System updates and rollback

Contents

This document focuses on the system update mechanism, but also partly addresses applications and how they interact with it.

Definitions

Base OS

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

Applications

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

Use cases

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

Embedded device on the field

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

The update process should be robust against power losses and low voltage situations, loss of connectivity, storage exhaustion, etc.

Typical system update

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

Critical security update

In the case of a critical security issue, the OEM could push an “update available” message to some component in the device that would in turn trigger the update. This requires an infrastructure to reference all devices on the OEM

side. The benefit compared to periodic polling is that the delay between the update publication and the update trigger is shortened.

Applications and base OS with different release cadence

Base OS releases involve many moving parts while application releases are simpler, so application authors want a faster release cadence decoupled from the base OS one.

Shared base OS

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

Reselling a device

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

Non use cases

User modified device

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

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

Unrecoverable hardware failure

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

Unrecoverable file system corruption

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

Development

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

Requirements

Minimal resource consumption

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

Work on different hardware platforms

Different devices may use different CPU architectures, bootloaders, storage technologies, partitioning schemas and file system formats.

The system update mechanism must be able to work across them with minimal changes, ranging from single-partition systems running UBIFS on NAND devices to more common storage devices using traditional file systems over multiple partitions.

Every updated system should be identical to the reference

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

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

Atomic update

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

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

Rolling back to the last known good state

If the system cannot boot correctly after an update has been installed successfully it must automatically roll back to a known working version.

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

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

Reset to clean state

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

Update control interface

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

Existing system update mechanisms

Debian tools

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

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

ChromeOS

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

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

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

- The OS installations are not deduplicated, the system stores the entire contents of the A and B installations separately, where as OSTree based systems only store the base system plus a delta between this and any update using Unix hard links. This means an update to the system only requires disk space proportional to the changed files.
- The A/B approach can be less efficient since it will need to add extra layers to work with different partitions, for example, using a specific layer to verify integrity of the block devices, where OSTree directly handles

operating system views and a content addressable data store (file system user space) avoiding the need of having different layers.
- Several partitions are usually required to implement this model, reducing the flexibility with which the storage space in the device can be utilized.

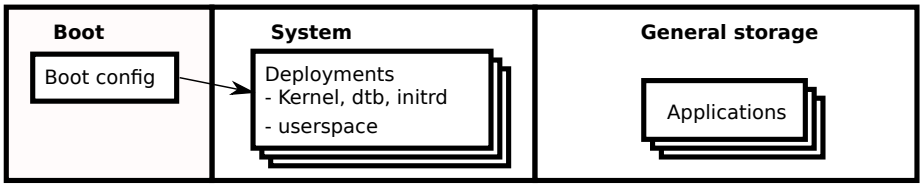
Approach

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

OSTree[1] provides atomic updates on top of any POSIX-compatible file system including UBIFS on NAND devices, is not tied to a specific partitioning scheme efficiently handles the available storage.

No specific requirements are imposed on the partitioning schema. Use of the GUID Partition Table (GPT[2]) system for partition management is recommended for being flexible while having fail-safe measures, like keeping checksums of the partition layout and providing some redundancy in case errors are detected.

Separating the system volume from the general storage volume, where applications and user data are stored, is also recommended.



GPT Partitions

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

Advantages of using OSTree

- OSTree operates at the Unix file system layer and thus on top of any file system or block storage layout, including NAND flash setups, and in containers.
- OSTree does not impose strict requirements on the partitioning scheme and can scale down to a single partition while fully preserving its resiliency

[1] http://ostree.readthedocs.io
[2] http://en.wikipedia.org/wiki/GUID_Partition_Table

guarantees, saving space on the device and avoiding extra layers of complexity (for example, to verify partition blocks). Depending on the setup, multiple partitions can still be used effectively to separate contents with different life cycles, for instance by storing user data on a different partition than the system files managed by OSTree.

- OSTree commits are centrally created offline (server side), and then they are deployed by the client. This gives much more control over what the devices actually run.
- It can store multiple file systems trees in a single repository.
- It is designed to implement fully atomic and resilient upgrades. If the system crashes or power is lost at any point during the update process, you will have either the old system, or the new one.
- It clearly separate the OS from the device configuration and user data, so resetting the system to a clean state simply involves deleting some directories and their contents.
- 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 GPG signatures and using 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[3] 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

[3] https://ostree.readthedocs.io/en/latest/manual/introduction/

- 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
- 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.

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[4] 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.

[4] https://ostree.readthedocs.io/en/latest/manual/atomic-upgrades/#atomic-upgrades

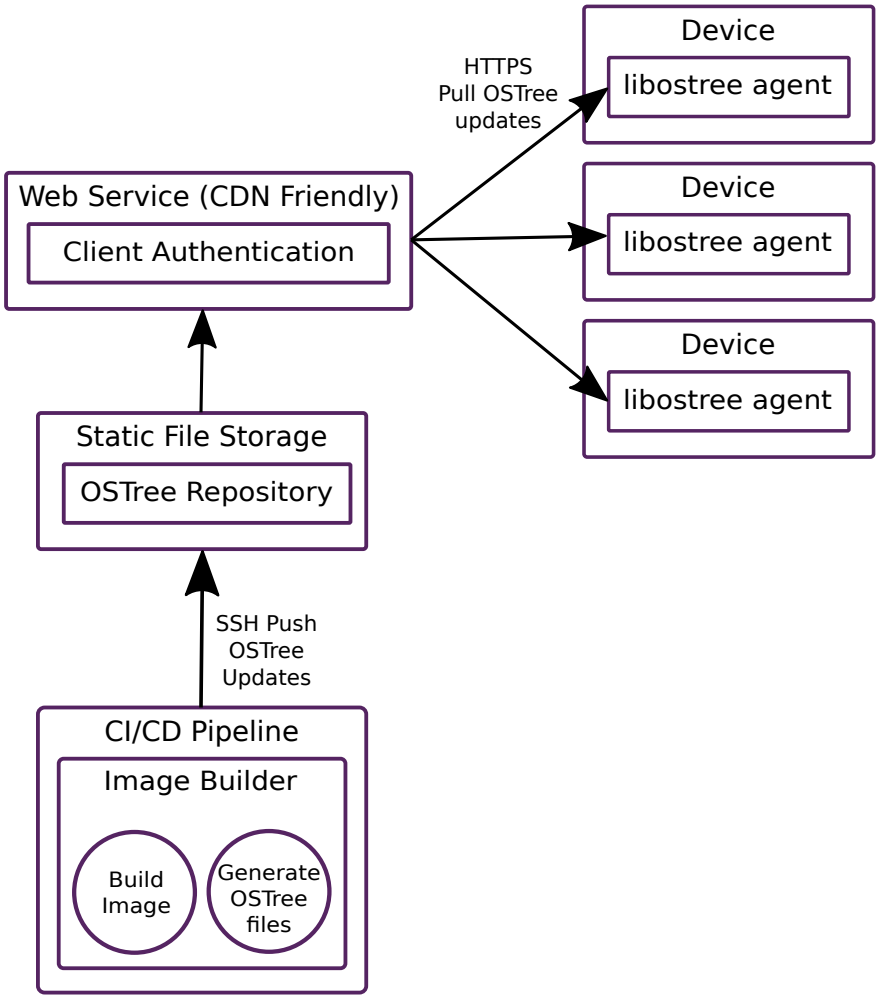
Online web-based OTA updates

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.

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



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

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

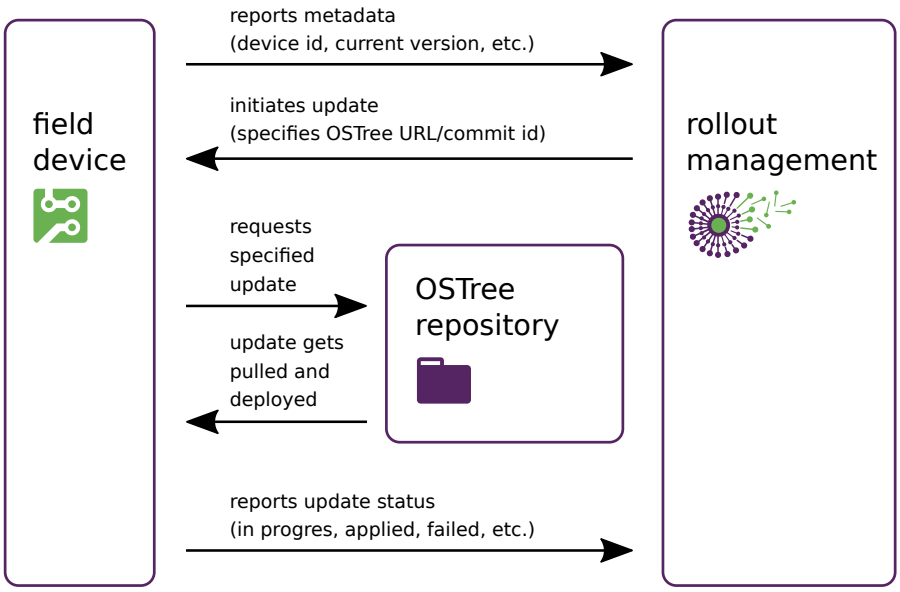
By only requiring static files, the web service can easily take advantage of CDN services to offer a cost efficient solution to get the data out to the devices in a way that is globally scalable.

The authentication to the web service can be done via HTTP Basic authentica-

tion, SSL/TLS client certificates, or any cookie-based mechanism that is most suitable for the chosen web service, as OSTree does not impose any constraint over plain HTTPS. OSTree separately checks the chain of trust linking the downloaded updates to the keys trusted by the system update manager. See also the Controlling access to the updates repository and Verified updates sections in this regard.

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

For instance, integration with roll out management suites like Eclipse hawkBit[5] can happen by disabling the polling in the OSTree updater and letting the management suite tell OSTree which commit to download and from where through a dedicated agent running on the devices.



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

Only the retrieval and application of the actual update data on the device would be offloaded to the OSTree-based update system, preserving its network and storage efficiency and the atomicity guarantees.

[5] https://www.eclipse.org/hawkbit/

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/v2020/armhf-uboot/minimal`. So for Apertis the "major upgrade" is technically considered as switching to another branch with a more recent Apertis version, for example `apertis/v2021/armhf-uboot/minimal`. 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:
  - apertis/v2021/{{$architecture}}-{{$board}}/{{$type}}
  - apertis/v2020/{{$architecture}}-{{$board}}/{{$type}}
```

this produce the commit compatible with version `v2020` allowing to install the new OS version `v2021` onto the board.

2. Set correct refs in repository

   After successful boot of the updated version all refs in `libostree` repository are still pointing to the previous branch due the nature of offline upgrade.

   This needs to be fixed for proper detection of further upgrades, including updates over the air. It is the responsibility of the update manager to update the refs once the update has been determined to be successful.

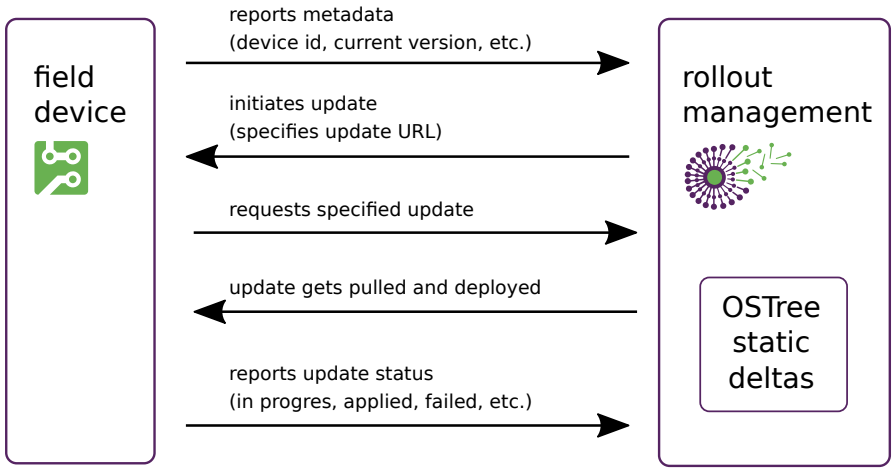
This functionality requires no changes to be made to previously released OSTree versions. The configuration that determines upgrade paths is held in the newer OSTree commit.

Apertis currently only supports upgrades to newer versions, downgrades to older versions of Apertis are not supported.

Online web-based OTA updates using OSTree Static Deltas

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

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



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.

Other mechanisms like IMA (Integrity Measurement Architecture) work instead at the file level, and thus can be used in conjunction with UBIFS and OSTree

on NAND devices.

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

Verified updates

Once a verified system is running, an OSTree update can be triggered. Apertis is using ed25519[6] variant of EdDSA signature. Ed25519 ensures that the commit was not modified, damaged, or corrupted.

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

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

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

Since OSTree is not capable to use PEM format directly, it is needed to extract the secret and public keys[7] from PEM file, for example:

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

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

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

On the client, ed25519 is also used to ensure that the commit comes from a trusted provider since updates could be acquired through different methods like OTA over a network connection, offline updates on plug-in mass storage devices, or even mesh-based distribution mechanism. To enable the signature check,

[6] https://ed25519.cr.yp.to/
[7] http://openssl.6102.n7.nabble.com/ed25519-key-generation-td73907.html

repository on the client must be configured by adding option `sign-verify=true` into the `core` or per-remote section, for instance:

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

OSTree searches for files with valid public signatures in directories `/usr/share/ostree/trusted.ed25519.d` and `/etc/ostree/trusted.ed25519.d`. Any public key in a file in these directories will be trusted by the client. Each file may contain multiple keys, one base64-encoded public key per string. No private keys should be present in these directories.

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

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

Offline update files with signed metadata

Starting with version v2020.7 `libostree` supports delta bundles with signed metadata[8], which allows to ensure that the whole delta bundle comes from a trusted source. Previous versions only allowed to assert the provenance of the commits in the bundle, leaving the metadata unverified.

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

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

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

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

Compatibility with upgrades

[8] https://github.com/ostreedev/ostree/pull/1985

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

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

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

Securing OSTree updates download

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

TLS can be configured in the remote configuration on the client using the following entries:

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

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

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

Controlling access to the updates repository

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

```
1    tls-client-cert-path
2                            Path      to      file      for      client-
3    side certificate, to present when making requests to
4        this repository.
5    tls-client-key-path
6                          Path     to     file     containing     client-
     side certificate key, to present when making
         requests to this repository.
```

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

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

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

Security concerns for offline updates over external media

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

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

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

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

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

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

Error handling

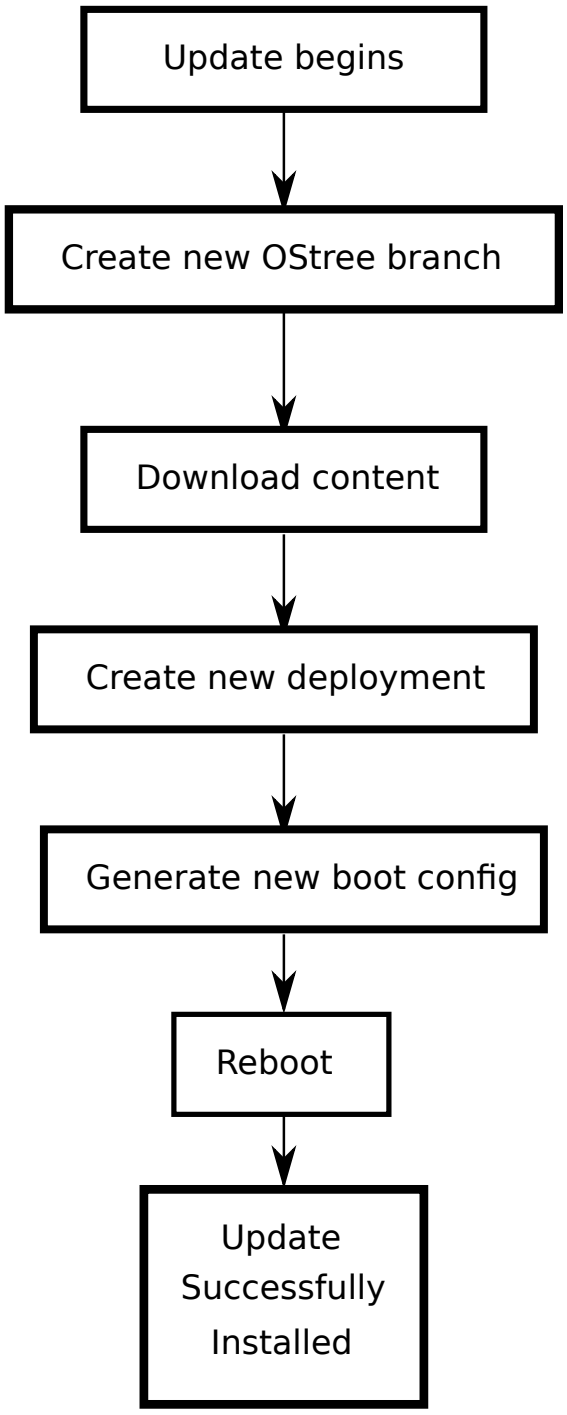
If for any reason the update process fails to complete, the update will be blacklisted to avoid re-attempting it. Another update won't be automatically attempted until a newer update is made available.

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

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

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

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


Update begins
Create new OStree branch
Download content
Create new deployment
Generate new boot config
Reboot
Update Successfully Installed

601

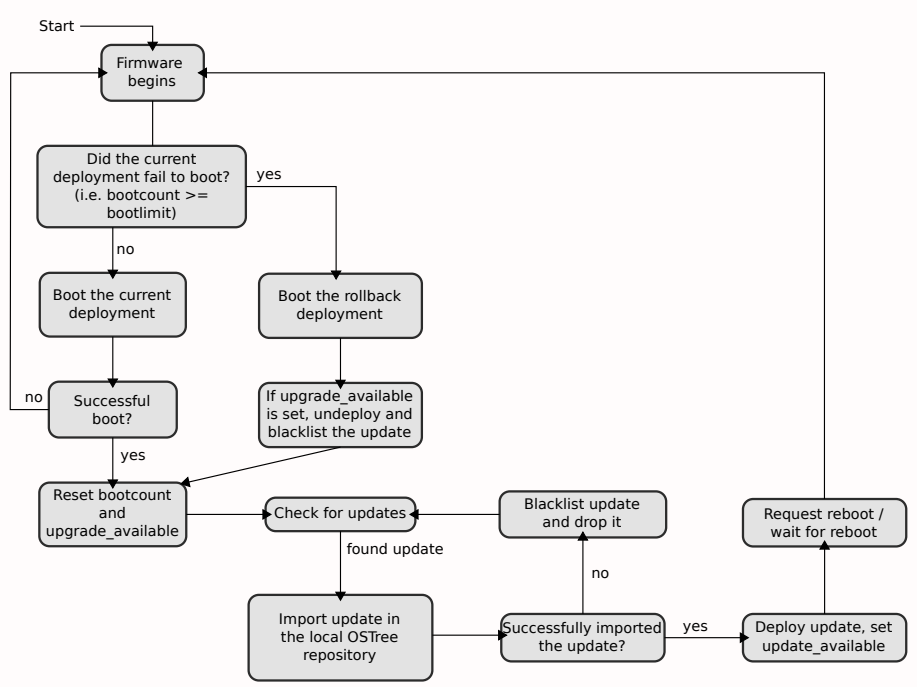
Implementation

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

- the updater daemon
- the bootloader integration
- the command-line HMI

The general flow

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



While the underlying approach differs due to the use of OSTree in Apertis over the dual-partition approach chosen by ChromeOS and the different bootloaders, the update/rollback process is largely the same as the one in ChromeOS[9].

The boot count

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

[9] https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate#TOC-Diagram

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

This boot counter needs to be handled in two places:

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

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

However, in the reference implementation the focus is on the most general solution first, while being flexible enough to accommodate other solutions whenever needed.

The bootloader integration

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

For the SabreLite ARM 32bit platform, integration with the U-Boot[10] bootloader is needed.

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

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

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

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

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

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

[10] http://www.denx.de/wiki/U-Boot/WebHome

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

The updater daemon

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

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

Detecting new available updates

For offline updates, the GVolumeMonitor[11] API provided by GLib/GIO is used to detect when a mass storage device is plugged into the device, and the GFile[12] GLib/GIO API is used to scan for the offline update stored as a plain file in the root of the plugged file system named `static-update.bundle`.

For online OTA updates, the `OstreeSysrootUpgrader`[13] is used to poll the remote repository for new commits in the configured branch.

When combined with roll out management systems like Eclipse hawkBit[14], the roll out management agent on the device will initiate the upgrade process without the need for polling.

Initiating the update process

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

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

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

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

Reporting the status to interested clients

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

[11] https://developer.gnome.org/gio/stable/GVolumeMonitor.html
[12] https://developer.gnome.org/gio/stable/GFile.html
[13] https://github.com/ostreedev/ostree/blob/master/src/libostree/ostree-sysroot-upgrader.c
[14] https://www.eclipse.org/hawkbit/

Resetting the boot count

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

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

Marking deployments

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

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

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

As a part of marking the deployment as successful the updater daemon checks the target branches from the OSTree commit metadata. If the booted deployment contains references to several branches, the updater daemon determines which branch has the highest version and resets all refs and the origin to that branch. Using the branches naming scheme used in Apertis, this could be considered as "major upgrade" of the system. From this point on, the system is fully switched to the new branch and accepts only upgrades created in the new branch.

Command line HMI

A command line tool is provided to query the status using the `org.apertis.ApertisUpdateManager` D-Bus API[15]:

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

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

[15] https://gitlab.apertis.org/appfw/apertis-update-manager/blob/master/data/apertis-update-manager-dbus.xml

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

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

Update validation

Before installing updates the updater check their validity and appropriateness for the current system, using the metadata carried by the update itself as produced by the build pipeline. It ensures that the update is appropriate for the system by verifying that the collection id in the update matches the one configured for the system. This prevents installing an update meant for a different kind of device, or mixing variants. The updater also checks that the update version is newer than the one on the system, to prevent downgrade attacks where a older update with known vulnerabilities is used to gain privileged access to a target.

Testing

Testing ensures that the following system properties for each image are maintained:

- the image can be updated if a newer update bundle is plugged in
- the update process is robust in case of errors
- the image initiates a rollback to a previous deployment if an error is detected on boot
- the image can complete a rollback initiated from a later deployment

To do so, a few components are needed:

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

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

Images can be updated

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

The update process is robust in case of errors

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

Images roll back in case of error

Injecting an error in the boot process checks that the image initiates the roll 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[16] 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

[16] https://sjoerd.pages.apertis.org/apertis-website/concepts/multiuser/

decouples applications from the base OS yet keeping the impact on storage consumption minimal.

Application storage

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

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

Application storage requirements can be divided into broad groups:

- An area for application exports to integrate with the system. This is managed by the application manager and not directly by applications themselves.
- User specific application data – for settings and any other per-user files. In the event of an application rollback, depending on the product this data may get rolled back with the application or the application needs to deal with potentially mismatching versions.
- Application specific application data – for data that is rolled back with an application but isn't tied to a user account – such as voice samples or map data. This data should be handled in the same way as user specific application data.
- Cache – easily recreated data. To save space, this should not be stored for rollback purposes, and should be cleared on a rollback in case applications change their cache data formats between versions.
- Storage for files in standard formats that aren't tied to specific applications, as explained in the Multiuser[17] design, this storage is shared between all users. This data should be exempt from the rollback system.

Further developments

- Handling a larger threat model using The Update Framework Specification[18] / Uptane[19] with Aktualizr[20]
- Integrating with server side management services like Eclipse hawkBit[21]
- Hardware-assisted verified boot[22] with TPM/OP-TEE

[17] https://sjoerd.pages.apertis.org/apertis-website/concepts/multiuser/
[18] https://github.com/theupdateframework/specification/blob/master/tuf-spec.md
[19] https://uptane.github.io/
[20] https://foundries.io/insights/2018/05/25/ota-part-1/
[21] https://www.eclipse.org/hawkbit/
[22] https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot

- File system-level integrity checks Integrity Measurement Architecture (IMA)/Extended Verification Module (EVM)[23]
- Add fail safe partition to handle file system corruption

Related Documents

A survey of system update managers:

- *https://wiki.yoctoproject.org/wiki/System_Update*

The OSTree bootable file systems tree store:

- *http://ostree.readthedocs.io*

The U-Boot Bootloader:

- *http://www.denx.de/wiki/U-Boot/WebHome*

The ChromeOS auto-update system:

- *https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate*

[23] https://sourceforge.net/p/linux-ima/wiki/Home/