Robustness

Contents

Introduction

This design identifies circumstances that, though undesired because of the risk of loss of functionality, cannot be completely avoided and provides suggestions for dealing with them in such a way that as little functionality as possible is lost.

Note that improving D-Bus' robustness is a topic that will be covered in a later stage in its own design document. About securing D-Bus services, please see the security design.

Requirements

Minimize loss of data and loss of functionality due to data corruption in these abnormal circumstances:

- Unexpected power loss
- Unexpected removal of storage devices
- Unexpected lack of disk space
- Physical damage to the media and other hardware errors

Minimize loss of functionality due to processes hogging these shared resources:

- CPU
- GPU
- I/O
- memory
- network queue
- D-Bus daemon

Approach

This section explains how to address the requirements in several specific cases, taking into account different data sets and circumstances.

Application data

This section contains recommendations about how to robustly deal with data generated by applications.

General guidelines

No software should assume that opening files will always succeed. Failure conditions should be dealt with and the process will either continue running with as little loss of functionality as possible, or will log a message and exit. Programs should do the same when writing data (the filesystem may be full, or any other mode of error might occur).

For example, if the browser application finds out at start up that the cookies file is corrupted, it should move the old file away (or just delete it) and run as usual other than past persistent cookies will have been lost. Or if there was an error when writing a new persistent cookie to disk, the browser would keep running with that cookie being transient (in memory only).

In order to reduce the effects of data corruption, regardless of the causes, it would make sense to store different data sets in separate files. So that if corruption happens in, for example, the browser cookie store, it would not affect unrelated functionality such as playlists.

For big data sets, Collabora recommends SQLite with either Write-Ahead Logging (WAL[1]) or roll-back journal[2]. For smaller data sets, a robust method is to write to a temporary file and rename it on top of the old one once finished. This method is called "atomic overwrite-by-rename" and is mostly used when editing a file in-place.

POSIX requires the atomicity of overwrite-by-rename[3]. Btrfs, Ext3 and Ext4 give atomic overwrite-by-rename guarantees, as well as atomic truncate guarantees. The FAT filesystem guarantees neither.

SQLite

For applications using SQLite for their storage, Collabora recommends using either WAL or the rollback journal so that transactions are committed atomically. In addition, filesystem-specific tuning would be done by configuring the SQLite system library for optimal performance.

WAL will be the best option in most cases, except when transactions will be very big (involving more than 100 MB) and when writes are very seldom, then the rollback journal would be preferred.

Collabora will run the TCL test harness[4] for SQLite in LAVA, to detect any issues in the specific configuration and software in the target platform. These include robustness tests that reproduce out-of-memory errors, input/output errors and abnormal termination (crashes or power loss).

Tracker

Tracker stores data in SQLite files, so the robustness considerations that apply to SQLite apply to Tracker as well. By default it uses WAL instead of the traditional rollback journal, which gives better performance for Tracker's workload with the same robustness guarantees.

[1] http://www.sqlite.org/draft/wal.html
[2] http://www.sqlite.org/draft/lockingv3.html#rollback
[3] http://pubs.opengroup.org/onlinepubs/009695399/functions/rename.html
[4] http://www.sqlite.org/testing.html#tcl

User settings

For configuration settings in general, Collabora recommends using the GSettings[5] API from GLib with the dconf[6] backend. When updating the database, dconf will write the whole new contents to a new file, then atomically renaming it on top of the old one.

For bigger pieces of data (individual settings whose data component exceeds 1 KB), Collabora recommends using plain files via a known-robust file-handling library (such as GKeyFile[7] from Glib, which is already a dependency) or SQLite.

Media

For media, the meta-data is stored in Tracker, with the actual data files in the /home filesystem and in attached removable devices.

If the Tracker database that contains the meta-data has been corrupted, it should be moved to the side (or deleted) and recreated again by indexing all available media files. To minimize the chances of corruption, refer to Tracker.

Software that reads the actual media files should assume media files may contain invalid data and ignore them without further loss of functionality. Corrupted media files should not be displayed in the UI.

Caches

All software that uses a cache file should be ready to find that the cache is unusable and cope with it without loss of functionality (temporary degradation of performance is obviously expected in this case though the mechanism by which the cache became corrupted will be treated by developers as a bug to fix).

For example, if during start-up the Folks caches are found to be unreadable, libfolks would remove the corrupted cache files and recreate them, taking a longer time to reply to queries. As the application using Folks would be executing the queries asynchronously, the UI would keep being functional while the query executes.

Examples of other components that use caches and that should cope with cache corruption are the browser and the email client.

Filesystems

The reliability with which data is stored depends on both the storage medium as well as the filesystem. In this section, we cover FAT32 and Btrfs. Ext4 is mentioned, as it is a popular default filesystem on many Linux distributions –

[5] http://developer.gnome.org/gio/stable/GSettings.html
[6] https://live.gnome.org/dconf
[7] https://developer.gnome.org/glib/2.37/glib-Key-value-file-parser.html

however it doesn't suit the needs of the rollback system – either for system rollbacks (See the System Update and Rollback Design) or for application rollbacks (See the Applications Design).

The FAT32 filesystem is not robust under abnormal circumstances since it was not made for devices which could be disconnected at any moment. In general, an approach where writes to the device are tightly controlled and restricted to small time-windows would help minimize the chances of corruption. See the *Media and Indexing* design for a detailed explanation of the issues and suggestions.

The Ext4 filesystem is quite robust under power failure by default. It can be made even more robust by mounting it under data=journal[8] mode, but at a large cost to performance.

Btrfs has been created on very robust principles, building upon the experience of Ext4. Some brief technical details are provided at the end of this document in BTRFS overview.

Filesystem options

Filesystems usually have parameters that can be tuned to suit specific workloads. Some of them affect performance as well as robustness; either by trading off between the two, or by taking advantage of specific hardware features available with the storage media.

- **FAT32** is a simple filesystem that does not have many filesystem options related to performance or robustness. Since we will not be creating any FAT32 partitions ourselves, only mount-time options are interesting for us. The recommended options are listed below:
  - sync, flush These filesystem options ensure that the kernel, as well as the filesystem, flush data to the partition as soon as possible. This greatly reduces the chances of data loss or filesystem corruption when USB drives are yanked out by the user.
  - ro (read only) It is recommended that FAT32 partitions be mounted read-only to avoid filesystem corruption, and other related problems as detailed in the "*Media and Indexing Design"* in the section "*Indexing database on removable device"*.
- **Btrfs** is relatively new, and so does not have many options relevant to our needs of enhancing reliability on eMMC storage media. The available options are listed below.
  - *Mount-time options:*
    * commit=number (default: 30) Set the interval of periodic commit. This option is recent (since kernel 3.12)[9].

[8] http://kernel.org/doc/Documentation/filesystems/ext4.txt
[9] https://btrfs.wiki.kernel.org/index.php/Mount_options

    - ssd This option enables SSD-specific optimizations and disables some optimisations specifically for rotating media. This option is enabled automatically on non-rotating storage.
    - Recovery (default: off) This option can be used to attempt recovery of a corrupted filesystem (See Repair and recovery).
  - *Filesystem creation options:*
    - -s sector-size This is the size of the filesystem blocks used for allocations. Ideally, this should be the same size as the block size for the storage medium.
    - -M

      This sets BTRFS to use "mixed block groups" - a mode that stores data and metadata chunks together on disk for more efficient space utilization for small filesystems – but incurs a performance penalty on large ones. This option is not mature and will be evaluated in the future.

The System Updates and Rollback Design describes the partition layout for Apertis. Not all the partitions have the same requirements, so both the FAT32 and BTRFS filesystems are used. The partitions are configured as:

- **Factory Recovery** – This partition is never mounted read-write and must be readable by the boot loader. Currently the boot loader for Apertis – U-boot – does not support BTRFS. While patches exist to add that functionality, they have not yet seen widespread testing. FAT32 will likely be the filesystem chosen for the factory recovery image.
- **Minimal Boot partitions** – These partitions must also be readable by the boot loader, and are currently FAT32. They are not normally mounted at run-time, instead they are created, mounted, and populated by the system update software once – and only ever accessed by the boot loader afterwards. They will be mounted with the "sync" and "flush" flags.
- **System** - Since BTRFS provides an excellent snapshot mechanism to assist system rollbacks (See Cheap, fast, and atomic snapshots and rollback), this partition will be populated with a BTRFS filesystem created with the appropriate sector size for the storage device. It may be created with mixed block groups to save storage space if that option does not lead to instability. It will be mounted with the ssd option as well as read-only. During a system update a single subvolume of the system subvolume will be mounted read-write. The repair mount option will never be attempted on the system partition, instead rollbacks or factory recovery will be used to avoid potentially putting the system into an unknown state.
- **General Storage** – This partition shares similar requirements to the system partition. It will be BTRFS, created with an appropriate sector size and possibly mixed block groups. It will be mounted with the ssd

option. This is the only built-in non-volatile storage that will always be mounted read-write. In the case of a damaged filesystem, repair may be attempted on this partition.

Additionally, there are 2 partitions for raw status flag data that do not use filesystems at all. See the System Updates and Rollback Design for more details.

Checksumming

Checksumming is used for detecting filesystem corruption due to any reason. Different filesystems have different mechanisms for checksumming which give us coverage for various different causes of filesystem corruption. Each mechanism consumes I/O and CPU resources, and that must be weighed against the advantages that it gives us.

It is important to note that checksumming does not protect us against corruption or help us in fixing the root cause of the corruption; it only allows us to detect filesystem corruption when it happens. Hence, it is only useful as a warning sign and recovering from data corruption is beyond the scope of this feature.

- **FAT32** is a very old and simplistic filesystem, and it has no inbuilt facilities for checksumming.
- **Btrfs** maintains a *checksum tree* for all the blocks that it allocates and writes to. Hence, all file data and metadata is checksummed. This is the default behaviour and the current checksum algorithm uses few resources. This method of checksumming can detect all the ways in which corruption can occur to data on the filesystem. See Checksumming for more detail.
- **Ext4** maintains checksums for journal data only, no checksumming of file data takes place.

Alignment

The first piece of tuning that a filesystem on flash storage needs, is a proper mapping of the filesystem blocks to the page size of the erase blocks on the flash. This consists of two parts:

1. Ensuring that the filesystem and storage erase block sizes match using filesystem creation options.
2. Aligning the block allocations in the filesystem with the storage blocks by using the appropriate offsets while partitioning, or while creating the filesystem.

If either of these is not satisfied, each filesystem block write will trigger two or more flash block writes, and reduce the performance as well as reliability of the MMC card.

The storage erase block size can be read[10] from /proc/mtd or from U-Boot but the flash storage can report something different than the real numbers. Some sizes are available on the Linaro wiki[11]. Linaro-image-tools is now able[12] to generate images with a correct alignment.

Testing

Collabora will add tests to LAVA for testing how FAT32 and Btrfs behave on the i.MX6 under stress, as well as for tuning the above mentioned parameters for reliability and performance.

Root filesystem

The approach will be to mount as many parts of the root filesystem read-only as possible such that the only writes to it would be during updates. This would reduce the chances of catastrophic filesystem corruption in the event of power failure and invalid system file modification by bugs in system or application software. The only partition that is to be mounted writable is the user partition that will be mounted in /home. All the other writable parts of the / filesystem will be backed by tmpfs, located in RAM. We will avoid the lack of space problem by only storing small files in tmpfs or files which don't take space (lock files, socket files). Bigger files such as programs, libraries, configuration files will remain on disk and available read-only.

See the *System Updates and Rollback* design for detailed information about the robustness of the update process.

Other filesystems

The system should be able to function even if mounting one or more of the non-essential file systems fails. Even if the system is able to keep running, it would do so with reduced functionality, so some recovery action would need to be taken in order to regain the lost functionality. The system should try to recover automatically as far as possible. In the case of unrecoverable system failure, the user can be instructed at system boot to request technical assistance at a service shop.

Main storage

In case of power loss, the flash media can become corrupted due to how writes are performed. Apertis will be notified via a GPIO signal 100 milliseconds before power is completely lost, in order to give the flash controller time to commit to non-volatile media what is in its cache.

[10] http://processors.wiki.ti.com/index.php/Get_the_Flash_Erase_Block_Size
[11] https://wiki.linaro.org/WorkingGroups/KernelArchived/Projects/FlashCardSurvey
[12] https://bugs.launchpad.net/linaro-image-tools/bug/626907

Given the short time available and the general slowness of flash devices when writing, we recommend that the signal is handled in the kernel, because userspace will not have enough time to react (depending on the load and the scheduler, it could take from 10 ms to 100 ms for the signal to start being processed by a userspace process). A device driver should be written that, when the GPIO signal is received:

1. stops flushing dirty pages to the drive,
2. tells the flash controller to flush its caches to permanent storage, and
3. starts the shutdown sequence.

The device driver will start handling the signal 10-100 µs after the GPIO is activated. In spite of this, if the device has big caches and is slow to write, corruption of arbitrary data blocks can still happen.

In general, drive health data should be monitored so that the user can be notified about disk failures which require a garage visit for hardware replacement.

As no more dirty pages will be flushed to the storage device when the GPIO signal is received, the data in the page cache will be lost. To reduce the amount of data that could be lost, eMMC reliable writes can be used, and the page cache configuration can be tuned. But it has to be noted that use of reliable writes and reducing the amount of in-flight data is a trade-off against performance that can be quantified only on the final hardware configuration through direct experimentation.

Removable devices

External devices that can be removed at any moment are not reliable for writing of critical data. In addition to the problem of corruption of files being written, wear leveling by the controller might corrupt unrelated blocks which might even contain the directory table or the file allocation table, rendering the whole partition unusable.

The quality of external storage devices such as flash drives varies greatly, in some cases the device will unexpectedly stop responding to commands, or data will be lost. Applications that write to removable drives must be robust enough to be able to continue in the face of such errors with minimal loss of functionality.

As mentioned in Filesystems, the safest way to use removable drives is by restricting the processes that can write to the drive, and minimizing the time-window for the writes. For that to be practical, there should be a system service that is the only one allowed to write to removable devices and that would accept requests from applications, remount the device read-write, write the new contents, then remount read-only again.

Since, for interoperability reasons, the filesystem used in removable devices is

FAT32, in addition to the issues mentioned in this section, the robustness considerations that were explained earlier in Filesystems also apply.

Mitigating the effects of lack of disk space

In order to reduce the chances that the system will find itself in a situation where lack of disk space is problematic, it is recommended that available disk space is monitored and applications notified so they can react and modify their behavior accordingly. Applications may chose to delete unused files, delete or reduce cache files or purge old data from their databases.

The recommended mechanism for monitoring available disk space is for a daemon running in the user session to call *statvfs* (2) periodically on each mount point and notify applications with a D-Bus signal. Example code[13] can be found in the GNOME project, which uses a similar approach (polling every 60 seconds).

Additionally, so error messages can be stored also in low-space conditions, it is recommended that *journald* is configured to leave an amount of free space smaller than the reserved blocks of the filesystem that backs the log files. This way, applications will still be able to log messages after applications have consumed all the sace available to them.

In case applications cannot be trusted to properly delete non-essential files, a possibility would be for them to state in their manifest where such files will be stored, so the system can delete them when needed.

In order to make sure that malfunctioning applications cannot cause disruption by filling filesystems, it would be required that each application writes to a separate filesystem.

It may be worth noting that temporary directories should be emptied on reboot.

Resource management

The robustness goal of resource management is to prevent one or more applications from disrupting basic functionality due to excessive resource consumption. The basic mechanism for this is to allocate resources in such a way that applications cannot starve services in the base system. This is to be achieved firstly by changing the resource allocation policy to give higher priority to services, and secondly by limiting the maximum amount of resources that an application can consume at a time.

Resource limits are capable of helping ensure a process does not render the whole system unresponsive. However, some design decisions also play an important role here. If the user has no way to kill the process that became too slow or unresponsive, the user experience will suffer. The same goes for the

[13] http://git.gnome.org/browse/gnome-settings-daemon/tree/plugins/housekeeping/gsd-disk-space.c#n693

case in which an application gets stuck into a failing scenario, such as a web browser automatically loading pages that were open when the browser closed unexpectedly. For these reasons care must be exercised while designing the user interactions for both the system chrome and applications to be sure such cases are addressed.

If, despite throttling, some processes still impact the overall user experience negatively because of excessive resource usage, there is the option of identifying those processes and terminating them. Collabora recommends against this because it is very difficult to automatically distinguish between processes that use large amounts of resources due to malfunction or maliciousness and processes that use excessive resources for legitimate purposes. Killing the wrong process may free up resources but is likely to be perceived by the user as a severe defect in the overall user experience.

As a general recommendation, for optimal responsiveness, applications should not block the UI thread when calling anything that is not assured to return almost immediately, which includes all local or remote I/O operations. When the potential duration of an operation is a considerable portion of the commonly-considered maximum acceptable response time (100 ms), it should be done asynchronously. GLib contains asynchronous APIs[14] for I/O in its file[15] and streaming[16] classes.

CPU

To make sure that important processes have available CPU cycles even when malfunctioning or malicious applications monopolise the CPU, it is recommended to set task scheduler priorities according to the importance of processes. Systemd can do this for services by setting the CPUSchedulingPriority[17] property in the service unit file of the process. When the process described by the service unit file starts new processes, they stay in the same cgroups and they keep the same CPUSchedulingPriority.

At present (Q1 2014), systemd manages the user session on target images but not on the SDK. With the user session managed by systemd, the priorities of applications are no longer set by the application launcher using sched_setscheduler (2)[18].

If there are processes that need real-time capabilities, or that should have very low CPU access, the CPUSchedulingPolicy property can be used to change to the rr (real-time) or idle scheduling policies. Real-time access for a process should be carefully considered and tested because it can have a negative impact on the process and even the entire system.

[14] http://developer.gnome.org/gio/stable/async.html
[15] http://developer.gnome.org/gio/stable/file_ops.html
[16] http://developer.gnome.org/gio/stable/streaming.html
[17] http://0pointer.de/public/systemd-man/systemd.exec.html
[18] http://www.kernel.org/doc/man-pages/online/pages/man2/sched_setscheduler.2.html

For identifying processes that use an excessive amount of CPU, the cpuacct[19] cgroups controller can be used.

Though it is not recommended to automatically terminate local applications with excessive CPU usage, it makes sense for web pages. Web pages are not screened before they execute on the system, hence it is important to ensure that their ability to disrupt system functionality is minimised. For this, WebKit can detect when a block of JavaScript code has been executing for too long, pause it, and give the embedding application the possibility of canceling the execution of this block of code. Collabora has added API to WebKit-Clutter for this.

I/O

Similar to CPU usage, Collabora recommends giving priority to important processes when there is contention for I/O bandwidth. Collabora recommends that important services have a value for the property IOSchedulingPriority lower than 4 (the default). If, for any reason, some applications need priorities other than the default, the application launcher can use the ioprio_set[20] (2) syscall to change their priority. When the process described by the service unit file starts new processes, they stay in the same cgroups and they keep the same IOSchedulingPriority.

Memory

Collabora recommends putting a single limit on the amount of memory that the whole application set can allocate so a fair reserve is left for the base software. This limit should be just big enough so that the Apertis instance never reaches the "out of memory" (OOM[21]) condition at the system level. For example, if the total of memory available for processes is 1GB, there is no swap, and we know that the services in the base system should need a maximum of 300MB, then all applications should belong to a cgroup that is limited to 700MB of memory.

In specific cases, it may make sense to put a different limit on a specific application, but it can easily be counterproductive and cause a waste of memory.

Something else worth doing is to make sure that the OOM killer[22] selects applications for killing instead of system services. For this, the systemd property OOMScoreAdjust can be used to reduce the chances that a service will be killed. For applications, it is recommended that the application launcher sets its /proc/<pid>/oom_score_adj (see here[23]) to be higher than 0. The ideal value may vary depending upon the importance of each application.

[19] http://www.kernel.org/doc/Documentation/cgroups/cpuacct.txt
[20] http://www.kernel.org/doc/man-pages/online/pages/man2/ioprio_set.2.html
[21] http://en.wikipedia.org/wiki/Out_of_memory
[22] http://lwn.net/Articles/317814/
[23] http://www.kernel.org/doc/Documentation/filesystems/proc.txt

With the example setup mentioned before, the OOM killer will terminate the bulkiest application when one of these conditions are met:

- The total memory taken by applications all together is going to increase over 700MB.
- The total memory taken by all processes (services plus applications) is going to increase over 1GB.

To make better use of the available memory, it's recommended that applications listen to the cgroup notification memory.usage_in_bytes[24] and when it gets close to the limit for applications, start reducing the size of any caches they hold in main memory. It may be good to do this inside the SDK and provide applications with a glib signal that they can listen for.

Network queue

Processes would be classified into cgroup classes such as:

- Interactive (VoIP, internet radio)
- Semi-interactive (web pages, maps)
- Asynchronous (mail, app notifications, etc)
- Bulk (downloads, system updates)

Cgroup controllers are only used for classification of outgoing packets. NET-PRIO_CGROUP[25] and NET_CLS_CGROUP[26] would be used for setting the priority, and for classifying processes into cgroups. By thus tagging packets with the cgroup of applications and services, *tc*[27] can be used to set limits to the rate at which processes send packets (http://lartc.org/howto/).

Bandwidth rate-limiting would be required to ensure interactive streams do not get starved by lower priority streams.

There is little we can do about latency for applications like VoIP, since even when the bandwidth is sufficient, the bottlenecks are the hardware buffers, queues, and scheduling on various devices outside the control of our system. This is an open problem in networking, and a large part of it is related to Bufferbloat[28].

Note that there's no robustness issue that can be prevented by limiting the rate at which processes receive incoming packets.

---

[24] http://www.kernel.org/doc/Documentation/cgroups/memory.txt
[25] http://lwn.net/Articles/474695/
[26] http://docs.fedoraproject.org/en-US/Fedora/16/html/Resource_Management_Guide/sec-net_cls.html
[27] http://lartc.org/manpages/tc.txt
[28] http://en.wikipedia.org/wiki/Bufferbloat

GPU

As explained in the WebGL design, the GL_EXT_robustness[29] extension provides a mechanism by which the watchdog in the GL implementation can reset the GPU, invalidating all GL contexts and thus stopping all GPU activity.

Unfortunately, this only prevents denial of service (DoS) conditions caused by WebGL, because processes must opt-in to use this extension. Thus, applications may intentionally or unintentionally ignore the extension and continue monopolising the GPU. Within the web browser, scripts that use WebGL and take over the GPU will be interrupted and terminated by the browser.

If it runs its own GL implementation, then it could monitor GPU resource usage and reset those contexts that seem to be disrupting the rest of the system. It could notify processes via the GL_EXT_robustness extension and even terminate them if they ignore the context reset notifications.

Accounting

Besides setting limits on resources, cgroups also allows to retrieve resource usage metrics. As examples, for CPU usage the *cpuacct* cgroup controller contains the *usage*, *stat* and *usage_percpu* reports; the *memory* controller provides usage data in its *stat* report; the *blkio* controller has *throttle.io_serviced* and *throttle.io_service_bytes*.

USB undervoltage

In the case that the system momentarily isn't able to power connected USB devices such as MP3 players or smartphones due to voltage drops, the system will power off and on again these devices, so that the connection gets reestablished and the user experience gets affected as little as possible.

Risks

- FAT32 is fundamentally unreliable, specially on removable devices.
- Robustness of flash media varies greatly and the user may not be able to distinguish failures caused by the hardware from failures due to the software.
- Excessively-low resource limits for applications can lead to resource waste; excessively-high may be less effective in avoiding DoS. There may not exist a good middle point.
- Heuristics used to determine when to kill a process with excessive resource usage are not perfect and can cause major failure from the user point of

[29] http://www.khronos.org/registry/gles/extensions/EXT/EXT_robustness.txt

view.

- If Vivante does not implement GL_EXT_robustness properly, web pages could DoS the whole system.
- Bugs in the OpenGL implementation can lead to instability, data loss and privacy breaches that can be triggered from web pages.
- If the flash media loses power while a block is open for writing, it is possible that several random blocks elsewhere in the same drive will be corrupted. This can affect other filesystems, even if they are mounted read-only.

Design notes

The following items have been identified for future investigation and design work later in the project and are thus not addressed in this design:

- Vulnerability to DoS attacks in D-Bus and proposed solutions.
- Optimization of the SQLite configuration parameters for the specific filesystems in use in Apertis.

No updates as of March 2014.

BTRFS Overview

The most powerful feature of Btrfs[30] is the fact that all information (data + metadata) is stored in the same basic data structures, and all modification of these data structures is performed in a copy-on-write (CoW) fashion.

Since all information on disk is stored using the same type of data structure, this allows metadata and data to share features such as checksumming and striping.

This combined with the fact that Btrfs uses CoW while modifying all information, means that in theory, the filesystem is always consistent if the storage device supports "Force Unit Access[31]" correctly. However, in practice, filesystem bugs, a lack of maturity in the code, and other (unforeseen) problems may prevent this.

BTRFS robustness supporting features

Cheap, fast, and atomic snapshots and rollback

All snapshots in Btrfs are CoW copies of the subvolume being snapshotted with an incremented reference count for the blocks. As a result, creating snapshots is very fast, and they take up a negligible amount of space. Just like every other operation, the snapshot is created atomically by the use of transactions

[30] https://btrfs.wiki.kernel.org/

[31] http://en.wikipedia.org/wiki/SCSI_Write_Commands#Write_.2810.29

and sequenced flushes. Further, all snapshots are actually just subvolumes, and hence can be mounted on their own.

Unlike LVM2, which creates snapshots in the form of block devices that can be mounted, Btrfs creates snapshots in the form of subvolumes, which are represented as subdirectories.

Even though snapshots are displayed in a subdirectory they are not “owned” by that subvolume. Snapshots and subvolumes are identical in Btrfs, and are first-class citizens with respect to other subvolumes. This means that the default subvolume can be set at any time. The change will be made the next time the filesystem is mounted. All the subvolumes, except the top-level subvolume, can also be deleted; irrespective of their relationships with each other.

Repair and recovery

If, for any reason, the root node or the superblock gets corrupted and the filesystem cannot be mounted, mounting in recovery mode will make btrfs check the superblock (or alternate superblocks if the superblock is also corrupted) for alternate roots from previous transactions. This is possible because all modifications to the Btrfs trees are done in a CoW manner and existing roots are not deleted. The filesystem stores the last four roots as a backup for the recovery option.

Checksumming

The header of every chunk of space in Btrfs has space for 32-bytes of checksums of the chunk itself. In addition, there is a checksum tree which maintains checksums for each block of data. Since the data as well as the metadata blocks are referenced in the checksum tree, all information in the filesystem is checksummed.

Currently, Btrfs uses the CRC-32 checksum algorithm, but there are plans to upgrade that, and add the option to set the checksum algorithm when the filesystem is created.