Software distribution and updates

Contents

Apertis is a mature platform that is compatible with modern and flexible solutions for software distribution and software update. This document describes user-driven and operator-driven use cases, explores the challenges of each use case to extract requirements, and finally propose building blocks for software distribution and software update.

Terminology

Application and services

Application and services are loosely defined terms that indicate single functional entities from the perspective of end users. However each application may be composed of more than one component:

- system services[1]
- user services[2]
- graphical programs[3]

From the perspectives of software updates and software distribution applications and services can be deployed as part of the base operating system or separately as bundles[4].

Base operating system

The base operating system is the core component of the software stack. It includes the kernel, and basic userspace tools and libraries such as process manager, connectivity services, and update manager. Additional components like an application manager may be part of the base OS, depending on the intended usage.

Bundles

A bundle or "application bundle" refers to a unit that represent all the components of an Application or service. Comparing to mobile phones a bundle is similar to a phone "app", and we would say that an Android .apk file contains a bundle. Some systems refer to this concept as a package, but that term is

[1] https://jwd.pages.apertis.org/apertis-website/glossary/#system-service
[2] https://jwd.pages.apertis.org/apertis-website/glossary/#user-service
[3] https://jwd.pages.apertis.org/apertis-website/glossary/#graphical-program
[4] https://jwd.pages.apertis.org/apertis-website/glossary/#application-bundle

strongly associated with dpkg/apt (.deb) packages in Debian-derived systems, and it only partially captures the concept of a bundle.

The granularity is usually different between packages and bundles. Installing an application using packages is likely to involve multiple packages, while the bundle approach in our context goes in the direction of a single monolithic bundle that contains all components of an application. A bundle, unlike a package, offers atomic updates, rollback, insulation from the base operating system, insulation from other applications and configurable run time permissions for user data and system resources.

Docker images, Flatpak bundles, and Snaps are all examples of application bundles.

Software distribution

Software distribution is the process of delivering software to users and devices. It usually refers to the distribution of binaries of software to be installed or updated. However software distribution is more than a transport layer for packages an it can include authorization, inventory, and deployment management.

Software updates

The most common goals of an update are fixing bugs, removing security vulnerabilities, and adding new features to already installed software. Updating a software component may also involve updating the chain of dependencies of that software component.

Operator-driven use cases

The operator is an entity with the responsibility of ensuring that the devices operate within pre-defined specifications. A device can have more than one operator such as the manufacturer and the owner of the devices, and the operators, and not the device user, have powers to install, remove and update software on the devices.

Building access control devices

Access control is used to restrict access to a particular place, building, room, or resource. To gain access an individual generally needs to be given permission to enter by someone who already has authorization.

Automated building access rely on control devices to authenticate identity and to control physical locks. These devices use a variety of authentication methods such as smart cards, biometric data, and passwords, and can control access devices such as doors, gates and turnstile.

For most use cases, building access control devices are only the interface for more complex systems that include secure networks and servers. Building access control devices collect authentication data and send it to a server. The server then decides if the physical access should be granted, and send commands back to the device for informing the user and for controlling the lock.

Building access control devices have a critical mission. Failing to grant access to authorized personnel or granting access to unauthorized personnel can have serious consequences that can go beyond financial losses. Mission critical devices have strict reliability and security requirements, which include protection against tampering, resilience to user operation, and resilience to minor failures on the devices.

Both the manufacturer and owner may operate a large fleet of building access control devices. Large fleets are vulnerable to unintended changes on the software stack as it can introduce reliability and security issues. Low severity variability issues can be solved remotely, but high severity issues require manual intervention on each affected device.

Another problem for large fleet of devices is software deployment. Updates and new features should be deployed to devices on the field with minimal risk of rendering devices unusable. Operators require information about the software stack(installed software, version, etc) of each device to make decisions about how and when to do software deployment.

Device manufacturers offer on-demand development services. A new feature is developed for a customer and then is deployed only to the devices of that specific customer. Delivering the custom features requires conditional deployment capabilities based on business rules such as device owner and service level.

Robotic lawn mower

A robotic lawn mower is an electric autonomous robot that cuts lawn grass in a pre-determined area. Common features of robotic lawn mowers include finding the recharging base automatically, avoiding obstacles, and using advanced algorithms to cover the working area efficiently.

High-end robotic lawn mower are connected to the cloud to allow the owner to configure and control the unit using a convenient web interface. The owner, acting as the operator, uses a website to configure the schedule and settings of the mower such as the cutting height. Some models also allow the operator to remote control the mower.

Connected robotic lawn mowers receive over-the-air updates that are installed when the mower is not in use, respecting the schedule that was configured by the operator.

User-driven use cases

There are two categories of user-driven use cases. The first one is built on top of operator-driven use cases. In this category the device allow users to install and remove optional applications, but keeps the operator in control of system updates and system applications. In the second category the device is left under full control of the user, without any operator involvement.

Infotainment system

An infotainment system is usually an interface between users and a vehicle showing information about the vehicle and allowing the user to configure options such as interior lights and air conditioning. An infotainment system also provides additional functionality such as navigation, connectivity with the user's phone, music, Internet browser, and allows the user to install and remove applications.

An infotainment system can offer a personalized set of features for different models of vehicles and for different users. Premium features and applications are only available for owners of premium models of the vehicle and for users willing to pay for them.

The life cycle of an infotainment system can go beyond a decade, creating a challenging scenario for support and maintenance of the software stack. The vast majority of software components used in an infotainment system have a much smaller release cycle, with more than one release per year being common.

Releases are important for software components because only the latest releases receive security and bug fixes. Failing to keep the software stack using fairly recent components results in an infotainment system with bugs and security vulnerabilities.

On the other hand, as users interact with infotainment systems while driving, these devices are heavily regulated. The device requires an expensive certification process before deployment, and software updates are also subject to certification. So while updates are important for bug and security fixes, the structure and costs of certification of changes makes pressure against too frequent updates.

Another important actor in the infotainment ecosystem is the application developer. Empowering the application developer results in greater availability of applications and in faster availability of updates. Having more applications is a competitive advantage for the infotainment system, as users may prefer the infotainment system that has more installable applications.

Application developers need to be able to target as many different infotainment products as possible without being tied to the release cycle of each specific product. In other words, it is important for the developer to be as close as possible to have a single application that runs without changes in different infotainment systems and in different releases of infotainment systems.

This is particularly challenging as the very long lifecycle of infotainment products means that there are significant differences in the kind and versions of components shipped as part of the base operating system of different products. As such an application developer should be capable of releasing and updating applications independently from the base operating system, and should be able to conveniently create bundles that are optimized for a modern development flow.

The physical deployment characteristics of infotainment systems also complicate maintenance and updates. An unrecoverable failure due to an over-the-air update may force vehicle owners to pay a visit to the closest service center making customers unhappy, and potentially causing significant financial loss when the problem affects tens of thousands of vehicles.

And finally resilience to user operation is also a challenge to infotainment systems. Users should not be able to render the device inoperative, or make the device to operate outside its design specifications by continuous use, by changing configurations, or by installing/uninstalling applications.

Power and measuring tools

Power tools are electrically driven tools such as drills and grinders, with most models being powered by batteries. Measuring tools are electronic devices for measuring, or helping the user to measure, physical properties of the environment. Examples of measuring tools are wall scanners, thermo cameras, and laser measures.

Connected power and measuring tools can receive over-the-air updates and offer a convenient interface for the user to adjust operating parameters and to see the device status. The user can choose between a web interface and a mobile phone application to interact with power and measuring tools.

Non-use-cases

- Product development: during product development developers need to privilege flexibility over robustness. However robustness is of primary importance in production environment, and as such flexibility to ease development is not a use case.
- Workstations: while the mechanism described here are valuable on workstations as well, they are not the focus of this document.

Requirements

Conditional software deployment based on business rules

It should be possible to restrict the selection of software components that users and operators can install, remove and update based on business rules such as payment, customer, service level, and market segment.

It should also be possible for the operator to configure the deployment to adhere to business rules such as available time slots for maintenance, and to split complex deployments in batches.

Configurable access rights to user data and system resources

Applications should have limited and configurable access to system resources and user data. For example, applications should not be capable of taking screen shots, and the music player should have access to only specific files and folders.

Consistent state across devices

Maintaining a large fleet of devices requires the software stack of each device to be in a known state. Devices in unknown state are challenging to maintain and may present reliability and security issues.

Independent release and update of application domains

It should be possible to release and update application domains independently from the base operating system.

Operator-driven software distribution and updates

On operator-driven use cases, the operator should be capable of controlling the software distribution and update of large fleets of devices.

Protecting the fleet from software deployment issues

There should be mechanisms in place to prevent software distribution and software update issues, such as an update that renders the devices unusable, to affect the entire fleet of devices.

Resilience to distribution and update failures

Minor problems such as an update failure due to download problem caused by a network issue on the device side should not render the device inoperative and should recover automatically without intervention.

Resilience to user operation

User actions including installing and removing optional applications should not render the device inoperative, or make the device to operate outside its design specifications.

Software inventory

Operators require software inventory information such as installed software, and software version to make decisions about how and when to do software deployment. As an example when a security vulnerability is discovered, having an overview of how many devices are affected is important to determine the severity of the vulnerability, and to plan a response.

Tampering protection

Mission critical devices and devices subject to regulation require protection against unauthorized modification. Users should not be allowed to modify the devices to operate outside its design specifications.

Unwanted changes to the software stack

A common method of attacking a device consists in changing software that is installed or installing malicious components. Preventing unwanted changes on the software stack, and preventing non-authorized software to be installed eliminates an important attack vector: attacks that require changes to the software stack.

Updates rollback

Software updates should be reversible, and allow to rollback to a previous working state. This requirement applies to system software and applications.

User-driven software distribution

The user should be capable of installing and removing software components on user-driven use cases.

High level features

Before describing existing solutions it is necessary to group the requirements in features that are implemented by these solutions. One requirement may be related to more than one feature such as the requirement *Consistent state across devices* being related to the features *Immutable software stack* and *Atomic updates*.

Immutable software stack

- Related requirements: *Consistent state across devices*, *Resilience to user operation*, *Tampering protection*, *Unwanted changes to the software stack*

One solution to address these requirements is to make the base operating system and the application domains immutable.

Atomic updates

- Related requirements: *Consistent state across devices*, *Protecting the fleet from software deployment issues*, *Resilience to distribution and update failures*, *Updates rollback*

Updates on traditional package-based Linux distributions are prone to errors. An update usually involves multiple packages, and each package update can fail in ways that are not trivial to automatically recover from. After a failure on a package-based update, the limited rollback functionality is not guaranteed to revert the problem, leading to manual intervention.

A robust approach for updates that are capable of reliable rollbacks is called atomic updates. Atomic updates perform the file operations in a staging area, and the changes are only committed if the update is successful. When a failure occurs during an update, the changes are not committed and do not affect the file system.

However the benefits of reliable rollbacks are limited to changes made to the filesystem. Changes that are not file operations, such as updating the bootloader are not guaranteed to rollback gracefully.

Separation between system and application domains

- Related requirements: *Conditional software deployment based on business rules*, *Configurable access rights to user data and system resource*, *Consistent state across devices*, *Independent release and update of application domains*, *Resilience to distribution and update failures*, *User-driven software deployment*

These requirements are related to separating the base operating system from application domains in regards to software distribution, software updates, and execution environment.

Separating base operating system from application domains allow product teams to develop their products with greater independence, and offers more flexibility on how application domains are deployed, updated and executed.

Deployment management

- Related requirements: *Conditional software deployment based on business rules*, *Consistent state across devices*, *Operator-driven software distribu-*

tion and updates, *Protecting the fleet from software deployment issues*, *Resilience to distribution and update failures*, *Software inventory*

Software distribution is more than a transport layer for packages, it includes authorization, inventory, and deployment management. The software distribution infrastructure for traditional tools such as `apt-get` basically consists of static content providers that were designed to replace the previous method based on CDs and DVDs.

This infrastructure works well for transporting packages over the network, but it lacks features to implement business rules such as customer, payment, and hardware profile. On large fleets of operator-driven use cases, the operator need control over the deployment of updates and new features. It is responsibility of the operator to run the deployment in conformity to business rules to for example schedule a reboot in an appropriate moment, and to divide the deployment in batches.

The main component of a deployment management solutions is usually the backend infrastructure that interfaces with agents running one the devices. A common goal to deployment management is to offer easy and flexible rollout of software with monitoring of progress which is essential for large fleets.

Existing systems

OSTree for base operating system

OSTree implements for the base operating system *Immutable software stack* and *Atomic updates*. It also offers the underlying framework to allow *Separation between system and application domains*.

OSTree is a feature-rich deployment and update mechanism for files and directories in Linux. It offers transactional upgrades and rollback, is capable of replicating content incrementally over HTTP, support multiple parallel bootable root filesystems, and have flexible support for multiple branches and repositories.

As mentioned earlier, rolling out updates using package management tools such as apt-get is prone to a high degree of variability. Each update involves multiple packages, and each package update can fail on file operations and on scripts. Current package management systems have only limited roll back capability(See apt-btrfs-snapshot[5]) meaning that a failure during a package update can leave the system in an unknown state making it challenging to secure and maintain.

Failures during an OSTree atomic update are not committed, meaning that a failed update have no effect on the running system. If an OSTree atomic update completes successfully but introduces software issues, rolling back to the previous working version is guaranteed to work.

[5] https://github.com/skorokithakis/apt-btrfs-snapshot

However OSTree does not directly address the needs of application domains. For software distribution and update of application domains we recommend using either Flatpak or Docker.

Flatpak and Docker for applications

Both Flatpak and Docker implement for applications *Immutable software stack*, *Atomic updates*, and *Separation between system and application domains*. One requirement that is also addressed by both is *Configurable access rights to user data and system resource*.

Both Flatpak and Docker are mature and feature rich solutions for application distribution and update. They offer decoupling from the system, give the application developer greater freedom, give the user greater control, and run applications insulated from the system and from other applications. These are advantages when compared to more conventional packaging and distribution systems such as dpkg and apt-get.

Flatpak purposely focuses on user-level applications and services, or in applications with a GUI, such as the ones to be used on a infotainment system. Flatpak applications are shipped in bundles named Flatpaks, and it uses libostree under the hood to provide OSTree efficiency and robustness to application management.

Docker is instead better suited for non-graphical applications. Docker ships containers, and it is a good solution for applications that are developed and deployed as a collection of loosely coupled services. In some cases some sort of container orchestration is used with Docker, but orchestration is a topic that goes beyond the scope of this document.

Flatpak and Docker can fulfill similar roles for decoupling applications from the base OS, and there are use cases for both in Apertis. A case-by-case evaluation needs to be done to find the most suitable mechanism for each application and service. As examples, for the infotainment system use case Flatpak is better suited for the applications the user can install and remove. For the building access control devices Docker is a better fit for headless applications that collect identity data and controls locking mechanisms.

Eclipse hawkBit

Eclipse hawkBit implements *Deployment management*.

Eclipse hawkBit is a back-end framework for deployment management of edge devices. It can manage both the base OS and applications, and it is relatively agnostic about the kind of applications used. A preliminary investigation of the feasibility of the integration of the hawkBit-based Bosch Software Innovations IoT management suite with Apertis has been done with positive outcome.

Microsoft Azure IoT Edge

Eclipse hawkBit implements *Deployment management*.

Microsoft Azure IoT Edge is a full hosted suite to manage the deployment of Docker containers on edge devices and it also offer deployment management capabilities.

A preliminary Apertis image with support for Docker containers has been evaluated to explore the feasibility of using Apertis with Microsoft Azure IoT Edge.

Appstore

An appstore should meet the requirements: *Conditional software deployment based on business rules*, *Independent release and update of application domains*, *Protecting the fleet from software deployment issues*, *Software inventory*, *User-driven software deployment*. It should also provide support to the high level feature *Deployment management* or integrate with an external *Deployment management* solution.

An appstore is the interface that allow users to browse, buy, install, remove, and update applications on their devices. Users interact with an appstore remotely over a web frontend, and locally over an application on the device.

The appstore sits at the highest level layer of software distribution and update and reflects the decisions made for the lower layers. For example the solution for bundles and for deployment management highly impact the appstore design.

As an interface with the user the appstore verifies user credentials, presents the software catalog, and processes payment. As an interface with the deployment management layer the appstore queries the software inventory, and issue software distribution commands such as install an application on the user device.

Unlike an user, the operator is responsible for the health of a fleet of devices, and an appstore may not be part of the use case. Instead the operator uses an interface to change device configuration and to control deployment of updates and new features.

Curridge

Curridge is a custom non-upstream solution based on the Magento web commerce framework. At the moment Curridge has only been part of demonstrations done by the RBEI team, but Apertis currently ships a component to interface with it named Frome.

Collabora is not aware of the current feature set, but we expect that it is possible to adapt Curridge to ship Flatpak bundles. However more information is needed to compare the feature set with the requirements of an appstore.

An alternative path is to extend Curridge to interface with external solutions such as Flathub and hawkBit. This interfacing could allow Curridge to focus on the appstore user, and offload other tasks such as deployment management and bundle compatibility to dedicated components.

Flathub

Flathub is the upstream appstore for applications distributed via Flatpak.

It provides a validated workflow for third-party application authors to publish their work[6].

Applications can be browsed on FlatHub itself or through the on-device applications for app management, such as GNOME Software or KDE Discover.

Flathub does not support payments at the moment, even though there's upstream interest in the feature. It does not provide any remote management solution.

Summary of recommendations

- Use OSTree for the base operating system for *Immutable software stack*, *Atomic updates*, and *Updates rollback*.
- Use Flatpak or Docker for applications for *Immutable software stack*, *Atomic updates*, *Separation between system and application domains*, and *Configurable access rights to user data and system resource*.
- Use Flathub and Docker registry for storage and content delivery systems
- For operator-driven management, provide integration with hawkBit and Microsoft Azure IoT Edge
  - Open point: should Apertis provide a default hawkBit instance for testing and guidance for product teams?
- Evaluate the effort to extend Curridge to interface with Flathub and hawkBit.
  - Open point: Should Curridge handle deployment management or offload it to other solution such as hawkBit?
- For user-driven application management, use Flathub on the back-end, and either adapt GNOME Software or write a custom GUI application on top of Flatpak for the on-device user interface
  - Open point: Should Curridge be adapted to interface with Flathub?

Reference: System updates and rollback

The System updates and rollback[7] document contains details about technologies that are currently being used for software distribution and software update such

[6] https://github.com/flathub/flathub/wiki/App-Submission#how-to-submit-an-app
[7] https://jwd.pages.apertis.org/apertis-website/concepts/system-updates-and-rollback/

as OSTree. Consider reading `System updates and rollback` after having read this document.