features, the cost is pushed onto users of the snapshot facility, who will need to implement appropriate policies themselves, either manually or via automation.

1.4.2 Goal: Simplify file operations on inactive snapshot volumes

In addition to booting into snapshots to perform administrative tasks, it is useful in some instances to be able to inspect these volumes from the running system, and even to perform actions within them (running a command, accessing file content).

Normally this would involve the administrator activating the volume, determining a path where it can be temporarily mounted, and issuing a command to mount the device before carrying whatever task is needed, and then reversing these steps once the volume is no longer required.

Providing automated mechanisms to allow these operations avoids the need to disrupt the system by rebooting for simple tasks, and simplifies the execution of these tasks on the running system by automating snapshot activation and mount operations.

1.4.3 Goal: Automatic rollback in the event of a failed boot

If an attempt to boot into a snapshot configuration fails, it is desirable for the system to automatically boot into the prior configuration (or to some earlier "*known good*" state).

Since these events fall outside the normal execution of the operating system (they involve only the failed attempt to boot, the firmware and bootloader) this goal is entirely dependent on the boot loader in use providing a means to configure this fall back mechanism.

Providing the ability to automatically recover a failed boot increases the ease-of-use of the snapshot boot facility, especially in environments where recovery is otherwise awkward: headless systems, restricted bootloaders, and systems with firmware that does not fully initialise the input subsystem before passing control to the bootloader

2 Linux boot process management

A Linux operating system instance is loaded in multiple phases:

Firmware Platform-specific initialisation carried out by embedded software.

Bootloader Selection, loading, and initialisation of a kernel image and auxiliary images.

Kernel Kernel and driver initialisation, VFS initialisation, process initialisation.

Early Userspace Selection and mounting of the root file system.

Init Control is passed to `init` daemon: auxiliary file systems mounted, services started.

Aside from the first phase, which is implemented by hardware specific embedded software, these phases are carried out by components provided by the operating system (in environments which permit booting more than one operating system the administrator may choose to create separate bootloader environments for each, or to manage a system wide, shared instance: no single convention for managing this situation has yet been adopted by the community).

The earliest phases of the Linux boot process (from the end of firmware initialisation to the point that control is handed to the kernel) have long lacked standardisation of both implementation and configuration.

Each distribution tends to carry its own tweaks, hacks and management scripts[13][14] to augment the bootloaders provided by upstream projects, to the point that in some cases a distribution's bootloader may be considered an independent fork of the upstream project.

Since today there is very little cooperation between distributions on shared management of a single, system-wide “/boot”, administrators who choose to configure the system in this way are left with the task of ensuring that the various installations do not conflict, and that there is a single, coherent bootloader installation visible to firmware at boot time.

Bootloaders in use today include:

elilo Bootloader for IA-64 based Intel systems (legacy)

lilo Bootloader for x86/x86_64 based PC systems (legacy)

grub1 Bootloader for x86/x86_64 based PC systems (legacy)

grub2 Cross-platform bootloader (x86/x86_64, IA-64, ARM/AARCH64, POWERPC)

yaboot Bootloader for POWERPC systems (legacy).

zipl Bootloader for IBM z SYSTEMS

Current RHEL and Fedora releases (RED HAT ENTERPRISE LINUX 7, and FEDORA 16 onwards) use the grub2 boot loader on all supported architectures except for IBM SYSTEM z. The current releases of UBUNTU (UBUNTU 16.04) and OPENSUSE (the rolling TUMBLEWEED release and OPENSUSE LEAP 42.2) both use the grub2 boot loader by default (although do not support the *BootLoader Specification* extensions discussed in 2.4).

2.1 Early user space interaction

The `PLYMOUTH`[15] graphical boot system provides a mechanism to present a graphical or text mode progress indicator to the user during system boot. Plymouth has been used as the graphical boot agent in Red Hat distributions since `FEDORA 10` and `RED HAT ENTERPRISE LINUX 6`, and is currently used in `UBUNTU (16.04)` and `OPENSUSE (TUMBLEWEED / LEAP 42.2)`

Plymouth consists of a daemon, `plymouthd`, which manages the display and handles requests from clients, and a command line and library interface used to notify the daemon of state changes, and to request user interaction via the console.

In addition to the progress indication and logging services provided by plymouth, limited interaction with the user is permitted in order to obtain answers to simple question/response prompts, or to provide specific information (typically authentication or decryption passwords).

The interaction verbs supported by current versions (0.8.9) of plymouth are[16]:

ask-for-password Ask user for password

ask-question Ask user a question

display-message Display a message

hide-message Hide a message

watch-keystroke Become sensitive to a keystroke

ignore-keystroke Remove sensitivity to a keystroke

These verbs accept the following options:

Options for `ask-for-password` command:

- `--command=<string>` Command to send password to via standard input
- `--prompt=<string>` Message to display when asking for password
- `--number-of-tries=<integer>` Number of times to ask before giving up (requires `--command`)
- `--dont-pause-progress` Don't pause boot progress bar while asking

Options for `ask-question` command:

- `--command=<string>` Command to send the answer to via standard input
- `--prompt=<string>` Message to display when asking the question
- `--dont-pause-progress` Don't pause boot progress bar while asking

Options for `display-message` command:

- `--text=<string>` The message text

Options for `hide-message` command:

- `--text=<string>` The message text

Options for `watch-keystroke` command:

- `--command=<string>` Command to send keystroke to via standard input
- `--keys=<string>` Keys to become sensitive to

Options for `ignore-keystroke` command:

- `--keys=<string>` Keys to remove sensitivity to

While it is technically possible to envisage using these primitives to implement a snapshot selection mechanism during early user space, the available primitives are intentionally limited and would make construction of menus having numerous entries awkward and clumsy, and enforces a question-response type of interaction (without the ability to, for example, scroll through the available entries).

2.2 Boot loader entries

Although each bootloader has its own configuration mechanism and syntax, generally a *boot loader entry* may be considered a tuple of (*kernel*, *initramfs*, *arguments*, [device tree])³, or (*kernel version*, *arguments*), with the precise image file names being formed from a template and the *kernel version* value. For architectures that support the use of a *device tree* (notably ARM/AARCH64), there is an additional tuple member to specify this parameter.

Boot loaders typically present the available boot entries in menu form, allowing the user to select from the available entries or to accept the default. This is a well known mechanism for controlling the system boot process although the details and precise capabilities vary among the existing bootloader implementations.

2.3 Bootloader abstractions

Several attempts have been made to create higher level abstractions over the multiple bootloader configuration formats, with varying degrees of success. Typically each distribution, or family of distributions, has a favoured tool which is used by the operating system installer, update tools, and package manager installation scripts to add, remove, or modify bootloader configuration.

These tools are generally designed to accommodate automated installation and update use-cases, for example: adding a boot configuration for a new kernel package, or removing all boot configurations corresponding to a specified kernel package.

Two examples are considered in detail here: the RED HAT grubby package, and SUSE's perl-Bootloader modules.

2.3.1 Grubby

The grubby[17] project has been used in RED HAT distributions since the early 2000s. It consists of a C program, grubby[18], with a command line interface, and a shell script, new-kernel-pkg[13], intended to be called from packaging scripts.

The grubby program understands configuration formats for a number of bootloaders (grub1, grub2, lilo, elilo, yaboot, and zip1), and is able to add, remove, modify, and display information for bootloader entries including the currently selected default entry.

Given a kernel, initramfs image, command line arguments, and title, the command will create a new entry, optionally substituting missing elements from a template generated from existing entries.

The tool is also able to adjust the default entry, and to probe for the presence of some supported bootloaders.

The bootloader abstractions implemented by grubby attempt to hide details of the individual formats (for instance, different keywords used to specify the kernel or initial ramfs image), as well as distribution-specific details such as the precise path of the bootloader configuration file in the “/boot” file system.

The project TODO[19] file records an intent to create a library interface but no progress is apparent at the time of writing.

The new-kernel-pkg script provides an additional layer over the grubby command which adds the ability to create or remove an initramfs image and to run RPM post-transaction hooks stored in “/etc/kernel/postinst.d”.

2.3.2 perl-Bootloader

The SUSE perl-Bootloader packages and update-bootloader script provide very similar functionality to the RED HAT grubby tool: a set of perl modules are provided that can read and write the configuration formats of various bootloaders (lilo, elilo, grub1, grub2,

³Some environments, for example, the Xen paravirtualised hypervisor, which implements the multiboot standard[23], require additional boot tuple values to specify auxiliary images to be loaded by the bootloader.

grub2efi), as well as a script to perform the task of adding, removing and updating entries for a particular kernel.

The upstream project has removed support for all bootloaders with the exception of grub2: other formats have been deprecated and removed, and are only supported in legacy downstream distributions.

Unlike grubby, the perl-Bootloader modules treat grub2 used in EFI configurations as a separate configuration family but this is a relatively minor implementation detail.

Like new-kernel-pkg and grubby, update-bootloader supports templating of new entries based on values used for previous configurations. Unlike the Red Hat package update script, the update-bootloader tool does not support generating a new initial ramfs image and requires the path to a previously generated image to be provided.

2.4 BootLoader Specification

The *BootLoader Specification*[20](BLS) is an attempt by the FREEDESKTOP.ORG[21] project to define a common standard for a shared, system-wide “/boot” file system.

The specification discusses the motivation for better standardisation of boot entry configuration, and the limitations of both existing open solutions and the EFI standards.

The specification documents requirements for the “/boot” file system and its directory structure, as well as a format for drop-in configuration *snippets* that each correspond to a single bootable instance of an operating system.

Compliant implementations create and delete files in the drop-in directory in order to add and remove bootable menu entries from the configuration. This avoids the need to edit entries embedded in a larger configuration file and the complexities and uncertainties that this introduces (a considerable part of the complexity of both grubby and perl-Bootloader is the need to accommodate both manual and machine modifications to the configuration as well as ambiguous or variable aspects of each boot loader’s native configuration syntax).

The current version of the BLS has the following requirements:

- Boot file system located at \$BOOT
 - \$BOOT place holder resolved at installation time to a location meeting the requirements of the specification. Specific requirements depend on the type of firmware (BIOS or EFI) and disk label (MBR or GPT) in use, and whether or not a usable EFI System Partition (ESP) already exists.
- \$BOOT/loader
 - Root directory for all BLS configuration.
- \$BOOT/loader/entries
 - Drop-in directory for BLS configuration snippets.

The BLS further specifies that the \$BOOT file system must be a VFAT 16 or 32 file system. This requirement attempts to harmonise with aspects of the (U)EFI specification and the requirements of the ESP.

Currently, BLS also requires that the partition containing the \$BOOT file system have partition type code 0xEA on systems using the MBR partition scheme. This may be problematic since this value is not reserved for this use and is currently in use by at least one other partitioning tool[22] for a conflicting purpose.

Although the specification was written some years ago, and is implemented in systemd-boot and several RED HAT distributions, development appears to have stalled.

The specification is appealing for the current work in part because it directly aims to solve the problem of a shared, system-wide “/boot”, and to solve the difficulties in configuration and file conflicts that arise.

These problems are a direct superset of the problems facing a snapshot boot implementation: where the operating system instances considered by the BLS are independent distributions (or even operating systems), the instances considered in the case of snapshot boot are simply multiple, divergent copies of one operating system instance.

It is this convergence of problem and requirement that makes the BLS solution readily adaptable to snapshot boot needs: an early prototype of the boot manager data store was developed prior to the authors adopting the BLS and considerable agreement existed between the two, in terms of the information present, data layout and directory structure.

For this reason, it is proposed that the immediate priority for the Boot Manager be to generate BLS compliant boot configuration on systems where a compliant boot loader is available, and that the formats used for this information by the Boot Manager and BLS be treated as the authoritative store of this data. For platforms where no bootloader that implements BLS is available (for example, IBM Z SYSTEMS), an extension may be created to render Boot Manager or BLS data into the native configuration syntax.

The `imgbased` project[26] currently generates BLS boot entries for managed snapshot volumes.

2.4.1 BLS Implementations

Two implementations of the BootLoader Specification are currently known: the `SYSTEMD-BOOT` project (formerly *gummiboot*), and the `GRUB2` packages shipped in RED HAT distributions.

systemd-boot The `systemd-boot`[27] project is the reference implementation of the BootLoader Specification. It is a native EFI application that operates entirely on the *EFI System Partition* (ESP). Among other Linux targets it is commonly used to boot distributions on the RASPBERRY PI 3 and similar EFI-only machines.

Red Hat Grub2 An implementation of a subset of the published BootLoader Specification is present in the `grub2` packages shipped in recent RED HAT ENTERPRISE LINUX and FEDORA releases. The implementation does not extend to the full set of restrictions on the `$BOOT` volume as described in the standard (which would need support from the installer and other components), but does provide a usable mechanism that may be harnessed to create additional boot entries on these systems.

2.4.2 BLS configuration snippets

A configuration snippet is a UNIX-formatted plain text file in UTF-8 encoding containing key/value pairs that define the values necessary to boot the entry. The following keys are currently defined:

title A human-readable title string for this entry ("`Fedora 24 (Workstation Edition)`").

version A human-readable version string for this entry ("`4.10.8-100.fc24.x86_64`").

machine-id contains the machine ID of the installation ("`611f38f...`").

linux the bootable kernel image to be loaded ("`vmlinux-4.10.8-100.fc24.x86_64`").

initrd the initial ram file system image to be loaded ("`initramfs-4.10.8-100.fc24.x86_64.img`").

options the kernel command line arguments to be passed ("`root=/dev/mapper/vg00-lvol0 rd.lvm.lv=vg00/lvol0`").

efi an arbitrary EFI program to be loaded (a path relative to `$BOOT`) ("`/System/Library/CoreServices/boot.efi`").

devicetree a binary device tree archive to be loaded on platforms that use DTB files ("`tegra20-paz00.dtb`").

Each configuration snippet must contain at least one linux key or one efi key, and must be stored in a separate file in "\$BOOT/loader/entries", named according to the scheme "<machine-id>-<uname -r>.conf".

For example:

```
6a9857a393724b7a981ebb5b8495b9ea-3.8.0-2.fc19.x86_64.conf
064d6dfabdea4552b3483779f63b656e-4.8.6-300.fc25.x86_64.conf
064d6dfabdea4552b3483779f63b656e-4.10.11-200.fc25.x86_64.conf
```

A BLS snippet for a Fedora 25 installation using kernel 4.10.11-200.fc25.x86_64 may contain:

```
title Fedora 25 (Workstation Edition)
version 4.10.11-200.fc25.x86_64
machine-id 064d6dfabdea4552b3483779f63b656e
options ro root=/dev/mapper/vg_f25-root rd.lvm.lv=vg_f25/root LANG=en_GB.UTF-8
linux /064d6dfabdea4552b3483779f63b656e/4.10.11-200.fc25.x86_64/vmlinuz
initrd /064d6dfabdea4552b3483779f63b656e/4.10.11-200.fc25.x86_64/initramfs
```

2.4.3 Proposed BLS enhancements (i)

Matthew Garrett has proposed a number of enhancements intended to address shortcomings in the BLS[28]. These include the following changes:

- Introduction of a formal name space in \$BOOT (" \$BOOT/org/freedesktop/.../")
- Support for chain loading entries via chainload key.
- Support for the Multiboot[23] standard via multiboot and module keys.
- Changes required MBR partition ID to 0xEF (EFI FAT-12/16/32).
- No requirement that the partition be VFAT.
- Support for kernel and ramfs images on other partitions.
- Remove the requirement that kernels be valid UEFI executables on UEFI systems.

These proposals have not yet been incorporated into the BLS. With the exception of the name space change (which although seemingly reasonable does not fully address the collision problem), and support for image paths outside of \$BOOT, these are all improvements to the specification in terms of its suitability for use in booting snapshot images.

2.4.4 Proposed BLS enhancements (ii)

We propose two minor extensions to the BootLoader Specification that will enhance the facility for the use of a snapshot boot implementation:

1. Addition of a simple, free text tagging mechanism to support optional grouping, filtering and sorting of entries.
2. Amendment of the suggested configuration file naming recommendation to include a free text discriminator field to allow snapshot entries to be given a unique name.

These changes (and bootloader support to make use of them) will improve the user experience when managing large numbers of bootable snapshot instances (see 1.4.1 on page 8).

These changes have not been formally proposed at this time. It is not clear whether the best route to having these changes included is to amend the published BootLoader Specification, or to first attempt to propose these features for the FEDORA grub2 package and upstream systemd-boot.

2.5 Other proposals

2.5.1 Directly booting LVM2 volumes and supporting “/boot” on LVM2 devices

It has been suggested by the LVM development team that the problem of booting directly to snapshot volumes (as well as other complex volume types) be addressed by enhancing the native grub2 support for booting LVM2 volumes and allowing both the target root file system, and the “/boot” file system to be hosted on LVM2 devices (or obviating the need for a separate “/boot” entirely, and allowing it to become a directory within the root file system).

Boot entries for snapshot volumes would then be identified either via the existing “grub2-mkconfig” discovery framework⁴, or by enhancing grub2 to directly detect and enumerate operating system instances located in LVM2 volumes.

Enhancing grub2 to directly comprehend LVM2 metadata, and to be able to place the “/boot” directory on LVM2 devices are existing long-term goals. Previous efforts in this area (by the grub2 developers) have been lacking in terms of safety guarantees and support for a comprehensive set of LVM2 features:

- Grub2 LVM2 runtime is developed independently of upstream LVM2
 - Duplicates internal constants (e.g. “GRUB_LVM_LABEL_SCAN_SECTORS”), assumptions, and implementation, requiring manual synchronisation
 - Independently implements and enforces metadata and device checks (e.g. whether a pvmove is in progress).
- Limited set of LVM2 features and targets supported.
 - No support for thinly provisioned volumes.
 - No support for cache volumes.
 - LVM2 RAID support may be out-of-sync with upstream due to separate development.
- Bugs in Grub2 LVM2 support may cause boot failures, or the bootloader, or kernel to crash, and in some cases could result in data corruption of LVM2 volumes or the devices that contain them.

The LVM development team has proposed the creation of a module, using native LVM2 code for discovery and metadata handling as a solution to these problems and to address the problem of grub2 and LVM2 device support becoming out-of-sync.

This is a viable long-term strategy but significant obstacles exist to realising it in the short term:

1. LVM2 user space run time must be ported to the grub2 run time
 - (a) Requires creation of a compatibility layer to allow LVM2 library code to be compiled and linked into both the usual ELF DSO used by applications on the running system, as well as an appropriate module for the grub2 run time.
 - (b) The grub2 environment is necessarily heavily restricted:
 - i. No standard I/O
 - ii. Limited memory management model
 - iii. No process management, no ioctl services
 - iv. Non-standard terminal interaction

⁴Using the DEBIAN “os-prober”[24] tool that is currently used to automatically detect Linux operating system instances installed to system devices

2. It is unclear how the build for the combined solution would work. Currently the modules linked into grub2 (either into the core image, or as separate loadable modules), are built from a common source tree. Since the proposed module would be developed and maintained in the LVM2 source tree from the existing user space code, a mechanism would need to be found to either generate source code for import to the grub2 tree, or to make the necessary build artefacts available from an LVM2 build to be used in order to build grub2.

These challenges can be addressed but involve considerable time-consuming development work and coordination with the maintainers of the Grub2 environment.

There is currently limited experience in developing and debugging software for the grub2 early boot environment in the LVM development team and many routine debugging aids (such as valgrind[25]) are not available in this context.

2.5.2 Adopting an existing bootloader abstraction

An alternative to using the *BootLoader specification* is to adopt one of the existing bootloader abstractions discussed in 2.3 on page 11.

In a similar manner as is proposed for the *Boot Manager* using BLS, these tools allow the creation, removal, and configuration of bootloader entries for a given tuple of boot values.

Although this would allow building on existing solutions, rather than developing a new *Boot Manager* component from scratch, this option was not further considered at this time for the following reasons:

- Choosing an abstraction commits the solution to a single distribution, or family of distributions
 - Otherwise, an additional abstraction layer would need to be developed to allow the solution to target multiple different bootloader management back-ends.
- The bootloader abstractions today represent the historical state of boot loader support for Linux: multiple divergent implementations and formats with some common aspects. Persisting with this model when virtually all supported architectures have now moved to a single, common bootloader implementation ties the project to unhelpful and unnecessary historical detail.

3 Existing solutions

Other operating systems and storage stacks (notably SOLARIS/ZFS, Linux/BTRFS, and Linux distributions shipping the ZFS file system) give the administrator the ability to boot a machine into a selected snapshot of the root file system (and in some cases to substitute other file system mounts with snapshot versions).

This section focuses on the implementations of most immediate relevance: ORACLE SOLARIS/ZFS, SUSE's SNAPPER using the BTRFS driver, as well as the proposed STRATIS STORAGE project.

3.1 Snapper

The SNAPPER[6] tool from SUSE includes both a command-line interface and a graphical application for creating, viewing and managing snapshots.

The tool also includes a mechanism to boot into a selected snapshot of a BTRFS file system. This feature relies on a BTRFS-specific superblock feature: the *default subvolumeid* field. By overwriting this value the tool controls the BTRFS subvolume that will be visible by default for any subsequent mount, including at system boot time.

This method avoids the need to modify the `fstab`, kernel command line parameters, or the `initramfs`, but has the disadvantage that recovery is made awkward in the case that the current default is for any reason unbootable. This behaviour is also considered objectionable by some users[7][8] since it assumes control of a file system feature normally available for the administrator's use.

There are also concerns with the fact that Snapper currently stores its snapshot meta data in a subdirectory within the volume that is being managed. This causes problems during recovery in some cases - a common workaround is to relocate the directory elsewhere and replace the original with a symbolic link[8].

- \neq Simple graphical user interface
- $+$ Ability to 'diff' and recover specific files (BTRFS only)
- $+$ Built-in snapshot expiration and clean-up (multiple policies)
- $-$ Advanced features tied to btrfs
- $-$ Requires administrator to give default subvolume control to Snapper
- $-$ Limited ability to create varied menu options and to control boot process at boot time
- $-$ Lacks ability to mount snapshots of non-root file systems at boot time.

3.2 Solaris ZFS and `beadm`

SOLARIS gained the ability to take snapshots of the root file system with the introduction of ZFS in SOLARIS 10. Since that time new management tools have been introduced to simplify the maintenance and use of multiple bootable configurations on a single system.

SOLARIS introduces the notion of a "*boot environment*": a bootable instance of the OS that encapsulates the file systems (or snapshots) to be made available as well as the kernel and ram disk images to be used.

An initial boot environment is created at installation time and further environments can be created by the administrator with the "`beadm`"[9] command. The command also provides the ability to list, modify and destroy environments and to change the active (default) boot environment.

The boot environment mechanism is strongly coupled to ZFS and uses ZFS pools, file system volumes, and snapshots to provide the file system mount points required by a boot environment.

An instance of a particular mountpoint (the origin volume, or a clone) is referred to as a *dataset*: each boot environment comprises exactly one *root dataset*, and one or more *additional datasets* (auxiliary volumes to be mounted below the root mountpoint). Datasets are associated with a given boot environment using the `beadm` tool. This feature allows a coherent set of snapshots to be combined into a single bootable environment.

The `beadm` command is also responsible for managing and updating bootloader menu entries for the managed boot environments, and for calling "`bootadm`"[10] to update a boot archive (equivalent to the use of an `initramfs` image for Linux systems). On x86 systems SOLARIS uses a modified build of the Grub1 bootloader.

Menu entries are generated automatically using the assigned "`name@description`" labels accepted by "`beadm create`"; labels are a free-text field which may include time and date strings, descriptions etc. to uniquely and meaningfully identify a given instance.

SOLARIS distinguishes between a "*snapshot*" of a boot environment, and a "*clone*": a *snapshot* is a read-only point-in-time image of a volume that is not bootable. To create a *clone*, that supports writes, and that can be used to boot the system, a previously created snapshot is used to create a new boot environment[11].

ZFS licensing precludes its inclusion in most distributions[12].

- \neq Closely integrated with file system and overall system administration suite.

- + Capability to associate auxiliary data volumes with boot configuration.
- + Simple interface to create, destroy, list, and modify boot environments.
- - Reliant upon SOLARIS' bootadm and Grub1 integration.
- - Directly tied to ZFS concepts and features.

3.3 Stratis Storage

The Stratis Storage project aims to provide ZFS/BTRFS-style volume management and file system features by integrating layers of existing technology: Linux's device-mapper subsystem, and the XFS file system. The project is relatively new but is under very active development and may represent a future snapshot management technology alongside the existing LVM2 and BTRFS stacks.

The Stratis Storage design document proposes the addition of boot capabilities in future versions[2] but this work is planned for later releases (Section 12.6 on page 24 proposes boot support for Stratis 4.0).

The Boot Manager may be extended to support Stratis file systems when it becomes possible to use these volumes as the system root file system.

Part II

Solution overview

4 User stories

Deborah is a system administrator responsible for managing a number of Red Hat Enterprise Linux servers.

Tom works in the DevOps team, developing and maintaining systems used by numerous teams for testing, staging, and production.

Derek is a manager with a technical background who runs Fedora on his desktop system.

Clive is a quality assurance engineer responsible for testing updates to the base operating system.

4.1 Snapshot Manager

4.1.1 Consistency checking

Deborah is concerned that a file system located on an LVM2 logical volume may contain data inconsistencies. She would like to carry out a check, using fsck and application specific tools but the application using the volume is currently in use and downtime will incur costs. Deborah decides to use a temporary snapshot to carry out the checks (Use case 1 on page 4)

1. Debora identifies the logical volume by name, and uses the Snapshot Manager to create a temporary snapshot.
2. Using the newly created snapshot devices, Deborah runs an fsck and checks the application data.
3. No problems are found and Deborah issues a Snapshot Manager command to delete the snapshot.

4.1.2 Point-in-time backups

Derek tells Deborah that the most recent scheduled backup for an important volume has failed. Data has not been backed up for 24 hours and Derek is concerned that an outage could result in significant data loss. He asks Deborah to create a backup as soon as possible.

Since the volume is currently in use, Deborah wants to take the backup with the least disruption possible. She decides to use a snapshot to create a consistent on-disk image for backup, without interfering with the live volume (Use case 2a on page 4).

1. Debora identifies the logical volume by name, and uses the Snapshot Manager to create a new snapshot
2. Deborah configures the backup tools to read data from the snapshot instead of the live volume and starts the backup.
3. Once the backup has completed successfully Deborah instructs the snapshot manager to delete the snapshot.

4.1.3 Frequent snapshots

Tom is working with a large batch application developed by another team which creates and removes a large number of files when running. While processing data the application sometimes logs an error indicating that files could not be created, but checks at this time indicate that plenty of space is available and that write permissions and other configuration are correct. To try to catch the activity that is causing the problem Tom decides to create frequent snapshots for inspection (Use case 3 on page 5).

1. Tom decides to use snapshots to try to investigate further.
2. He identifies the name of the affected logical volume and configures the Snapshot Manager.
3. He sets up a very frequent schedule of snapshots on the affected volume creating one snapshot every minute and allows the application to run.
4. Using the `df` and `diff` commands of the Snapshot Manager he inspects the snapshots as they are created.
5. In one snapshot the `df` command reports a sudden jump in file system usage.
6. By inspecting the content of the volume at this time, using the Snapshot Manager, Tom determines that a large number of temporary files have been created, filling the file system and preventing new file creation.
7. He reports this to the application team who investigate and provide a fix.

4.1.4 Before-and-after snapshots

Deborah is carrying out a system update using `dnf`. In case there is a problem during the update process she would like to have a backup of the system state before carrying on.

Additionally, once the update has been applied and tested she would like to create an image of the current state for reference, and in case there is a need to revert to it at a later time.

Because the backup system will take several hours to create a full backup of the system for each of these steps, Deborah decides to use snapshots instead (Use case 4 on page 5).

1. Deborah identifies the volumes on the system that contain the software to be updated.
2. She uses the Snapshot Manager to create a snapshot of these volumes at the same point in time.
3. Deborah starts the update process using the `dnf` command.

4. Once the backup process has completed Deborah instructs the Snapshot Manager to create a second snapshot for the system volumes.
5. Deborah inspects the system following the update, and identifies that some configuration files were not updated
6. Deborah reviews the `.rpmnew` files and merges them into the system configuration.
7. Deborah completes the check of the system and identifies no further problems.
8. Deborah deletes the first snapshot and creates a new snapshot to include the configuration changes.
9. Deborah keeps both of the remaining snapshots for future reference.

4.1.5 Testing updates (success)

Tom has been talking to Deborah about her recent update, and would like to update his RED HAT ENTERPRISE LINUX workstation too as new software updates are available.

Rather than take a snapshot of the system state to keep, as Deborah did, Tom decides to take a snapshot first, and apply the updates to the snapshot, before reviewing and either accepting the update (by *merging* the snapshot back into the origin volume), or, if there are problems, to reject it and delete the snapshot volume (use case 5 on page 5).

1. Tom takes a snapshot of the logical volume containing his root file system and creates a boot entry for it using the Snapshot Manager.
 - (a) Tom reboots the system and selects the newly created snapshot entry in the boot loader menu.
2. Once the system has booted, Tom logs in and begins the update process.
3. With the update complete, Tom begins to check the system to ensure all his regular applications are working correctly. He then reboots the system to test the newly updated kernel package, and waits for the boot to complete.
4. Once the system has booted Tom logs back in and completes his verification of the system.
5. On finding everything working as expected Tom instructs the Snapshot Manager to begin merging the snapshot into the main root file system and reboots the system.
6. Tom selects the entry for the system the snapshot was created from in the boot menu.
7. During activation the system starts the snapshot merge process.
8. Tom continues working on the updated system.

4.1.6 Testing updates (failure)

Derek sees a notification that updates are available for his FEDORA system. The update manager suggests rebooting to apply updates. Derek decides that he will use a snapshot to test the update before applying it to his system's root file system (Use case 5 on page 5 and boot use cases 2a and 2b on page 5).

1. Derek identifies the logical volume backing the root file system and uses the Snapshot Manager to create a new snapshot and corresponding boot configuration.
2. Derek reboots the system and selects the snapshot environment from the choices presented in the boot menu.

3. The software update begins and applies changes to the snapshot volume. Once the updates are complete the system continues to boot to the normal environment.
4. Derek logs in and begins checking the results of the software update. He soon finds that his favorite application, `LIBRE OFFICE`, is not working correctly following the update: as Derek needs to use the program, he decides to revert to the old configuration and re-attempt the update at a later time.
5. Derek reboots the system and selects the boot menu entry for the system the snapshot was created from.
6. Derek uses the Snapshot Manager to delete the temporary snapshot and continues with his work.

4.1.7 Creating stable testing images

Clive needs to test an update to `RED HAT ENTERPRISE LINUX` in a variety of standardised configurations. Each configuration represents a particular customisation of the operating system using the provided administrative tools. To save time on each configuration, Tom suggests that Clive manually installs a base system and then creates read-write snapshots for each separate configuration to be tested (Use case 7). The tests can then be run automatically against the snapshots with changes made as required.

1. Clive creates a new virtual machine using a thin-provisioned LVM2 logical volume
2. He boots the Anaconda installer using a kickstart configuration and installs the OS to the device
3. When the installation completes he shuts the virtual machine down.
4. Clive uses the Snapshot Manager to create several snapshots: one for each test scenario.
5. Clive boots each snapshot in turn, running a script to apply the necessary changes.
6. Once the images have been created, Clive uses the Snapshot Manager to create a new snapshot of each volume as a new baseline image.

4.2 File system substitution

The file system substitution tool is primarily expected to be configured by other software components: the Snapshot Manager and Boot Manager, as well as third party components that have a similar need to define modifications to the VFS configuration at boot time.

These, and additional manual use-cases will be further defined at a later time.

4.3 Boot Manager

4.3.1 Creating a boot entry for an existing snapshot (LVM2)

Derek carried out a major system upgrade a few weeks ago, using his distribution's bootable media. He later discovers that an application he previously used is no longer present, and the configuration files are not in `/etc`.

Prior to upgrading the system, Derek had created a snapshot of the root file system and had kept the volume group, and snapshot logical volume, when carrying out the update. He decides to create a boot entry for the snapshot to allow him to go back and inspect the application and its configuration so that he can install it on the updated system (use case 1 and requirement 5.1.1).

Using a bootable configuration will allow him to test the application directly without needing to take any special steps to mount the snapshot, and make it usable, on his current system (see 6.5.2).

1. Derek identifies the snapshot logical volume by name.
2. He uses the Boot Manager to create a new *boot entry* by specifying the *root logical volume* and a *label*.
3. The Boot Manager identifies the kernel versions present in the snapshot and verifies that they exist in “/boot”.
4. The Boot Manager generates the required *boot entry* keys via templates and writes a BLS configuration snippet.
5. Derek reboots the system and selects the newly-created *boot entry* from the menu.
6. The system boots into the snapshot allowing Derek to inspect the system and recover the application configuration.

4.3.2 Finding information on an existing entry

Derek would like to confirm the settings used for the snapshot *boot entry* that he has created. He decides to use the Boot Manager to list the available boot entries, including the entry that he has just created (requirement 5.1.3).

1. He opens a terminal and runs the Boot Manager to list the entries.
2. The Boot Manager accesses the BLS configuration path and collects the available entries for display.
3. The Boot Manager formats the results according to a default configuration, or options given on the command line.
4. Results are printed to the terminal in a human readable column formatted report.

4.3.3 Checking the currently selected defaults

Before rebooting, Derek would like to check the default boot entry on his system (requirement 5.1.4).

1. Derek opens a terminal and runs the Boot Manager to print the current default boot entry.
2. The Boot Manager interacts with the system bootloader to determine the default entry.
3. The Boot Manager formats the result and prints it to the terminal.

4.3.4 Overriding the selected entry for subsequent boots

Derek has to take a call while he is working on the recovery procedure. To avoid the need to pay attention to the system boot process, and to select the correct entry at the appropriate point, he decides to temporarily set the default boot entry to the snapshot entry he has created (requirement 5.1.4).

1. Derek opens a terminal window and runs the Boot Manager to check the current default.
2. He runs the Boot Manager a second time, this time to set the default entry to the label of the newly-created snapshot configuration.
3. The Boot Manager passes the new default to the system bootloader and confirms that it has been set.
4. Derek re-runs the Boot Manager to display the default, to double-check the setting.

4.3.5 Manually removing an existing boot entry

Derek has finished inspecting his previous installation and has successfully re-created the configuration on his new system. He decides to remove the boot entry for the snapshot, although he will keep the snapshot itself in case he needs to refer to it again (requirement 5.1.2).

1. Derek checks the logical volume name for the configuration he created earlier and uses the Boot Manager to verify that it is the correct entry, by checking the label that was used to create it.
2. Derek uses the Boot Manager to remove the configuration by specifying its label.
3. The Boot Manager verifies that the entry exists and, since Derek specified the label to be removed, deletes it without confirmation.

4.3.6 Removing all boot entries corresponding to a specified kernel version

Tom is applying updates to a server his group uses. The team has decided to adopt a new baseline kernel version supplied by the operating system vendor and Tom must remove the old kernel and its configuration from the system (requirement 5.1.2). Tom knows that a number of snapshot boot entries have been created that correspond to this version and he wishes to remove them all.

1. Tom identifies the kernel version that he wishes to remove.
2. Tom uses the Boot Manager to delete boot entries matching this kernel version.
3. Since multiple boot entries matching this specification exist the tool requests confirmation.
4. Tom confirms that the Boot Manager should remove these entries.
5. The entries are deleted from the BLS configuration path.

5 Requirements

5.1 Boot Manager requirements

The Boot Manager will be responsible for creating and deleting BLS snippets corresponding to bootable snapshot instances. It is the first new software deliverable for the proposed project, and forms the basis of much of the functionality to be delivered in the near term.

The tool is envisaged as a command line interface and library bindings to allow an administrator, or another software component to inspect and manage the set of BLS boot loader entries present on the system.

5.1.1 Ability to create BLS compliant boot loader entries

The tool must be able to accept a set of boot tuple values on the command line or via the library interface, and to create a corresponding BLS compliant boot loader entry ("BLS entry") capable of booting the system to that configuration.

On platforms that require the use of a *device-tree* the tool must accept and propagate this key in addition to the basic boot tuple values.

5.1.2 Ability to remove BLS boot loader entries

The tool must be able to remove a currently configured BLS entry when given appropriate selection criteria via the command line or library interface.

Possible criteria include the entry *Title*, *root=* value, or the unique per-entry identifier generated at the time an entry is created.

Where the given criteria correspond to more than one entry the tool will remove all matching entries (for example, to remove all entries associated with a specific kernel, or snapshot volume). Deletion of multiple entries simultaneously may require confirmation via a prompt or command line parameter.

5.1.3 Ability to list the available BLS boot loader entries and their properties

It must be possible to display all available BLS entries on the system, and to list relevant properties for them. It is desirable for the listing to also include a filtering mechanism, allowing entries to be selected using similar criteria to those used for deletion and other operations.

5.1.4 Ability to display and modify the selected default boot entry

In order to provide comprehensive management of the snapshot boot facility the tool should be able to both report the current default boot entry, and to modify the default stored by the bootloader (for bootloaders that support persistent configuration).

5.2 File system substitution requirements

The component responsible for file system substitution must accept a value on the kernel command line and perform the requested mount point substitutions during the boot process.

There are two points at which the opportunity to make these changes exists: (i) during early user space processing (initial ramfs), after the selection and mount of the root device, but before the pivot of the system root, and (ii) after the pivot, but before `systemd` reaches the `local-fs` target.

To substitute file system mounts during boot it is necessary to either modify the visible copy of `/etc/fstab`, or to modify the system mount state after local file systems have been mounted. Modifying the `fstab` has the advantage that tools such as `mount(8)` which operate on this file will behave as expected.

5.2.1 Accept file system substitutions on the kernel command line

The file system substitution mechanism must accept a specification on the kernel command line.

This maximises the flexibility of the feature, and allows experienced users to create or alter this configuration on-the-fly using the boot loader's native editing facilities.

5.2.2 Perform substitutions before general system initialisation

All file system substitutions must be fixed early on in the system boot process, in particular, before any services or other long-running processes that might otherwise use the substituted mount points have been started.

5.2.3 Support clean shut-down of substituted mount points

In addition to creating the required configuration during system boot, the file system substitution component must correctly reverse this configuration during shut-down, unmounting file systems and releasing the corresponding block devices at the appropriate points in the system shut-down sequence.

5.3 Snapshot Manager requirements

The snapshot manager provides a higher level mechanism for creating, removing, and managing snapshots and their associated boot configuration and meta data. As well as providing a simplified interface to administrators, and combining multiple steps into a single operation, the manager provides an abstraction over the various low-level snapshot mechanisms and creates an integration point which may be used to introduce further features and services.

5.3.1 Allow creation and removal of snapshots using LVM2, BTRFS or Stratis.

The tool must allow the user to specify an existing device, logical volume name, or file system mount point to be used to create a new snapshot. The user should be able to specify a distinguishing label and other properties for the snapshot, and to override existing defaults and inherited settings where applicable.

The tool must allow an already existing snapshot to be removed by providing a unique reference to the snapshot (the snapshot name, or device name where applicable, or some other unique identifier either created by the underlying snapshot mechanism or generated by the Snapshot Manager).

5.3.2 Automatically create boot configuration for snapshots of the root file system

The Snapshot Manager must be able to automatically create an appropriate boot entry for new snapshots of the root file system; required values that are not specified by the user should be obtained from either system templates or existing configuration as appropriate.

When a snapshot is removed using the Snapshot Manager tool, the manager must remove any corresponding boot entries that were previously created.

5.3.3 Support expiry of snapshots by user specified policy

The Snapshot Manager should allow the configuration of a policy controlling snapshot expiry, either on a system wide basis, or a per-snapshot basis.

Possible expiry policies include:

- Snapshot count
 - Snapshots are expired when the number of snapshots exceeds a user-defined threshold. Snapshots are removed in least-recently-created order until the limit is reached.
- Snapshot age
 - Snapshots are expired when they reach a threshold age specified by the user. Snapshots are removed in least-recently-created order until the limit is reached.
- Preservation tags and priority
 - Tags are applied to snapshots to indicate that they are to be preserved, or to declare properties allowing prioritisation.
 - The presence of a preservation flag exempts a snapshot from any other expiry policy.
 - The presence of priority flags may be used to select among possible candidates for deletion.
 - * For example, a system update may take precedence over a small, regularly scheduled snapshot.
- Snapshot space usage
 - Snapshots are expired when a threshold of space consumption in the snapshot pool or host file system is exceeded. Snapshots are removed in least-recently-created order until the limit is reached.

5.3.4 Simultaneous creation of snapshots of multiple volumes

It should be possible for a user to specify multiple volumes (one of which may be the root file system) for which snapshots will be created simultaneously.

When creating a boot configuration for a multi-volume snapshot, the tool must be able to automatically create an appropriate file system substitution configuration, and to add it to the boot configuration for the snapshot.

5.3.5 Snapshot file operations

It is desirable to be able to perform additional high level operations on snapshot volumes as proposed in 6.5.2 on the next page. This functionality requires that the Snapshot Manager be either able to mount snapshot volumes as needed, or otherwise be able to inspect content within the image contained in the snapshot (for example using the `libguestfs`[29] library). In the case that it is necessary to preserve superblock time stamps while performing mounts of snapshot volumes, the Snapshot Manager may choose to create new, temporary read-only or read-write snapshots in order to satisfy this requirement.

6 Proposals

6.1 Make snapshots accessible from the boot menu

The bootloader menu provides a familiar mechanism for selecting among the available bootable environments on the system. By adding the ability to automatically create and manage boot entries for snapshot volumes, without the need for very detailed knowledge of either boot loader, kernel command line, or system configuration, the feature is made accessible to the widest audience of system administrators and users.

Using well-known technology such as the `grub` menu maximises the administrator's familiarity with the new feature and enables the use of existing skills to further customise the configuration (for example, interactive edits to the bootloader entry).

By creating tooling to exploit existing features of the operating system to enable easy generation of bootloader configuration for snapshots created using any technology (LVM2, Stratis or BTRFS) the need to create additional, LVM2-specific, tooling is avoided and the maintenance burden is spread over a greater number of components.

6.2 Develop an integrated snapshot management tool

Since there are multiple snapshot mechanisms (LVM2, BTRFS and potentially Stratis), and since adding support for updating boot manager configuration would require changes to the tools for each of these mechanisms, it is proposed that a new, higher-level snapshot management tool be created to perform the task of creating and removing snapshots of the required type, and updating the boot configuration as appropriate.

As well as avoiding the need to modify the tools of every snapshot implementation this creates a consistent and uniform experience for users, independent of the particular snapshot technology in use. This benefits users migrating from one technology to another (for example, from a distribution using BTRFS for the root file system to one using LVM2, or a move from classic LVM2 snapshots to LVM2 using thin provisioning), allows the creation of consolidated documentation and other learning materials, and simplifies the extension of the feature to additional back-end mechanisms (for example, STRATIS).

This addition of a new component for snapshot management creates an integration point for further enhancements, for example adding policies to control various aspects of snapshot maintenance, and interfaces to provide monitoring and statistical information on snapshot usage.

6.3 introduce a mechanism for boot time file system substitution.

A mechanism will be created to allow a list of file system mount points to be given for substitution with specified snapshot versions, at the time the system is booted. The syntax must extend to LVM2, BTRFS and Stratis snapshot volumes, and should allow VFS mount options to be inherited from the existing configuration for the mount point under substitution, or specified at the time of substitution.

6.4 Create multiple snapshots simultaneously as a coherent set

The Snapshot Manager tool will allow the user to specify multiple volumes for which snapshots will be created simultaneously, and will create a corresponding boot entry that includes the necessary file system substitution if the root file system is included.

6.5 Future extensions

Extensions beyond the basic ability to boot into snapshots of the root file system and auxiliary file systems will be explored in detail in a separate document: key features of these proposals are documented here for reference.

6.5.1 Provide policy controlled tools for snapshot life cycle management

Once a tool exists to create and remove system snapshots and their corresponding boot configuration, it becomes a relatively simple step to apply policies to automate some aspects of that management, for example creating snapshots to a particular schedule, around certain events, or expiring snapshots by age, count or some other applicable mechanism.

By providing hooks that the administrator, or other applications can use to modify policy actions the flexibility of these mechanisms can be increased.

6.5.2 Implement familiar file operations for inactive snapshots

A set of operations that can be applied to snapshot volumes is proposed. These operations are based on familiar UNIX command idioms and provide a means to access and inspect snapshot content from the system.

df report snapshot disk space usage.

du estimate snapshot file space usage.

cp Copy a file between the snapshot and the file system.

cat Concatenate snapshot file content and print to standard output.

diff compare file content between the snapshot and file system line-by-line.

exec Execute the specified command in the snapshot name space.

Although each of these operations may be implemented manually by a sufficiently knowledgeable administrator, providing a simple command interface to perform these complex, multi-step operations greatly eases their use and makes this powerful functionality available to a larger audience.

Part III

Implementation

7 Introduction

Initial efforts will focus on the x86_64 platform and current RED HAT ENTERPRISE LINUX 7 and FEDORA releases, since these include support for BLS.

It is expected that additional architectures and distributions can be addressed with relatively little effort at a later time⁵.

A number of steps can be taken to solve the problems discussed in this document at this time; these may be implemented in an iterative manner, building on the results of each successive unit of work (with work proceeding in parallel on independent components where possible).

1. Document use of `root=` and `rd.lvm.lv=` to boot from snapshot volumes
 - (a) Address dracut support for thin-provisioned snapshots[4].
2. Provide tooling to create and manage instances of bootable snapshots, and to integrate these with the existing bootloader menu system or boot-time display manager.
3. Enable additional auxiliary file system snapshots to be associated with a bootable environment, to be automatically mounted in place of the origin file systems at boot time.
4. Provide a higher level snapshot management tool that handles the low-level aspects of snapshot creation and removal, as well as the creation and removal of corresponding bootloader entries.
5. Provide features to enable the management of snapshot life cycles, expiring and removing snapshots over time according to administrator controlled policy.
6. Allow snapshots of multiple volumes to be taken simultaneously, permitting a coherent snapshot of system state to be created.
7. Allow familiar operations on files and file systems contained within snapshots to be carried out from the running system.

8 Development languages

As each component of the current proposal is an entirely new software element there is considerable freedom in selecting the language used for the implementation.

Where some components must execute in restricted environments (early user space, `pre-basic.target`) there may be restrictions on both specific languages and the types of languages (e.g. compiled vs. interpreted) that can easily be supported.

For interpreted languages, PYTHON has the advantage of widespread adoption in RED HAT distributions, and a large and active development community. Alternatives include PERL and RUBY but these have a smaller systems programming footprint in today's distributions.

The GO and RUST languages[31] [32] have gained popularity in recent years for systems programming and infrastructure. They are high-level, compiled languages heavily influenced by C and related languages, with the addition of features such as garbage collection, memory safety, type inference (RUST) and built in concurrency primitives[33]. While the popularity of RUST and GO seems set to increase they are still relatively young languages, and questions remain over

⁵In particular, the grub2 bootloader is not currently used for IBM Z SYSTEMS running RED HAT operating systems: it may be necessary to render Boot Manager generated BLS snippets into the native zip1 configuration format.

long-term maintenance and support. In addition there is limited expertise in either language in the device-mapper and LVM2 developer community.

While C imposes additional burdens on development it is well-proven and the basis for much of the software in the distribution. Since the components that must be written in a compiled language are expected to be relatively small it seems to be the natural choice for these parts.

9 Software Components

The proposed implementation involves creating three new software components:

1. Boot Manager
 - (a) A command line tool to manage snapshot boot entries
 - (b) Fixes to dracut required to support thin snapshots
2. File system substitution mechanism
 - (a) A software component to perform file system substitutions passed on the kernel command line.
 - (b) Integration with the Boot Manager to allow passing of a substitution specifications to boot entries.
3. Snapshot Manager
 - (a) A command line tool to create, remove, and manipulate snapshot volumes and associated boot configuration
 - (b) Support for creation of snapshots of multiple volumes simultaneously.
 - (c) Policies to manage expiry and deletion of snapshots, based on:
 - i. Snapshot count
 - ii. Snapshot age
 - iii. Preservation tags
 - iv. Space consumption
 - (d) Support for common file operations on snapshot data.

Initial development will focus on (1) and (2).

9.1 Boot Manager

The Boot Manager does not need to execute during early user space, or before the `init` daemon has fully initialised the system. The tools are expected to be used from one or more operating system installations present on the system, or a similar rescue environment or live image, and will have full access to the normal libraries and runtime environments those operating systems provide.

Some initial prototyping work has been carried out in PYTHON and this is a commonly used language for similar development projects in RED HAT ENTERPRISE LINUX and FEDORA: considerable expertise exists among these communities that can be exploited if required.

9.2 File System Substitution

The component responsible for substituting file system mounts according to a specification passed on the command line must run during early phases of system boot: either during early user space (initial ramfs), or during the `init` phase of boot, but before the system has reached the `local-fs` or `basic` targets. This means that local file systems may not be mounted, and some libraries and runtime environments may not be available.

This suggests a need for this component to be either so simple that it is guaranteed to run in all of these contexts (for example, a POSIX shell script), or to be written in a compiled language with strict control over library usage in order to produce a binary that can be executed during these phases of system initialisation.

A number of possible mechanisms exist to implement the actual substitution of specified file system mounts. Some testing and initial prototyping has been carried out using the bind mount mechanism (9.2.2) but this will need to be further refined and developed into a usable component.

9.2.1 Direct modification of `fstab`

It is possible to control the snapshot used by the system by directly modifying the system `fstab` before non-root file systems are mounted. While this is the simplest approach it has a number of disadvantages that make it unsuitable for the proposed work:

- Modifying file system content modifies the snapshot being booted
- The substitution needs to be manually torn down at system shut-down time.
 - A backup must be taken of the original content and restored during shut-down.
 - A crash, power failure, or other unexpected reboot would leave the system in an inconsistent state: booting into some default root file system with the snapshot auxiliary volumes from the previous boot. This has the potential to lead to confusion, data loss or corruption, and considerable user frustration.
- User changes made while inside the snapshot environment must also be backed up and restored or they will be lost when the environment is next initialised and the `fstab` rewritten.

9.2.2 Substituting `fstab` via bind mounts

One method for modifying the `fstab` (without needing to revert the changes during shut-down, making the technique safe in the event of a system crash or power failure) is to substitute a modified copy using a *bind mount*[30]. A bind mount remounts part of the file system hierarchy (including a path that resolves to a regular file) to another location in the name space.

For example:

```
# Bind '/proc' to '/tmp/proc'
# mkdir /tmp/proc
# mount --bind /proc/ /tmp/proc
# ls -d /proc/1
/tmp/proc/1
# umount /tmp/proc
```

Or using a file path as the bind target:

```
# Create a file named 'bar' and bind it to 'foo'
# echo bar > bar
# touch foo
# mount --bind bar foo
# cat foo
bar
```

This allows a copy of the system “`fstab`” to be taken in another location, to which substitutions are made according to a specification. The modified copy is then substituted for the original by binding to “`/etc/fstab`”. The modified copy will then remain visible in that name space for the duration of the mount (or until another mount to the same path occurs, or the system is halted). This has the benefit that the mount is torn down automatically, even in the event of an unclean shut-down.

9.2.3 Substituting individual mounts via bind mounts

It is also possible to use the bind mount mechanism to substitute individual mounts, either before or after, they are mounted from the location specified in the `fstab`. This has a number of disadvantages for the current proposal:

- Binding substitutions after the system has reached `local-fs.target` would leave the original file systems mounted (possibly with some open file descriptors belonging to long-running processes).
- Bind mounting a file system requires that it is already mounted somewhere in the name space: this would create additional steps to first mount, and then re-mount the substituted file systems.
- Tools that read the `fstab` may misbehave or give confusing output if the file does not appear to match the current mount state.

9.3 Snapshot Manager

The Snapshot Manager is currently the most distant, and least certain aspect of the proposal: many decisions can be deferred until other aspects of the work are complete, including the choice of implementation language. The ideal model may involve a small set of core components written in a compiled language, with more peripheral elements (for example, a graphical user interface) written in other higher-level languages as appropriate.

10 Library Interfaces

Both the Boot Manager and Snapshot Manager should export functionality via clearly defined programmatic interfaces, in addition to the user-oriented command line interface.

The file system substitution mechanism only offers a single, simple interface with substitution specifications passed on the kernel command line. No need is anticipated for a means to re-configure the facility at runtime since it is expected to be configured once, at boot time, and to remain in place until system shut-down.

References

- [1] Stratis Storage <https://stratis-storage.github.io/>
- [2] Stratis Design <https://stratis-storage.github.io/StratisSoftwareDesign.pdf>
- [3] Dracut https://dracut.wiki.kernel.org/index.php/Main_Page
- [4] Dracut PR#223 <https://github.com/dracutdevs/dracut/pull/223>
- [5] btrfs-property <https://btrfs.wiki.kernel.org/index.php/Mainpage/btrfs-property>
- [6] Snapper <http://snapper.io/>
- [7] Snapper Issue#178 <https://github.com/openSUSE/snapper/issues/178>
- [8] Snapper layout discussion <https://bbs.archlinux.org/viewtopic.php?id=194491>
- [9] beadm documentation https://docs.oracle.com/cd/E23824_01/html/E21801/index.html#scrolltoc
- [10] bootadm documentation https://docs.oracle.com/cd/E26505_01/html/E29492/gglaj.html

- [11] Creating and copying snapshots https://docs.oracle.com/cd/E23824_01/html/E21801/snapshot.html#scrolltoc
- [12] ZFS license compatibility <https://sfconservancy.org/blog/2016/feb/25/zfs-and-linux/>
- [13] new-kernel-pkg(8) <https://linux.die.net/man/8/new-kernel-pkg>
- [14] update-bootloader(1) <http://www.polarhome.com/service/man/?qf=update-bootloader&af=0&sf=0&of=0openSuSE&tf=2>
- [15] Plymouth <https://www.freedesktop.org/wiki/Software/Plymouth/>
- [16] plymouth(8) <https://linux.die.net/man/8/plymouth>
- [17] Grubby <https://github.com/rhinstaller/grubby>
- [18] grubby(8) <https://linux.die.net/man/8/grubby>
- [19] grubby-TODO <https://github.com/rhinstaller/grubby/blob/master/TODO>
- [20] BootLoaderSpec <https://www.freedesktop.org/wiki/Specifications/BootLoaderSpec/>
- [21] Freedesktop.org <https://www.freedesktop.org/wiki/>
- [22] Rufus <https://github.com/pbatard/rufus>
- [23] multiboot <https://www.gnu.org/software/grub/manual/multiboot/multiboot.html>
- [24] Debian os-prober <https://joeyp.name/code/os-prober/>
- [25] Valgrind instrumentation framework <http://valgrind.org/>
- [26] imgbased_ <https://github.com/fabiand/imgbased>
- [27] systemd-boot <https://www.freedesktop.org/wiki/Software/systemd/systemd-boot/>
- [28] BootLoaderSpec-mjg59 <https://www.freedesktop.org/wiki/MatthewGarrett/BootLoaderSpec/>
- [29] libguestfs <http://libguestfs.org/guestfs.3.html>
- [30] Bind and shared subtree support <https://github.com/torvalds/linux/blob/master/Documentation/filesystems/sharedsubtree.txt>
- [31] Go Language <https://golang.org/>
- [32] Rust Language <https://www.rust-lang.org/en-US/>
- [33] Go Description [https://en.wikipedia.org/wiki/Go_\(programming_language\)](https://en.wikipedia.org/wiki/Go_(programming_language))