



System updates and rollback

1	<b>Contents</b>	
2	<b>Definitions</b>	<b>2</b>
3	Base OS . . . . .	2
4	Applications . . . . .	2
5	<b>Use cases</b>	<b>2</b>
6	Embedded device on the field . . . . .	2
7	Typical system update . . . . .	2
8	Critical security update . . . . .	2
9	Applications and base OS with different release cadence . . . . .	3
10	Shared base OS . . . . .	3
11	Reselling a device . . . . .	3
12	<b>Non use cases</b>	<b>3</b>
13	User modified device . . . . .	3
14	Unrecoverable hardware failure . . . . .	3
15	Unrecoverable file system corruption . . . . .	3
16	Development . . . . .	3
17	<b>Requirements</b>	<b>4</b>
18	Minimal resource consumption . . . . .	4
19	Work on different hardware platforms . . . . .	4
20	Every updated system should be identical to the reference . . . . .	4
21	Atomic update . . . . .	4
22	Rolling back to the last known good state . . . . .	4
23	Reset to clean state . . . . .	5
24	Update control interface . . . . .	5
25	Handling settings and data . . . . .	5
26	<b>Existing system update mechanisms</b>	<b>5</b>
27	Debian tools . . . . .	5
28	ChromeOS . . . . .	6
29	<b>Approach</b>	<b>7</b>
30	Advantages of using OSTree . . . . .	7
31	The OSTree model . . . . .	8
32	Resilient upgrade workflow . . . . .	10
33	Online web-based OTA updates . . . . .	11
34	Offline updates . . . . .	14
35	Switching to the new branch . . . . .	14
36	Online web-based OTA updates using OSTree Static Deltas . . . . .	15
37	OSTree security . . . . .	16
38	Verified boot . . . . .	16
39	Verified updates . . . . .	17
40	Offline update files with signed metadata . . . . .	18

41	Securing OSTree updates download . . . . .	19
42	Controlling access to the updates repository . . . . .	19
43	Security concerns for offline updates over external media . . . . .	19
44	Settings . . . . .	20
45	Error handling . . . . .	21
46	<b>Implementation</b>	<b>23</b>
47	The general flow . . . . .	23
48	The boot count . . . . .	23
49	The bootloader integration . . . . .	24
50	The updater daemon . . . . .	25
51	Detecting new available updates . . . . .	25
52	Initiating the update process . . . . .	25
53	Reporting the status to interested clients . . . . .	25
54	Resetting the boot count . . . . .	26
55	Marking deployments . . . . .	26
56	Command line HMI . . . . .	26
57	Update validation . . . . .	27
58	Testing . . . . .	27
59	Images can be updated . . . . .	27
60	The update process is robust in case of errors . . . . .	27
61	Images roll back in case of error . . . . .	27
62	Images are a suitable rollback target . . . . .	28
63	<b>User and user data management</b>	<b>28</b>
64	<b>Application management</b>	<b>28</b>
65	Application storage . . . . .	29
66	<b>Further developments</b>	<b>29</b>
67	<b>Related Documents</b>	<b>30</b>
68	This document focuses on the system update mechanism, but also partly ad-	
69	dresses applications and how they interact with it.	

## 70 Definitions

### 71 Base OS

72 The core components of the operating system that are used by almost all Apertis  
73 users. Hardware control, resource management, service life cycle monitoring,  
74 networking

### 75 Applications

76 Components that work on top of the base OS and are specific to certain usages.

## 77 **Use cases**

78 A variety of use cases for system updates and rollback are given below.

### 79 **Embedded device on the field**

80 An Apertis device is shipped to a location that cannot be easily accessed by a  
81 technician. The device should not require any intervention in the case of errors  
82 during the update process and should automatically go back to a know-good  
83 state if needed.

84 The update process should be robust against power losses and low voltage situ-  
85 ations, loss of connectivity, storage exhaustion, etc.

### 86 **Typical system update**

87 The user can update his system to run the latest published version of the soft-  
88 ware. This can be triggered either via periodic polling, upon user request, or  
89 any other suitable mean.

### 90 **Critical security update**

91 In the case of a critical security issue, the OEM could push an “update avail-  
92 able” message to some component in the device that would in turn trigger the  
93 update. This requires an infrastructure to reference all devices on the OEM  
94 side. The benefit compared to periodic polling is that the delay between the  
95 update publication and the update trigger is shortened.

### 96 **Applications and base OS with different release cadence**

97 Base OS releases involve many moving parts while application releases are sim-  
98 pler, so application authors want a faster release cadence decoupled from the  
99 base OS one.

### 100 **Shared base OS**

101 Multiple teams using the same hardware platform want to use the same base  
102 OS and differentiate their product purely with applications on top of it.

### 103 **Reselling a device**

104 Under specific circumstances, the user might want to reset his device to a clean  
105 state with no device-specific or personal data. This can happen before reselling  
106 the device or the user encountered an unexpected failure.

## 107 **Non use cases**

### 108 **User modified device**

109 The user has modified his device. For example, they mounted the file system  
110 read write, and tweaked some configuration files to customize some features. As  
111 a result, the system update mechanism may no longer be functional.

112 It might still be possible to restore the operating system to a factory state but  
113 the system update mechanism cannot guarantee it.

### 114 **Unrecoverable hardware failure**

115 An hardware failure has damaged the flash storage or another core hardware  
116 component and the system is no longer able to boot. Compensating for hardware  
117 failures is not part of the system update mechanism.

### 118 **Unrecoverable file system corruption**

119 The file system became corrupted due to a software bug or other failure and is  
120 not able to automatically correct the error. How to recover from that situation  
121 is not part of the system update and rollback mechanism.

## 122 **Development**

123 Developers need to modify and customize their environment in a way that often  
124 conflicts with the requirements for devices on the field.

## 125 **Requirements**

### 126 **Minimal resource consumption**

127 Some devices only have a very limited amount of available storage, the system  
128 update mechanism must keep the impact storage requirement as low as possible  
129 and have a negligible impact at runtime.

### 130 **Work on different hardware platforms**

131 Different devices may use different CPU architectures, bootloaders, storage tech-  
132 nologies, partitioning schemas and file system formats.

133 The system update mechanism must be able to work across them with mini-  
134 mal changes, ranging from single-partition systems running UBIFS on NAND  
135 devices to more common storage devices using traditional file systems over mul-  
136 tiple partitions.

## 137 **Every updated system should be identical to the reference**

138 The file system contents of the base OS on the updated devices must match  
139 exactly the file system used during testing to ensure that its behavior can be  
140 relied upon.

141 This also means that applications must be kept separate from the base OS to  
142 be able to update them while keeping the base OS immutable.

## 143 **Atomic update**

144 To guarantee robustness in case of errors, every update to the system must be  
145 atomic.

146 This means that if an update is not successful, it must not be partially installed.  
147 The failure must leave the device in the same state as if the update did not start  
148 and no intermediate state must exist.

## 149 **Rolling back to the last known good state**

150 If the system cannot boot correctly after an update has been installed success-  
151 fully it must automatically roll back to a known working version.

152 Applications must be kept separated to be able to roll back the base OS while  
153 preserving them or to roll them back while keeping the base OS unchanged.

154 The policy deciding what to roll back and when is product-specific and must  
155 be customizable. For instance, some products may chose to only roll back the  
156 base OS and keep applications untouched, some other products may choose to  
157 roll applications back as well.

158 Rollbacks can be misused to perform [downgrade attacks](https://en.wikipedia.org/wiki/Downgrade_attack)<sup>1</sup> where the attacker  
159 purposefully initiates a rollback to an older version to leverage vulnerabilities  
160 fixed in the currently deployed version.

161 For this reason care need to be taken about the conditions on which a rollback  
162 is to be initiated. For instance, if the system is not explicitly in the process of  
163 performing an upgrade, rollback should never be initiated even in case of boot  
164 failure as those are likely due to external reasons and rolling back to a previous  
165 version would not produce any benefit. Relatedly, once a specific version has  
166 been booted successfully, the system should never roll back to earlier versions.  
167 This also simplifies how applications have to deal with base OS updates: since  
168 the version of the successfully booted deployment can only monotonically in-  
169 crease, user applications that get launched after the successful system boot has  
170 been confirmed will never have to deal with downgrades.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Downgrade\\_attack](https://en.wikipedia.org/wiki/Downgrade_attack)

## 171 **Reset to clean state**

172 The user must be able to restore his device to a clean state, destroying all user  
173 data and all device-specific system configuration.

## 174 **Update control interface**

175 An interface must be provided by the updates and rollback mechanism to allow  
176 HMI to query the current update status, and trigger updates and rollback.

## 177 **Handling settings and data**

178 System upgrades should keep both settings and data safe and intact as this  
179 process should be as transparent as possible to the end user. As described in  
180 [preferences and persistence](#)<sup>2</sup> settings have a default value, which can change on  
181 upgrade, this results in the required solution being more complex than it might  
182 initially seem.

## 183 **Existing system update mechanisms**

### 184 **Debian tools**

185 The Debian package management binds all the software in the system. This can  
186 be very convenient and powerful for administration and development, but this  
187 level of management is not required for final users of Apertis. For example:

- 188 • Package administration command line tools are not required for final users.
- 189 • No support for update roll back. If there is some package breakage, or  
190 broken upgrade, the only way to solve the issue is manually tracking the  
191 broken package and downgrading to a previous version, solving dependen-  
192 cies along the way. This can be an error prone manual process and might  
193 not be accomplished cleanly.

194 In relation to system settings as defined in [preferences and persistence](#)<sup>3</sup>, Debian  
195 tools use a very simple approach. On package upgrades the `dpkg` will perform a  
196 check taking into account

- 197 • current version default configuration file
- 198 • new version default configuration file
- 199 • current configuration file

200 Different scenarios arise depending on whether user has applied changes to the  
201 configuration file. If current default configuration file is the same as current,  
202 then the user hadn't change it, which implies that it can be safely upgraded (if  
203 it is required).

---

<sup>2</sup><https://www.apertis.org/concepts/preferences-and-persistence/>

<sup>3</sup><https://www.apertis.org/concepts/preferences-and-persistence/>

204 However, if the current default configuration file is different from current the  
205 user had applied some changes, so it can't be upgraded silently. In this case  
206 `dpkg` asks the user to choose the version to use. This approach is not suitable  
207 for automated upgrades where there is no user interaction.

208 To overcome some of these limitations modern systems tend to use overlays to  
209 have a read-only partition with default values and an upper layer with custom  
210 values.

## 211 **ChromeOS**

212 ChromeOS uses an A/B parallel partition approach. Instead of upgrading the  
213 system directly, it installs a fresh image into B partition for kernel and rootfs,  
214 then flag those to be booted next time.

215 The partition metadata contains boot fields for the boot attempts (successful  
216 boots) and these are updated for every boot. If a predetermined number of  
217 unsuccessful boots is reached, the bootloader falls back to the other partition,  
218 and it will continue booting from there until the next upgrade is available. When  
219 the next upgrade becomes available it will replace the failing installation and  
220 will attempt booting from there again.

221 There are some drawbacks to this approach when compared to OSTree:

- 222 • The OS installations are not deduplicated, the system stores the entire  
223 contents of the A and B installations separately, where as OSTree based  
224 systems only store the base system plus a delta between this and any  
225 update using Unix hard links. This means an update to the system only  
226 requires disk space proportional to the changed files.
- 227 • The A/B approach can be less efficient since it will need to add extra  
228 layers to work with different partitions, for example, using a specific layer  
229 to verify integrity of the block devices, where OSTree directly handles  
230 operating system views and a content addressable data store (file system  
231 user space) avoiding the need of having different layers.
- 232 • Several partitions are usually required to implement this model, reducing  
233 the flexibility with which the storage space in the device can be utilized.

## 234 **Approach**

235 Package-based solutions fail to meet the robustness requirements, while dual  
236 partitioning schemes have storage requirements that are not viable for smaller  
237 devices.

238 **OSTree**<sup>4</sup> provides atomic updates on top of any POSIX-compatible file system  
239 including UBIFS on NAND devices, is not tied to a specific partitioning scheme  
240 efficiently handles the available storage.

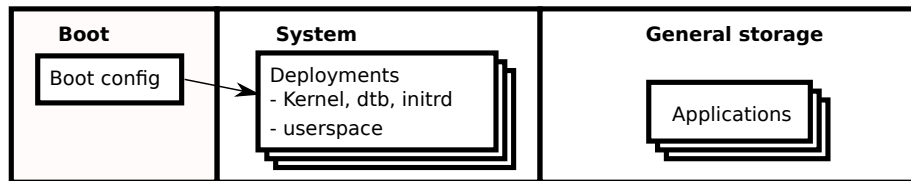
---

<sup>4</sup><http://ostree.readthedocs.io>



241 No specific requirements are imposed on the partitioning schema. Use of  
242 the GUID Partition Table (GPT<sup>5</sup>) system for partition management is  
243 recommended for being flexible while having fail-safe measures, like keeping  
244 checksums of the partition layout and providing some redundancy in case  
245 errors are detected.

246 Separating the system volume from the general storage volume, where applica-  
247 tions and user data are stored, is also recommended.



## 248 GPT Partitions

249 More complex schemas can be used for instance by combining OSTree with read-  
250 only fallback partitions to handle file system corruption on the main system  
251 partition, but this document focuses on a OSTree-only setup that provides a  
252 good balance between complexity and robustness.

### 253 Advantages of using OSTree

- 254 • OSTree operates at the Unix file system layer and thus on top of any  
255 file system or block storage layout, including NAND flash setups, and in  
256 containers.
- 257 • OSTree does not impose strict requirements on the partitioning scheme  
258 and can scale down to a single partition while fully preserving its resiliency  
259 guarantees, saving space on the device and avoiding extra layers of com-  
260 plexity (for example, to verify partition blocks). Depending on the setup,  
261 multiple partitions can still be used effectively to separate contents with  
262 different life cycles, for instance by storing user data on a different parti-  
263 tion than the system files managed by OSTree.
- 264 • OSTree commits are centrally created offline (server side), and then they  
265 are deployed by the client. This gives much more control over what the  
266 devices actually run.
- 267 • It can store multiple file systems trees in a single repository.
- 268 • It is designed to implement fully atomic and resilient upgrades. If the  
269 system crashes or power is lost at any point during the update process,  
270 you will have either the old system, or the new one.
- 271 • It clearly separate the OS from the device configuration and user data,  
272 so resetting the system to a clean state simply involves deleting some  
273 directories and their contents.

<sup>5</sup>[http://en.wikipedia.org/wiki/GUID\\_Partition\\_Table](http://en.wikipedia.org/wiki/GUID_Partition_Table)

- OSTree is implemented as a shared library, making it very easy to build higher level projects or tools on top of it.
- The files in `/usr` contents are mounted read-only from subfolders of `/ostree/deploy`, minimizing the chance of accidental deletions or changes.
- OSTree has no impact on startup performance, nor does increase resource usage during runtime: since OSTree is just a different way to build the rootfs once it is built it will behave like a normal rootfs, making it very suitable for setups with limited storage.
- OSTree already offers a mechanism suitable for offline updates using static deltas, which can be used for updates via a mass-storage device.
- Security is at the core of OSTree, offering content replication incrementally over HTTPS via cryptographic signatures (using the ED25519 algorithm on Apertis) and SHA256 hash checksums.
- The mechanism to apply partial updates or full updates is exactly the same, the only difference is how the updates are generated on the server side.
- OSTree can be used for both the base OS and applications, and its built-in hard link-based deduplication mechanism allow to share identical contents between the two, to keep them independent while having minimal impact on the needed storage. The Flatpak application framework is already based on OSTree.

## The OSTree model

The conceptual model behind OSTree repositories is very similar to the one used by `git`, to the point that the [introduction in the OSTree manual](https://ostree.readthedocs.io/en/stable/manual/introduction/)<sup>6</sup> refers to it as “git for operating system binaries”.

Albeit they take different tradeoffs to address different use-cases they both have:

- file contents stored as blobs addressable by their hash, deduplicating them
- file trees linking filenames to the blobs
- commits adding metadata such as dates and comments on top of file trees
- commits linked in a history tree
- branches pointing to the most recent commit in a history tree, so that clients can find them

Where `git` focuses on small text files, OSTree focuses on large trees of binary files.

On top of that OSTree adds other layers which go beyond storing and distributing file contents to fully handle operating system upgrades:

- repositories - store one or more versions of the file system contents as described above
- deployments - specific file system versions checked-out from the repository

---

<sup>6</sup><https://ostree.readthedocs.io/en/stable/manual/introduction/>

- statEROOTS - the combination of immutable deployments and writable directories

Each device hosts a local OSTree repository with one or more deployments checked out.

Checked out deployments look like traditional root file systems. The bootloader points to the kernel and initramfs carried by the deployment which, after setting up the writable directories from the statEROOT, are responsible for booting the system. The bootloader is not part of the updates and remains unchanged for the whole lifetime of the device as any changes has a high chance to make the system unbootable.

- Each deployment is grouped in exactly one statEROOT, and in normal circumstances Apertis devices only have a single apertis statEROOT.
- A statEROOT is physically represented in the `/ostree/deploy/$statEROOT` directory, `/ostree/deploy/apertis` in this case.
- Each statEROOT has exactly one copy of the traditional Unix `/var` directory, stored physically in `/ostree/deploy/$statEROOT/var`. The `/var` directory is persisted during updates, when moving from one deployment to another and it is up to each operating system to manage this directory.
- On each device there is an OSTree repository stored in `/ostree/repo`, and a set of deployments stored in `/ostree/deploy/$statEROOT/$checksum`.
- A deployment begins with a specific commit (represented by a SHA256 hash) in the OSTree repository in `/ostree/repo`. This commit refers to a file system tree that represents the underlying basis of a deployment.
- Each deployment is primarily composed of a set of hard links into the repository. This means each version is deduplicated; an upgrade process only costs disk space proportional to the new files, plus some constant overhead.
- The read-only base OS contents are checked out in the `/usr` directory of the deployment.
- Each deployment has its own writable copy of the configuration store `/etc`.
- Deployments don't have a traditional UNIX `/etc` but ship it instead as `/usr/etc`. When OSTree checks out a deployment it performs a 3-way merge using the old default configuration, the active system's `/etc`, and the new default configuration.
- Besides the exceptions of `/var` and `/etc` directories, the rest of the contents of the tree are checked out as hard links into the repository.
- Both `/etc` and `/var` are persistent writable directories that get preserved across upgrades. Additionally since `/home` is used to store user specific data it is also writable and preserved across updates.

## Resilient upgrade workflow

The following steps are performed to upgrade a system using OSTree:

- The system boots using the existing deployment

- A new version is made available as a new OSTree commit in the local repository, either downloading it from the network or by unpacking a static delta shipped on a mass storage device.
- The data is validated for integrity and appropriateness.
- The new version is deployed.
- The system reboots into the new deployment.
- If the system fails to boot properly (which should be determined by the system boot logic), the system can roll back to the previous deployment.

During the upgrade process, OSTree will take care of many important details, like for example, managing the bootloader configuration and correctly merging the `/etc` directory.

Each `commit` can be delivered to the target system over the air or by attaching a mass storage device. Network upgrades and mass storage upgrades only differ in the mechanism used by `ostree` to detect and obtain the update. In both cases the `commit` is first stored in a temporary directory, validated and only then it becomes part of the local OSTree repository before the real upgrade process starts by rebooting in the new deployment.

Metadata such as EdDSA or GPG signatures can be attached to each `commit` to validate it, ensuring it is appropriate for the current system and it has not been corrupted or tampered. The update process must be interrupted at any point during the update process should any check yield an invalid result; the [atomic upgrades mechanism in OSTree<sup>7</sup>](#) ensures that it is safe to stop the process at any point and no change is applied to the system up to the last step in the process.

The atomic upgrades mechanism in OSTree ensures that any power failure during the update process leaves the current system state unchanged and the update process can be resumed re-using all the data that has already been already validated and included in the local repository.

## Online web-based OTA updates

OSTree supports bandwidth-efficient retrieval of updates over the network.

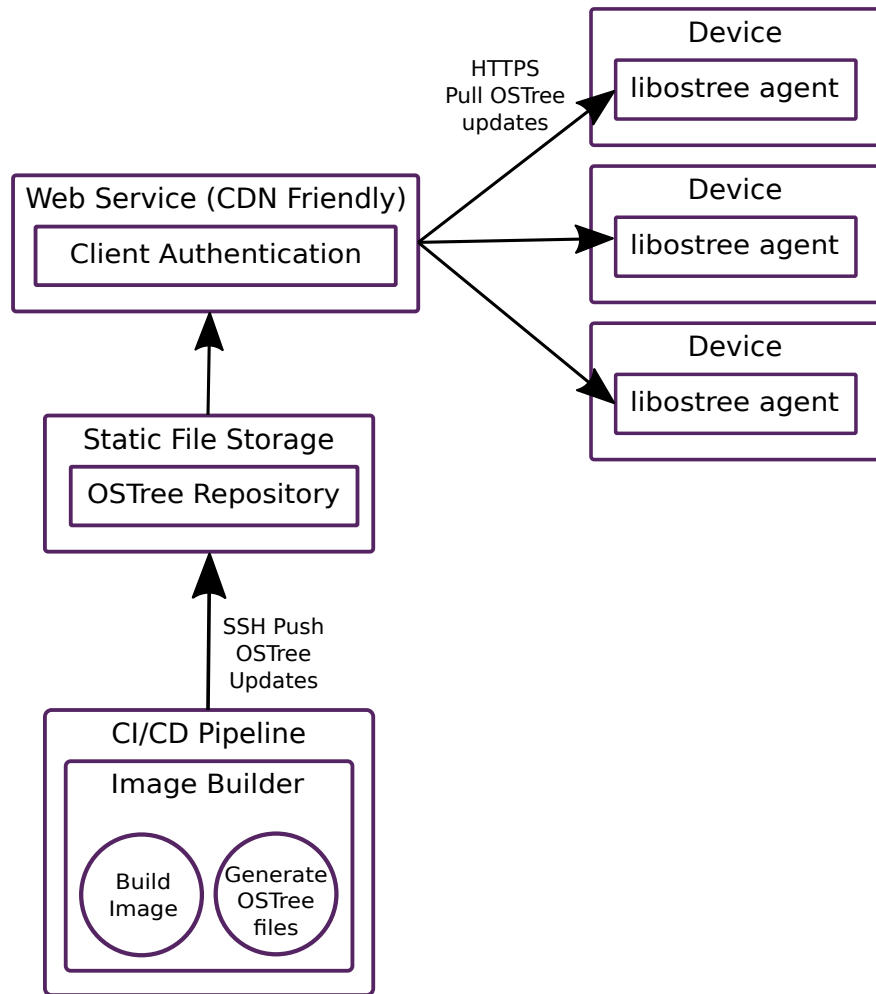
The basic workflow involves the actors below:

- the image building pipelines pushes commits to an OSTree repository on each build;
- a standard web server provides access over HTTPS to the OSTree repository handling it as a plain hierarchy of static files, with no special knowledge of OSTree;
- the client devices poll the web server and retrieve updates when they get published.

---

<sup>7</sup><https://ostree.readthedocs.io/en/stable/manual/atomic-upgrades>

393 The following diagram shows how the data flows across services when using the  
 394 web based OSTree upgrade mechanism.



395  
 396 Thanks to its repository format, OSTree client devices can efficiently query the  
 397 repository status and retrieve only the changed contents without any OSTree-  
 398 specific support in the web server, with the repository files being served as plain  
 399 static files.

400 This means that any web hosting provider can be used without any loss of  
 401 efficiency.

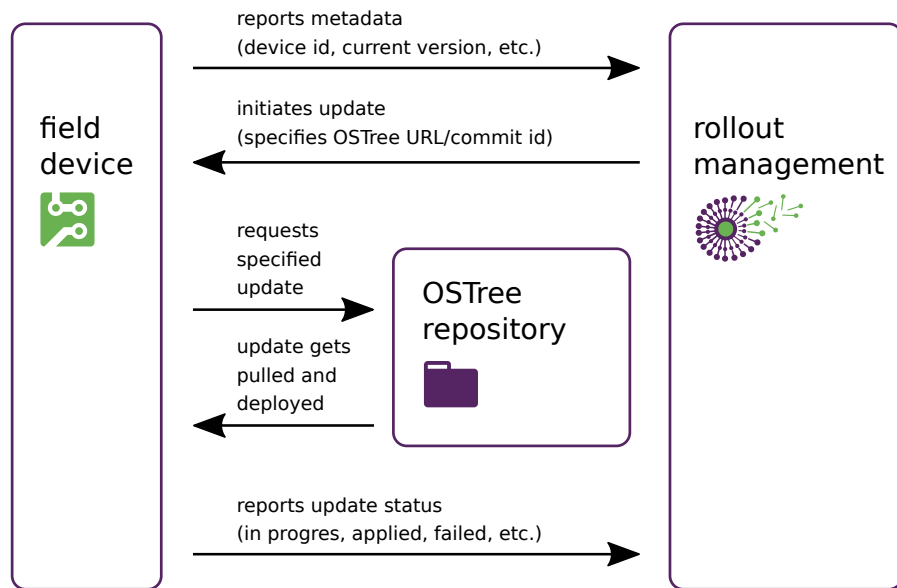
402 By only requiring static files, the web service can easily take advantage of CDN  
 403 services to offer a cost efficient solution to get the data out to the devices in a

404 way that is globally scalable.

405 The authentication to the web service can be done via HTTP Basic authentication,  
406 SSL/TLS client certificates, or any cookie-based mechanism that is most  
407 suitable for the chosen web service, as OSTree does not impose any constraint  
408 over plain HTTPS. OSTree separately checks the chain of trust linking the down-  
409 loaded updates to the keys trusted by the system update manager. See also the  
410 [Controlling access to the updates repository](#) and [Verified updates](#) sections in  
411 this regard.

412 Monitoring and management of devices can be built using the same HTTPS  
413 access as used by OSTree or using completely separated mechanisms, enabling  
414 the integration of OSTree updates into existing setups.

415 For instance, integration with roll out management suites like [Eclipse hawkBit](#)<sup>8</sup>  
416 can happen by disabling the polling in the OSTree updater and letting the man-  
417 agement suite tell OSTree which commit to download and from where through  
418 a dedicated agent running on the devices.



419

420 This has the advantage that the roll out management suite would be in complete  
421 control of which updates should be applied to which devices, implementing  
422 any kind of policies like progressive staged roll outs with monitoring from the  
423 backend with minimal integration.

424 Only the retrieval and application of the actual update data on the device

<sup>8</sup><https://www.eclipse.org/hawkbit/>

would be offloaded to the OSTree-based update system, preserving its network and storage efficiency and the atomicity guarantees.

## Offline updates

Some devices may not have any connectivity, or bandwidth requirements may make full system updates prohibitive. In these cases updates can be made available offline by providing OSTree “static delta” files on external media devices like USB mass storage devices.

The deltas are simple static files that contains all the differences between two specific OSTree commits. The user would download the delta file from a web site and put it in the root of an external drive. After the drive is mounted, the update management system would look for files with a specific name pattern in the root of the drive. If an appropriate file is found, it is checked to be a valid OSTree static bundle with the right metadata and, if that verification passes, the user would get a notification saying that updates are available from the drive. If the update file is corrupted, is targeted to other platforms or devices, or is otherwise invalid, the upgrade process must stop, leaving the system unchanged and a notification may be reported to the user about the identified issues.

Static deltas can be partial if the devices are known beforehand to have a specific OSTree commit already available locally, or they can be full by providing the delta from the `NULL` empty commit, thus ensuring that the update can be applied without any assumption on the version running on the devices at the expense of a potential increase of the requirements on the mass storage device used to ship them. Both partial and full deltas leading to the same OSTree commit will produce identical results on the devices.

## Switching to the new branch

The branches naming schema used in Apertis contains the major version, for instance: `apertis/v2022/armhf-uboot/fixedfunction`. So for Apertis the “major upgrade” is technically considered as switching to another branch with a more recent Apertis version, for example `apertis/v2023/armhf-uboot/fixedfunction`. By default such kinds of offline upgrade with switching to another branch is restricted by the update manager.

Offline upgrades between branches (including “major updates”) consists of 2 steps which should be a part of offline upgrade:

1. Prepare the proper commit at build time

This step is pretty simple – while preparing the relevant commit we just need to add the branch name(s) from which we are supposed to be able to upgrade to the current version. For example, while preparing commit for `v2021` version just add following into `ostree-commit` action:

```
ref-binding:
```

```

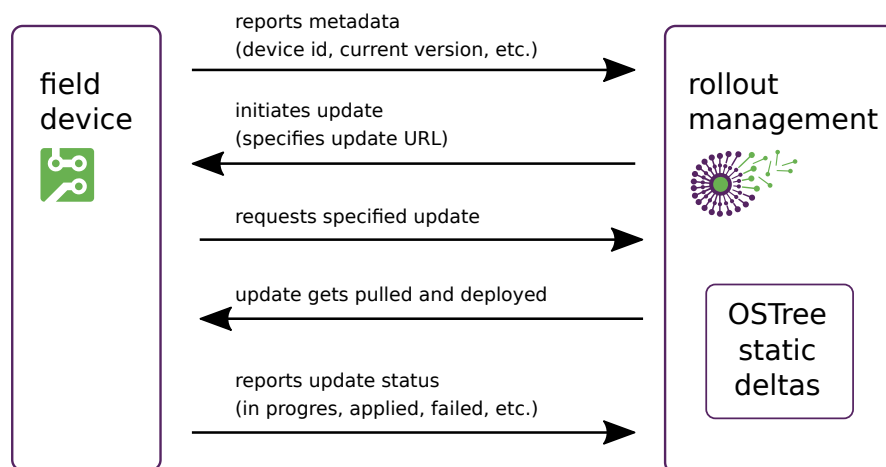
464         - apertis/v2021/{{ $architecture }}-{{ $board }}/{{ $type }}
465         - apertis/v2020/{{ $architecture }}-{{ $board }}/{{ $type }}
466
467     this produce the commit compatible with version v2020 allowing to install
468     the new OS version v2021 onto the board.
469
470     2. Set correct refs in repository
471
472     After successful boot of the updated version all refs in libostree repository
473     are still pointing to the previous branch due the nature of offline upgrade.
474
475     This needs to be fixed for proper detection of further upgrades, including
476     updates over the air. It is the responsibility of the update manager to
477     update the refs once the update has been determined to be successful.
478
479     This functionality requires no changes to be made to previously released OSTree
480     versions. The configuration that determines upgrade paths is held in the newer
481     OSTree commit.
482
483     Apertis currently only supports upgrades to newer versions, downgrades to older
484     versions of Apertis are not supported.

```

## 479 Online web-based OTA updates using OSTree Static Deltas

480 The OSTree based OTA update mechanism described above, whilst taking full  
481 advantage of OSTree's repositories, may not suit all users of Apertis. Where  
482 existing deployment services such as hawkBit are already deployed, a need to  
483 expose and maintain an extra service, the OSTree repository, may not be  
484 welcome.

485 The ability of OSTree to perform updates using static deltas provides us with  
486 the option to serve these via the deployment infrastructure instead of serving  
487 updates from the OSTree repository.



488



Such an approach would likely need to utilize full OSTree deltas in order to ensure updates were viable on target devices regardless of how frequently they had previously been updated. As a result this approach does not take advantage of the bandwidth efficiency that would be presented by **Online web-based OTA updates** as previously discussed, however it does enable updates to be performed via existing installed deployment infrastructure.

This is the approach currently used by the Apertis example implementation.

## OSTree security

OSTree is a distribution method. It can secure the downloading of the update by verifying that it is properly signed using public key cryptography (EdDSA or GPG). It is largely orthogonal to verified boot, that is ensuring that only signed data is executed by the system from the bootloader, to the kernel and user space. The only interaction is that since OSTree is a file-based distribution mechanism, block-based verification mechanism like `dm-verity` cannot be used. OSTree can be used in conjunction with signed bootloader, signed kernel, and IMA (Integrity Measurement Architecture) to provide protection from offline attacks.

## Verified boot

Verified boot is the process which ensures a device is only runs signed code. This is a layered process where each layer verifies signature of its upper layer.

The bottom layer is the hardware, which contains a data area reserved to certificates. The first step is thus to provide a signed bootloader. The processor can verify the signature of the bootloader before starting it. The bootloader then reads the boot configuration file. It can then run a signed kernel and `initramfs`. Once the kernel has started, the `initramfs` mounts the root file system.

At the time of writing, the boot configuration file is not signed. It is read and verified by signed code, and can only point to signed components.

Protecting bootloader, kernel and `initramfs` already guarantees that policies baked in those components cannot be subverted through offline attacks. By verifying the content of the rootfs the protection can be extended to user space components, albeit such protection can only be partial since device-local data can't be signed on the server-side like the rest of the rootfs.

To protect the rootfs different mechanisms are available: the block-based ones like `dm-verity` are fundamentally incompatible with file-based distribution methods like OSTree, since they rely on the kernel to verify the signature on each read at the block level, guaranteeing that the whole block device has not been changed compared to the version signed at deployment time. Due to working on raw block devices, `dm-verity` is also incompatible with UBIFS and thus it is unsuitable for NAND devices.

528 Other mechanisms like IMA (Integrity Measurement Architecture) work instead  
529 at the file level, and thus can be used in conjunction with UBIFS and OSTree  
530 on NAND devices.

531 It is also possible to check that the deployed OSTree rootfs matches the server-  
532 provided signature without using any other mechanism, but unlike IMA and  
533 `dm-verity` such check would be too expensive to be done during file access.

### 534 **Verified updates**

535 Once a verified system is running, an OSTree update can be triggered. Apertis is  
536 using [ed25519](https://ed25519.cr.yp.to/)<sup>9</sup> variant of EdDSA signature. Ed25519 ensures that the commit  
537 was not modified, damaged, or corrupted.

538 On the server, OSTree commits must be signed using ed25519 secret key. This  
539 occurs via the `ostree sign --sign-type=ed25519 <COMMIT_ID>` command line. The  
540 secret key could be provided via additional CLI parameter or file by using option  
541 `--keys-file=<path_to_file>`.

542 OSTree expect what secret key consists of 64 bytes (32b seed + 32b public)  
543 encoded with base64 format. The ed25519 secret and public parts could be  
544 generated by numerous utilities including `openssl`, for instance:

```
545 openssl genpkey -algorithm ed25519 -outform PEM -out ed25519.pem
```

546 Since OSTree is not capable to use PEM format directly, it is needed to [extract](#)  
547 [the secret and public keys](#)<sup>10</sup> from PEM file, for example:

```
548 PUBLIC="$(openssl pkey -outform DER -pubout -in ${PEMFILE} | tail -  
549 c 32 | base64) "  
550 SEED="$(openssl pkey -outform DER -in ${PEMFILE} | tail -c 32 | base64) "
```

551 As mentioned above, the secret key is concatenation of SEED and PUBLIC  
552 parts:

```
553 SECRET="$(echo ${SEED}${PUBLIC} | base64 -d | base64 -w 0) "
```

554 On the client, ed25519 is also used to ensure that the commit comes from a  
555 trusted provider since updates could be acquired through different methods like  
556 OTA over a network connection, offline updates on plug-in mass storage devices,  
557 or even mesh-based distribution mechanism. To enable the signature check,  
558 repository on the client must be configured by adding option `sign-verify=true`  
559 into the `core` or `per-remote` section, for instance:

```
560 ostree config set 'remote "origin".sign-verify' "true"
```

561 OSTree searches for files with valid public signatures in directories  
562 `/usr/share/ostree/trusted.ed25519.d` and `/etc/ostree/trusted.ed25519.d`.  
563 Any public key in a file in these directories will be trusted by the client. Each

---

<sup>9</sup><https://ed25519.cr.yp.to/>

<sup>10</sup><http://openssl.6102.n7.nabble.com/ed25519-key-generation-td73907.html>

564 file may contain multiple keys, one base64-encoded public key per string. No  
565 private keys should be present in these directories.

566 In addition it is possible to provide the trusted public key per-remote by  
567 adding into remote's configuration path to the file with trusted public keys (via  
568 `verification-file` option) or even single key itself (via `verification-key`).

569 In the OSTree configuration, the default is to require commits to be signed.  
570 However, if no public key is available, no any commit can be trusted.

### 571 **Offline update files with signed metadata**

572 Starting with version v2020.7 `libostree` supports delta bundles with [signed meta-](#)  
573 [data](#)<sup>11</sup>, which allows to ensure that the whole delta bundle comes from a trusted  
574 source. Previous versions only allowed to assert the provenance of the commits  
575 in the bundle, leaving the metadata unverified.

576 Support for delta bundles with signed metadata is available in the Apertis Up-  
577 date Manager since version 0.2020.20, which can also handle delta bundles with  
578 unsigned metadata. Previous versions of the Apertis Update Manager also  
579 supported an experimental format for signed metadata, which has now been  
580 dropped in favor of the format that has been landed upstream in `libostree`  
581 2020.7.

582 To improve the security on target devices the repository configuration must have  
583 additional option `core.sign-verify-deltas` set to `true`:

```
584 ostree config set core.sign-verify-deltas "true"
```

585 This is forcing AUM and `libostree` to accept only update bundles with signed  
586 metadata.

### 587 **Compatibility with upgrades**

588 Until v2021pre there were no support of upgrade bundles with signed metadata  
589 in Apertis.

590 For upgrading from version v2020 to v2021 we have produced additional upgrade  
591 bundle. This additional bundle has unsigned metadata, allowing offline upgrades  
592 from the previous release. So for v2021, and only for v2021 we have 3 update  
593 bundle types:

- 594 • `*.delta` – depending on CI it may have signed or unsigned metadata. This  
595 version is uploaded into hawkBit server and used for tests.
- 596 • `*.delta.enc` – encrypted bundle containing delta file above.
- 597 • `*.compat-v2020.delta` – bundle with unsigned metadata compatible with  
598 previous Apertis release. This file should be used for upgrading the Apertis  
599 version v2020.

---

<sup>11</sup><https://github.com/ostreedev/ostree/pull/1985>

## 600 **Securing OSTree updates download**

601 OSTree supports “pinned TLS”. Pinning consist of storing the public key of the  
602 trusted host on the client side, thus eliminating the need for a trust authority.

603 TLS can be configured in the `remote` configuration on the client using the follow-  
604 ing entries:

```
605 tls-ca-path  
606     Path to file containing trusted anchors instead of the system CA database.
```

607 Once a key is pinned, OSTree is ensured that any download is coming from a  
608 host which key is present in the image.

609 The pinned key can be provided in the disk image, ensuring every flashed device  
610 is able to authenticate updates.

## 611 **Controlling access to the updates repository**

612 TLS also permit the OSTree client to authenticate itself to the server before  
613 being allowed to download a commit. This can also be configured in the `remote`  
614 configuration on the client using the following entries:

```
615 tls-client-cert-path  
616     Path to file for client-side certificate, to present when making requests to  
617     this repository.  
618 tls-client-key-path  
619     Path to file containing client-side certificate key, to present when making  
620     requests to this repository.
```

621 Access to remote repositories can also be controlled via HTTP cookies. The  
622 `ostree remote add-cookie` and `ostree remote delete-cookie` commands will up-  
623 date a per-remote lookaside cookie jar, named `$remotename.cookies.txt`. In this  
624 model, the client first obtains an authentication cookie before communicating  
625 this cookie to the server along with its update request.

626 The choice between authentication via TLS client-side certificates or HTTP  
627 cookies can be done depending on the chosen server-side infrastructure.

628 Provisioning authentication keys on a per-device basis at the end of the deliv-  
629 ery chain is recommended so each device can be identified and access granted  
630 or denied at the device granularity. Alternatively it is possible to deploy au-  
631 thentication keys at coarser granularities, for instance one for each device class,  
632 depending on the specific use-case.

## 633 **Security concerns for offline updates over external media**

634 OSTree static deltas includes the detached metadata with signature for the  
635 contained commit to check if the commit is provided by a valid provider and its  
636 integrity.

637 The signed commit is unpacked to a temporary directory and verified by OSTree  
638 before being integrated in the OSTree repository on the device, from which it  
639 can be deployed at the next reboot.

640 This is the same mechanism used for commit verification when doing OTA  
641 upgrades from remote servers and provides the same features and guarantees.

642 Usage of inlined signed metadata ensures that the provided update file is aimed  
643 to the target platform or device.

644 Updates from external media present a security problem not present for directly  
645 downloaded updates. Simply verifying the signature of a file before decompress-  
646 ing is an incomplete solution since a user with sufficient skill and resources  
647 could create a malicious USB mass storage device that presents different data  
648 during the first and second read of a single file – passing the signature test, then  
649 presenting a different image for decompression.

650 The content of the update file is extracted into the temporary directory and the  
651 signature is checked for the extracted commit tree.

## 652 Settings

653 As described in [preferences and persistence](#)<sup>12</sup> there are different types of settings  
654 which should be preserved across updates. The setting should either be kept  
655 intact or updated to reflect new logic of the application.

656 When using OSTree, most of the file system is read-only. Since system settings  
657 need write support, the `/etc` and `/var` partitions are configured to be read-write.  
658 This also applies to the `/home` partition, with it being configured as read-write  
659 so user data and settings can be preserved.

660 During an OSTree upgrade, a new commit is applied on the OSTree repo, this  
661 provides the new content that will be used for the read-only portions of the  
662 rootfs, but does not modify the read-write parts. To handle the upgrade of  
663 system settings stored in `/etc`, a copy of its default values are kept in `/usr/etc`  
664 which is updated with the new commit. Thanks to this information OSTree can  
665 detect the files that have been changed and apply a 3-way merge, to update the  
666 `/etc`.

667 This process allows to update settings to new defaults for files that were not  
668 modified and keep intact those that were.

669 Applications are encouraged to handle settings adaptation to new version follow-  
670 ing the guidelines described in [user and user data management](#) and [preference](#)  
671 [and persistence](#)<sup>13</sup>.

---

<sup>12</sup><https://www.apertis.org/concepts/preferences-and-persistence/>

<sup>13</sup><https://www.apertis.org/concepts/preferences-and-persistence/>

## 672 **Error handling**

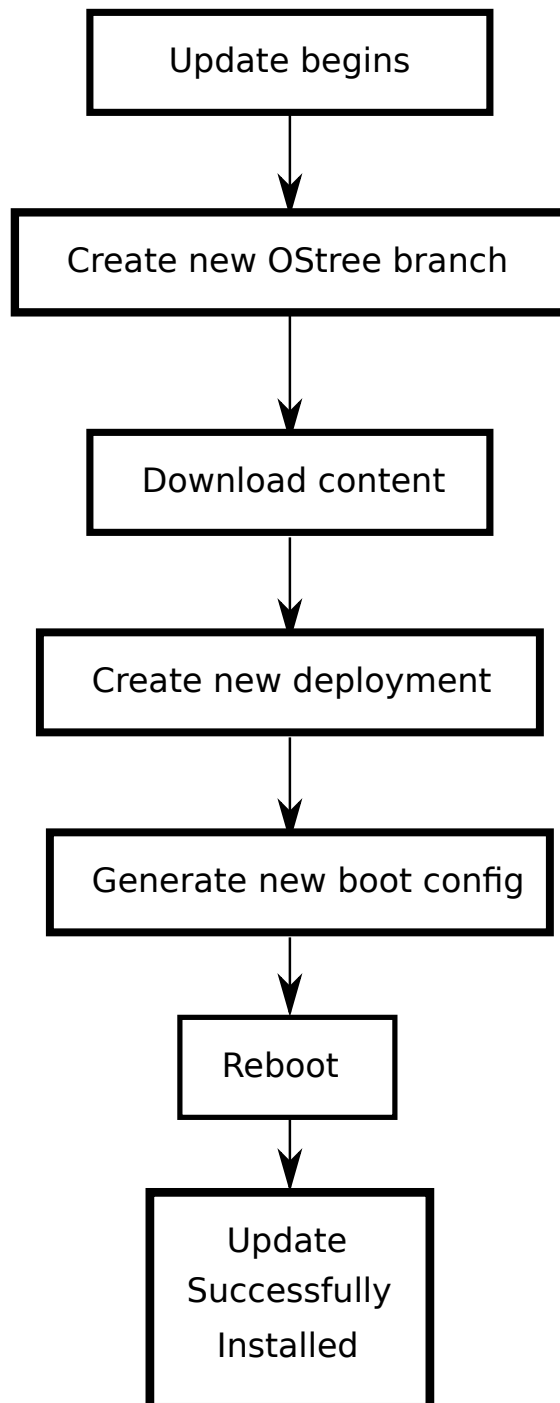
673 If for any reason the update process fails to complete, the update will be black-  
674 listed to avoid re-attempting it. Another update won't be automatically at-  
675 tempted until a newer update is made available.

676 The only exception from this rule is failure due incorrect signature check. The  
677 commit could be re-signed with the key not known for the client at this moment,  
678 and as soon as client acquire the new public key blacklist mechanism shouldn't  
679 prevent the update.

680 It is possible that an update is successfully installed yet fail to boot, resulting in  
681 a rollback. In the event of a rollback the update manager must detect that the  
682 new update has not been correctly booted, and blacklist the update so it is not  
683 attempted again. To detect a failed boot a watchdog mechanism can be used.  
684 The failed updates can then be blacklisted by appending their OSTree commit  
685 ids to a list.

686 This policy prevents a device from getting caught in a loop of rollbacks and  
687 failed updates at the expense of running an old version of the system until a  
688 newer update is pushed.

689 The most convenient storage location for the blacklist is the user storage area,  
690 since it can be written at runtime. As a side effect of storing the blacklist there,  
691 it will automatically be cleared if the system is reset to a clean state.



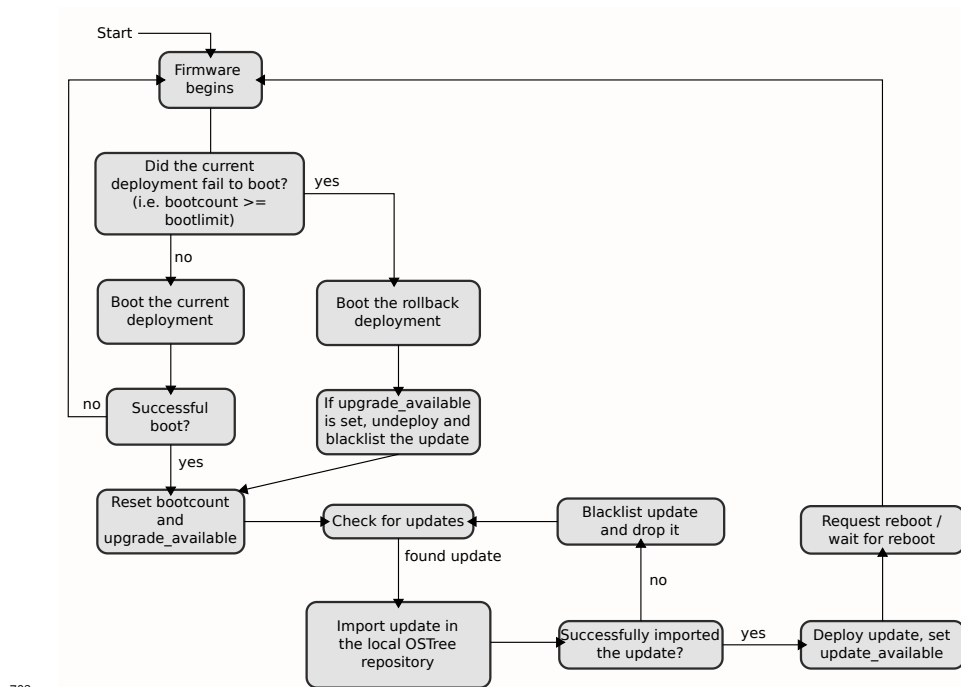
## 693 Implementation

694 This section provides some more details about the implementation of offline  
695 system updates and rollback in Apertis, which is split in three main components:

- 696 • the updater daemon
- 697 • the bootloader integration
- 698 • the command-line HMI

## 699 The general flow

700 The Apertis update process deals with selecting the OSTree deployment to boot,  
701 rolling back to known-good deployments in case of failure and preparing the new  
702 deployment on updates:



703  
704 While the underlying approach differs due to the use of OSTree in Apertis over  
705 the dual-partition approach chosen by ChromeOS and the different bootloaders,  
706 the update/rollback process is largely the same as [the one in ChromeOS<sup>14</sup>](https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate#TOC-Diagram).

## 707 The boot count

708 To keep track of failed updates the system maintains a persistent counter that  
709 it is increased every time a boot is attempted.

<sup>14</sup><https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate#TOC-Diagram>



710 Once a boot is considered successful depending on project-specific policies (for  
711 instance, when a specific set of services has been started with no errors) the  
712 boot count is reset to zero.

713 This boot counter needs to be handled in two places:

- 714 • in the bootloader, which boots the current OSTree deployment if the  
715 counter is zero and initiates a rollback otherwise
- 716 • in the updater, which needs to reset it to zero once the boot is considered  
717 successful

718 Using the main persistent storage to store the boot count is viable for most  
719 platform but would produce too much wear on platforms using NAND devices.  
720 In those cases the boot count should be stored on another platform-specific  
721 location which is persistent over warm reboots, there's no need for it to persist  
722 over power losses.

723 However, in the reference implementation the focus is on the most general solu-  
724 tion first, while being flexible enough to accommodate other solutions whenever  
725 needed.

## 726 The bootloader integration

727 Since bootloaders are largely platform-specific the integration needs to be done  
728 per-platform.

729 For the SabreLite ARM 32bit platform, integration with the [U-Boot](#)<sup>15</sup> boot-  
730 loader is needed.

731 OSTree already provides dedicated hooks to update the `u-boot` environment to  
732 point it to the latest deployment.

733 Two separate boot commands are used to start the system: the default one boots  
734 the latest deployment, while the alternate one boots the previous deployment.

735 Before rebooting to a new deployment the boot configuration file is switched  
736 and the one for the new deployment is made the default, while the older one is  
737 put into the location pointed by the alternate boot command.

738 When a failure is detected by checking the boot count while booting the latest  
739 deployment, the system reboots using the alternate boot command into the  
740 previous deployment where the rollback is completed.

741 Once the boot procedure completes successfully the boot count gets reset and  
742 stopped, so failures in subsequent boots won't cause rollbacks which may worsen  
743 the failure.

744 If the system detects that a rollback has been requested, it also need to make  
745 the rollback persistent and prevent the faulty updates to be tried again. To do

---

<sup>15</sup><http://www.denx.de/wiki/U-Boot/WebHome>

746 so, it adds any deployment more recent than the current one to a local blacklist  
747 and then drops them.

## 748 **The updater daemon**

749 The updater daemon is responsible for most of the activities involved, such as  
750 detecting available updates, initiating the update process and managing the  
751 boot count.

752 It handles both online OTA updates and offline updates made available on  
753 locally mounted devices.

## 754 **Detecting new available updates**

755 For offline updates, the [GVolumeMonitor](https://developer.gnome.org/gio/stable/GVolumeMonitor.html)<sup>16</sup> API provided by GLib/GIO is used  
756 to detect when a mass storage device is plugged into the device, and the [GFile](https://developer.gnome.org/gio/stable/GFile.html)<sup>17</sup>  
757 GLib/GIO API is used to scan for the offline update stored as a plain file in the  
758 root of the plugged file system named `static-update.bundle`.

759 For online OTA updates, the [OstreeSysrootUpgrader](https://github.com/ostreedev/ostree/blob/master/src/libostree/ostree-sysroot-upgrader.c)<sup>18</sup> is used to poll the remote  
760 repository for new commits in the configured branch.

761 When combined with roll out management systems like [Eclipse hawkBit](https://www.eclipse.org/hawkbite/)<sup>19</sup>, the  
762 roll out management agent on the device will initiate the upgrade process with-  
763 out the need for polling.

## 764 **Initiating the update process**

765 Once the update is detected, it is verified and compared against a local blacklist  
766 to skip updates that have already failed in the past (see [Update validation]).

767 In the offline case the static delta file is checked for consistency before being  
768 unpacked in the local OSTree repository.

769 During online updates, files are verified as they get downloaded.

770 In both cases the new update results in a commit in the local OSTree repository  
771 and from that point the process is identical: a new deployment is created from  
772 the new commit and the bootloader configuration is updated to point to the  
773 new deployment on the next boot.

## 774 **Reporting the status to interested clients**

775 The updater daemon exports a simple D-Bus interface which allows to check  
776 the state of the update process and to mark the current boot as successful.

---

<sup>16</sup><https://developer.gnome.org/gio/stable/GVolumeMonitor.html>

<sup>17</sup><https://developer.gnome.org/gio/stable/GFile.html>

<sup>18</sup><https://github.com/ostreedev/ostree/blob/master/src/libostree/ostree-sysroot-upgrader.c>

<sup>19</sup><https://www.eclipse.org/hawkbite/>

## 777 **Resetting the boot count**

778 During the boot process the boot count is reset to zero using an interface that  
779 abstracts over the platform-specific approach.

780 While product-specific policies dictate when the boot should be considered  
781 successful, the reference images consider a boot to be successful if the `multi-`  
782 `user.target` target is reached.

## 783 **Marking deployments**

784 Rolled back deployments are added to a blacklist to avoid trying them again  
785 over and over.

786 Deployments that have booted successfully get marked as known good so that  
787 they are never rolled back, even if at a later point a failure in the boot process is  
788 detected. This is to avoid transient issues causing an unwanted rollback which  
789 may make the situation worse.

790 To do so, the boot counting is stopped once the current boot is considered  
791 successful, effectively marking the current boot as known-good without the need  
792 to maintain a whitelist and synchronize it with the bootloader.

793 As a part of marking the deployment as successful the updater daemon checks  
794 the target branches from the OSTree commit metadata. If the booted deploy-  
795 ment contains references to several branches, the updater daemon determines  
796 which branch has the highest version and resets all refs and the origin to that  
797 branch. Using the branches naming scheme used in Apertis, this could be con-  
798 sidered as “major upgrade” of the system. From this point on, the system is  
799 fully switched to the new branch and accepts only upgrades created in the new  
800 branch.

## 801 **Command line HMI**

802 A command line tool is provided to query the status using [the `org.apertis.ApertisUpdateManager`](#)  
803 [D-Bus API](#)<sup>20</sup>:

```
804 $ updatectl  
805 ** Message: Network connected: No  
806 ** Message: Upgrade status: Deploying
```

807 The current API exposes information about whether the updater is idle, an  
808 update is being checked, retrieved or deployed, or whether a reboot is pending  
809 to switch to the updated system.

810 It can also be used to mark the boot as successful:

```
811 $ updatectl --mark-update-successful
```

---

<sup>20</sup><https://gitlab.apertis.org/pkg/apertis-update-manager/-/blob/apertis/v2021/data/apertis-update-manager-dbus.xml>

## 812 **Update validation**

813 Before installing updates the updater check their validity and appropriateness  
814 for the current system, using the metadata carried by the update itself as pro-  
815 duced by the build pipeline. It ensures that the update is appropriate for the  
816 system by verifying that the collection id in the update matches the one config-  
817 ured for the system. This prevents installing an update meant for a different  
818 kind of device, or mixing variants. The updater also checks that the update ver-  
819 sion is newer than the one on the system, to prevent downgrade attacks where  
820 a older update with known vulnerabilities is used to gain privileged access to a  
821 target.

## 822 **Testing**

823 Testing ensures that the following system properties for each image are main-  
824 tained:

- 825 • the image can be updated if a newer update bundle is plugged in
- 826 • the update process is robust in case of errors
- 827 • the image initiates a rollback to a previous deployment if an error is de-  
828 tected on boot
- 829 • the image can complete a rollback initiated from a later deployment

830 To do so, a few components are needed:

- 831 • system update bundles have to be built as part of the daily build pipeline
- 832 • a know-good test update bundle with a very large version number must  
833 be create to test that the system can update to it

834 At least initially, testing is done manually. Automation from LAVA will be  
835 researched later.

## 836 **Images can be updated**

837 Plugging a device with the known-good test update on it bundle the expectation  
838 is that the system detects it, initiates the update and on reboot the deployment  
839 from the known-good test bundle is used.

## 840 **The update process is robust in case of errors**

841 To test that errors during the update process don't affect the system, the device  
842 is unplugged while the update is in progress. Re-plugging it after that checks  
843 that updates are gracefully restarted after transient errors.

## 844 **Images roll back in case of error**

845 Injecting an error in the boot process checks that the image initiates the roll  
846 back to a previous deployment. Since a newly-flashed image doesn't have any

previous deployment available, one needs to be manually set up beforehand by downloading an older OSTree commit.

## Images are a suitable rollback target

A known-bad deployment similar to the known-good one can be used to ensure that the current image works as intended when it is the destination of a rollback initiated by another deployment.

After updating to the known-bad deployment the system should rollback to the version under test, which should then complete the rollback by cleaning the boot count, blacklisting the failed update and undeploy it.

## User and user data management

As described in the [Multiuser](#)<sup>21</sup> design document, Apertis is meant to accommodate multiple users on the same device, using existing concepts and features of the several open source components that are being adopted.

All user data should be kept in the general storage volume on setups where it is available, as it enables simpler separation of concerns, and a simpler implementation of user data removal.

Rolling back user and application data cannot be generally applied and no existing general purpose system supports it. Applications must be prepared to handle configuration and data files coming from later versions and handle that gracefully, either ignoring unknown parameter or falling back to the default configuration if the provided one is not usable.

Specific products can choose to hook to the update system and manage their own data rollback policies.

## Application management

Application management on Apertis has requirements that the main update management system does not:

- It is unreasonable to expect a system restart after an application update.
- Each application must be tracked independently for rollbacks. System updates only track one “stream” of rollbacks, where the application update framework must track many.

Flatpak matches the requirements and is also based on OSTree. The ability to deduplicate contents between the system repository and the applications decouples applications from the base OS yet keeping the impact on storage consumption minimal.

---

<sup>21</sup><https://www.apertis.org/concepts/multiuser/>

## 881 Application storage

882 Applications can be stored per-device or per-user depending on the needs of the  
883 product.

884 An application may require storage space for personal settings, license informa-  
885 tion, caches, and any manner of long term private storage. These files should  
886 generally not be easily accessible to the user as directly modifying them could  
887 have detrimental effects on the application.

888 Application storage requirements can be divided into broad groups:

- 889 • An area for application exports to integrate with the system. This is  
890 managed by the application manager and not directly by applications  
891 themselves.
- 892 • User specific application data – for settings and any other per-user files.  
893 In the event of an application rollback, depending on the product this data  
894 may get rolled back with the application or the application needs to deal  
895 with potentially mismatching versions.
- 896 • Application specific application data – for data that is rolled back with  
897 an application but isn't tied to a user account – such as voice samples or  
898 map data. This data should be handled in the same way as user specific  
899 application data.
- 900 • Cache – easily recreated data. To save space, this should not be stored for  
901 rollback purposes, and should be cleared on a rollback in case applications  
902 change their cache data formats between versions.
- 903 • Storage for files in standard formats that aren't tied to specific applica-  
904 tions, as explained in the [Multiuser](#)<sup>22</sup> design, this storage is shared between  
905 all users. This data should be exempt from the rollback system.

## 906 Further developments

- 907 • Handling a larger threat model using [The Update Framework Specifica-](#)  
908 [tion](#)<sup>23</sup> / [Uptane](#)<sup>24</sup> with [Aktualizr](#)<sup>25</sup>
- 909 • Integrating with server side management services like [Eclipse hawkBit](#)<sup>26</sup>
- 910 • Hardware-assisted [verified boot](#)<sup>27</sup> with TPM/OP-TEE

---

<sup>22</sup><https://www.apertis.org/concepts/multiuser/>

<sup>23</sup><https://github.com/theupdateframework/specification/blob/master/tuf-spec.md>

<sup>24</sup><https://uptane.github.io/>

<sup>25</sup><https://foundries.io/insights/2018/05/25/ota-part-1/>

<sup>26</sup><https://www.eclipse.org/hawkbite/>

<sup>27</sup><https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot>

- 911 • File system-level integrity checks [Integrity Measurement Architecture](#)  
912 [\(IMA\)/Extended Verification Module \(EVM\)](#)<sup>28</sup>
- 913 • Add fail safe partition to handle file system corruption

## 914 Related Documents

915 A survey of system update managers:

- 916 • [https://wiki.yoctoproject.org/wiki/System\\_Update](https://wiki.yoctoproject.org/wiki/System_Update)

917 The OSTree bootable file systems tree store:

- 918 • <http://ostree.readthedocs.io>

919 The U-Boot Bootloader:

- 920 • <http://www.denx.de/wiki/U-Boot/WebHome>

921 The ChromeOS auto-update system:

- 922 • [https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-](https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate)  
923 [autoupdate](https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate)

---

<sup>28</sup><https://sourceforge.net/p/linux-ima/wiki/Home/>