Over The Air (OTA) Updates - OpenHarmony OS

1. Summary

Over The Air (OTA) updates is one of the core functionalities of products aiming for post-release support and maintenance. In the context of OpenHarmony OS, the update mechanism (or mechanisms) need be kernel agnostic (for the supported types of kernels) from the perspective of the user. We define here a “user” as being the operator of the device deployment and roll-out activities. Also, the same mechanisms need to provide a seamless experience to roll-out updates targeting different types of entities: a device, a fleet, a group, a fleet filter etc.

The OTA is one of the main functionalities that cross the software stack from the BSP to the OS, to the update client and up to the server endpoint. The high importance of this topic is driven by the fact that the project needs to have a robust post-initial deployment path for maintaining devices, fixing bugs in the OS, updating the OS with new functionality and deploying security fixes.

2. Motivation

The rationale behind this core component is, at its core, the acknowledgement that software evolves. This evolution can happen due to various factors: e.g. a regression in a certain functionality, improved or new functionality, security fixes. These evolutions are easily handled when the device is readily available, on one's desk, provided that there is a flashing (loading,

deploying) path for that specific device. The OTA comes to fill the gap where the targeted device, or fleet of devices, cross the boundaries of physical access and rely heavily on network connectivity. Devices in such an environment, need to run update software that can't rely on stable power and network access while still being robust and consistent with its operations.

It is worth emphasizing the fact that while software updates target various use cases, regular security fixes are standard for maintaining trust with the end-users. That is especially valid when the software stack is based on a variety of opensource components that in turn are tracked and provide security reports and fixes regularly. All these security updates need to be surfaced by the OpenHarmony OS software stack through its update mechanism. As an example, Android maintains a monthly security bulletin since August 2015 providing security fixes for its disclosed vulnerabilities.

2.1 Goals

The main goal for a successful OTA system is to be able to deploy software on devices in an "atomic", safe and robust way. This responsible team will focus on designing such a system, taking advantage of the extensive experience and expertise of the team.

Before addressing the update story and goals, the user experience starts with the board software loading or flashing. To satisfy this, there needs to be a clear and unified process to load the OpenHarmony OS software stack to a target in one of the following states:

- no software is loaded on the target
- OpenHarmonyOS incompatible software is running on the target

These software loading processes will assume physical access.

All software on the device needs to be updateable – subject to the manufacturer's exposed capabilities and also subject to the "update" design. For example, in an A/B system, everything needs to be updatable up the point of where the indirection happens. Beyond that point, the update can happen but under different operation requirements (for the accessible software blobs).

The OTA system must hide the target's hardware and software (kernel) specifics when providing the update process. From the perspective of deploying an update, doing that on target type A, or target type B should not affect the update experience (where A and B are any supported targets).

When handling update operations, the system must be able to target individual entities, groups of entities (fleets) and be also able to extend this to filtered groups.

The system will provide device telemetry data.

The update process needs to be modelled with a concept of "update strategy". This would allow a configuration of how the process is handled when deploying an update to multiple targets. These strategies will consider restricting factors (e.g. bandwidth, location).

Minimizing the transferred data when performing update actions is to be considered core functionality since IoT devices are usually under metered or pay-as-you-go data transfer channels.

The device support will provide a clear process allowing new targets to easily be adapted to the update platform.

The update system design will prioritize reusing existing opensource solutions that are in line with the system's requirements. When these components are identified, the team will approach the development process with an "upstreaming-first" mindset. Forks of opensource components are to be treated as staging areas while the upstream process takes place.

Security is another important goal of this update system and aims at designing a system that relies on proven security practices, leveraging existing cryptography solutions. In doing so, the strategy of storying, deploying and consuming update payloads needs to assume all the aspects of the security requirements a per the team's considerations.

2.2 Non-Goals

The OTA software stack is not aiming for 100% ability to update the device software stack due to eventual hardware limitations and robustness requirements.

Manual modifications on remote targets are not in the scope of this document.

Device OS metamorphosis (migrating an OS without support for the HarmonyOS update system to one that does) is to be treated separately if at all.

The team will minimize internal development while maximizing opensource components usage and contributions. With that as a goal, it means that a non-goal for this document is to develop an entire update system stack from scratch.

The OTA update system is only responsible through its design and implementation of the potential attacks and vulnerabilities to the update process itself. It doesn't protect or is concerned about the security vulnerabilities outside of the update process. For this kind of attacks, the OS needs to provide its security components and strategies. Even so, the update system will facilitate, provide a channel for addressing OS security concerns/issues by being the updating "vehicle".

2.3 Proposal

The update system is to be separated into three main components:

1. A server stack. This is the service provider of the update system including a management dashboard. It adheres to the redundancy and availability requirements under the defined cloud policy.
2. A device update client. This is a software component (or a set of software components) in charge of querying for an update, consuming the update and providing metrics to the backend.
3. The OS. The OS needs to provide a foundation for the update stack to ensure that the update system requirements are satisfied. This can mean stateless root filesystem, atomic bootloader switch etc.

There are three major challenges in designing this system.

The system must be able to provide a fallback and act accordingly based on update health checking.

The second challenge is about communication. The update payload must be sent to the device so the protocol must be accounted for in the software design.

The last challenge is security. The targets must be able to verify the update payload proving in this way that the server is a trusted party (authentication). While the payload is transferred, its content must be obfuscated as it might contain sensitive information (confidentiality). Also, the update payload has to provide means of checking its integrity on the client-side. This will allow detecting corrupted payloads transmitted over the network (integrity).

The key elements of an update system are:

- Power-cut and network instability resilience
- Scalability
- Security

IoT devices are usually deployed in environments where network and power are not assumed to be stable. The software stack needs to be able to cope with unexpected power cuts (and/or network access). To handle this, updates need to be atomic and durable. That also applies for any filesystem writes (at the level of filesystem entries) but it is vital for robust updates as a transaction.

Once there is a robust update system available, and a process to use it, the next important aspect is scalability. The system needs to handle an individual entity (one target) using the same update process as with a fleet entity.

Security is one of the core activities of an update system. In the scope of the update system, there are three main areas of concern:

- Server-side
- Client-side
- Tools

One the server-side, there are decisions to be taken in terms of where to host the instance(s), how to design and implement the communication in between the client (a target) and the server, user authentication and management, how to authenticate the devices and uniquely identify a target along with designing and implementing a logging and reporting mechanism for the update framework. Part of these decisions will also affect or shape the process of storing the update payloads and the key management of the target devices.

On the client-side, the communication to the server is one of the important aspects. It should be encrypted and provide the ability of the target to verify that the communication happens with the expected server endpoint while guarding against man-in-the-middle attacks. The way the devices store their keys is also an important aspect. This is because these devices are to be assumed in hostile environments therefore using hardware-based security keys is preferable. The client should also be able to verify the update payload as authentic and unmodified - digital signatures.

Lastly, on the tooling-side of security, the update system needs to provide primitives and processes to replace keys over time. Also, the systems processing the payloads and signing updates will need to be carefully designed while still being able to produce the required artefacts. Tools are to be selected to manage these secrets at the level of CI, build system and server endpoint deployment.

All these concerns are the primary focus of this update system.

With these principles, concerns and guidelines in mind, we can move on to defining update OS strategies.

The first update strategy, usually also the standard in the Linux-based embedded systems, is the dual copy update (A/B). It implies setting up an inactive partition before atomically switching the bootloader to use it at next boot. The main disadvantage is the fact that it doubles the OS disk usage and also, the OS and the application are treated as one update payload. The advantage is that we can update the entire OS in one go, as an atomic operation. This strategy can easily support implementing fallback mechanisms as the current inactive is usually the old OS (so a health fail detection on active can always fallback). On top of this strategy, we can use existing components like RAUC and SWUpdate along with existing management systems like HawkBit. Mender is also another solution but the management update server is not HawkBit compatible. using this strategy, the atomicity will be at the level of the root filesystem - image-based updates. The update can also treat as a transaction multiple filesystem updates in more complex scenarios.

The second strategy is container-based updates. The main idea here is that there is a separation in between apps and OS where the OS provides the means to run and manage the lifecycle of the application containers through an agent which in turn is managed by a cloud-based management system. Balena.io is a full-featured solution in this segment where each application is packaged as a Docker image, updates are deployed as Docker images which trigger the agent to respawn local containers. There is extra complexity on how to deploy a stack of components with dependencies and various extra deployment strategies but the main idea is that using this kind of system, there is a clear separation in between applications and OS in terms of updates deployment. There is extra care needed here because the container engine needs to also adhere to the robustness principles outlined above (eg. power-cut resilience). This strategy raises the transaction at the level of the container/application image guarding the atomic operation as the application container.

In the same container-based strategy, various other tools can be used to maintain and deploy updates – for example, “osTree” for delta generations and “runc” for running the containers. This can be combined to generate a Balena similar system without the complexity of a container engine like “Docker”.

The simplest strategy and the last one raised in this document (only mentioned for completeness) is a single copy update. This strategy handles the update while the system is unusable and also, it is hard to implement power failure resilience because the update partition is also the same one used to run the OS.

The strategies above are mainly targeting Linux-based system. This is because components like containers are not feasible for “edge” devices. Both ZephyrOS and FreeRTOS have OTA capabilities (implemented through various primitives) but to make the update experience simple, no matter the device type/kernel, the proposal is to consider a Linux gateway. Such a gateway device would act as an updater application, that is aware (can detect) of it’s connected MCUs and knows the primitives to act upon deploying updates.

On the side of the update payloads, the content will be defined in terms of its scope with implications on the risk and security aspects. A payload can be classified as:

- patch update: security fix, CVE fix, no impact refactoring etc.
- minor, major: updates of core components that break currently assumed ABI/API (compiler, kernel, etc) as opposed to "leaf updates" that only deal with downstream components where few dependent components might be affected.

These payload classifications can later be used as filters or "update rules" for managing risk and frequency/size requirements.

Update payloads will be minimized (or optimized) for speed and size. Both aspects are closely related but they target two different perspectives: speed provides a UX feature while size provides a functional feature where devices are restricted in terms of data transfer (e.g. targets over 5G network). There are various strategies in this regard:

- Use minimum deltas of only the strictly changed blobs in the payload (it will always be specific to the combination of running version and target version making it the heaviest on the delta generating side).
- Use optimized deltas where they provide changed blobs for a set of versions (easier on the delta generating server but heavier in size).
- Use update strategy restrictions at a per target or per group level. This means that the model should have a concept of "update rules" where an entity (target, group, fleet) can be enriched with metadata of types updates the entity accepts. A simple example is having a fleet that only accepts major updates while having another fleet accepting only security fixes. This approach has implications on the risk management side as well. A fleet "listening" only on

the "major updates" channel will take less risk as opposed to one listening to any update provided by the update server endpoint.

2.3.1 Story 1

Deploy an update for a specific device type. The system shall hide the deployment process making it device type agnostic.

2.3.2 Story 2

Starting from a single target, rollout an update payload to a fleet of devices. The update process will have a similar user experience if the update is targeting a constrained device running an RTOS, a fully-fledged system running Linux, or a fleet of multiple devices of the same type.

2.3.3 Story 3

The system can provide rollout strategies when targeting a fleet of devices. This can include staged rollouts, groups rollouts, “canary” deployments, etc.

2.3.4 Story 4

Device details and metrics will be provided in a unified dashboard regardless of the device type. Information like IP, CPU load, memory and various other metrics to be provided by the device and exposed by the management platform in a device-agnostic way.

2.3.5 Story 5

A device to which an unbootable update was deployed, will fall back to the initial software stack/version after trying (and failing) to boot.

2.3.6 Story 6

The dashboard will provide device logs.

2.3.7 Story 7

A target with no software running will be provided with clear documentation and tooling to successfully load and run the OpenHarmony OS software stack.

2.4 Notes/Constraints/Caveats

The general constraint in terms of achieving a high degree of update ability is usually in the hands of the board manufacturing. There will always be a balance in between the degree of update ability on one side and the robustness, atomicity and unification on the other side. In other words, the higher we raise the point of atomic switch, the more device will adhere to the same atomic interface but also more software elements are left hard to update behind the atomic switch.

2.5 Risks and Mitigations

To achieve close to 100% updatable software/firmware stack, there will always be risks in terms of what the hardware exposes and how those primitives are handling aspects like “atomicity”. When there are updates that can’t be handled atomically, the system can provide a way to flag the remote device of the risk and let the device/user proceed only when the environment is under close control. This should be maintained as an exception mechanism and only be used for exceptional situations (eg. non-redundant, non-atomic, bootloader firmware update due to a security vulnerability).

3 Design Details

The loading/flashing strategy is to be defined at the target level. For Linux based system there are proven strategies used by various existing solutions in production while keeping the approach as generic as possible. For example, once a bootloader can boot from an external storage device (or NFS), a flasher “initramfs” routine packaged with a target OS image, can proceed in loading the OS, providing visual progress when available and rebooting into the flashed system. For other targets, manufacturing tools are to be used when available – for example, “mfgtools”.

The implementation details should provide a stateless flashing mechanism such that the same flashing payload can be used on multiple targets. For example, a USB stick loaded with a “flasher” image, would not maintain state in between flashing processed so that each operation would produce the same outcome.

The stateless aspect is also important when designing the OS to facilitate various functionalities of the update system. A design separating the OS and the state of the device will make the update process simpler. This can be done using separate partitions leveraging symlinks, bind mounts or an overlay structure. This also enables consuming update deltas as opposed to always downloading a full software stack. There are more implementation details to be considered here: version-specific deltas as opposed to designing deltas applicable to multiple versions from the same baseline. These strategies and solutions are to be carefully considered by the responsible team.

3.1 Test Plan

Each update payload needs to be tested in terms of the supported devices being able to consume and also being able to fall back to the old version. The testing matrix needs to be able to validate all update paths included in the system.

3.2 Graduation Criteria

3.2.1 Alpha -> Beta Graduation

NA

3.2.2 Beta -> Stable Graduation

NA

3.3 Upgrade / Downgrade Strategy

NA

4 Implementation History

NA

5 Drawbacks

NA

6 Alternatives

There is a set of existing out-of-the-box solutions targeting the IoT market along with handling updates that are robust and scalable. Some examples are:

1. Balena.io
2. Mender
3. FullMetaUpdate

7 Infrastructure Needed

The system will need a server-side for storing, controlling and deploying updates. These services will run under defined software architecture to achieve targeted availability and scalability.

The OTA system also assumes communication infrastructure between the server and the client - in the scope of this document, the target. This infrastructure needs to be in line with the security decision taken by the team.

8 References

NA