Tampere University

Milan Mäkipää

COMPARISON OF OTA UPDATE FRAMEWORKS FOR LINUX BASED IOT DEVICES

Master's thesis
Faculty of Information Technology and Communication Sciences (ITC)
Examiner: Prof. David Hästbacka
November 2022

ABSTRACT

Milan Mäkipää: Comparison of OTA update frameworks for linux based iot devices
Master's thesis
Tampere University
November 2022

Remotely updating IoT devices introduces several issues concerning the device and the delivery of updates. The devices should be able to recover from failed updates to a functional state. Delivery of updates should scale with the size of the fleet in addition to the possibility of monitoring the state of the devices.

The complexity of the updates has an effect on how difficult these previous issues are to overcome. Only updating parts of the software is simpler and possible to implement in a way where the integrity of the operating system is not endangered. Updating the entire operating system makes it possible to deliver more overarching updates remotely, but failures in installation can endanger the integrity of the system.

This paper introduces an use case based on an existing IoT system and compares three update frameworks, SWUpdate, Mender and BalenaOS, for their suitability for providing remote updates. The comparison is first done based on relevant documentation and other literature and by further examining one with a proof of concept style prototype. This work only evaluates the frameworks in prototype use cases. In production use all of them would require additional work to integrate with the required security features.

Of the three frameworks, BalenaOS provides most of the required features while avoiding some of the shortcomings of the others. However some of the advantages it has over other solutions come with significant vendor lock-in. Additionally getting started with prototyping systems with BalenaOS is somewhat difficult, if additional libraries have to be added to the root operating system.

Keywords: Internet of things, OTA updates, Linux, Yocto, Mender, SWUpdate, BalenaOS

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Milan Mäkipää: Linux -pohjaisten iot-järjestelmien päivitystyökalujen vertailu
Diplomityö
Tampereen yliopisto
Marraskuu 2022

Päivitysmahdollisuuden lisääminen IoT laitteisiin luo haasteita sekä päivitettävään laitteeseen, että päivitysten toimittamiseen liittyen. Päivityksiä vastaanottavien laitteiden tulisi säilyä toimintakuntoisina, vaikka päivitysten asentaminen epäonnistuisi. Päivitysten toimitusketjun pitäisi myös skaalautua laitteiden määrän kasvaessa. Myös laitteiden etävalvontaan liittyvät ominaisuudet saattavat olla tarpeellisia.

Päivitysten kattavuus vaikuttaa edellämainittujen ongelmien ratkaisemisen haastavuuteen. Päivittämällä vain osan järjestelmän ohjelmistosta, voi päivitystyökalu olla koskematta sellaisiin käyttöjärjestelmän osiin, joiden muokkaaminen voisi vaarantaa järjestelmän toiminnan. Koko käyttöjärjestelmän päivittäminen mahdollistaa laitteen toimintojen laajemman muokkaamisen, mutta päivityksen epäonnistuminen saattaa vaarantaa laitteen toiminnan.

Tässä työssä esitellään käyttötapaus olemassaolevalle IoT järjestelmälle. Käyttötapausta varten vertaillaan kolmea päivitystyökalua, jotka ovat SWUpdate, Mender ja BalenaOS. Vertailu toteutetaan kirjallisuuskatsauksena saatavilla olevan dokumentaation ja muun kirjallisuuden perusteella. Yhden työkalun toimintaan tutustutaan tarkemmin toteuttamalla sillä prototyyppi päivitettävästä järjestelmästä. Työssä käytettyjä työkaluja vertaillaan vain kehityskäyttöön soveltuvassa ympäristössä. Mikään niistä ei sellaisenaan sovellu tuotantoympäristössä käytettävään laitteeseen, sillä tuotantoympäristössä esimerkiksi tarvittavien tietoturvaominaisuuksien integrointi työkaluihin vaatisi lisätyötä.

Kolmesta vertaillusta työkalusta BalenaOS kattaa parhaiten käyttötapaukseen liittyvät vaatimukset. Puutteita on myös muita työkaluja vähemmän. Työkalu on kuitenkin paljon muita vaihtoehtoja riippuvaisempi palveluntarjoajasta. Työkalun käyttö kehitysvaiheessa on haastavaa, sillä tarvittavat kirjastot on lisättävä käyttöjärjestelmään ennen sen asennusta kohdelaitteelle.

Avainsanat: Esineiden internet, Etäpäivitykset, Yocto, Mender, SWUpdate, BalenaOS

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

I want to thank Atostek for providing the opportunity for writing this thesis on an interesting subject and the possibility of using work hours for the writing process. I also want to thank my thesis supervisors, David Hästbacka from Tampere University and Ari Kettunen from Atostek, for guidance and feedback during the thesis.

Tampere, 11th November 2022

Milan Mäkipää

CONTENTS

1. Introduction . . . 1
2. Background . . . 3
  2.1 OTA updates instead of firmware patches . . . 3
  2.2 Conventional IoT and edge computing . . . 3
  2.3 IoT security risks . . . 4
  2.4 Risks of remote updates . . . 5
  2.5 Challenges of delivering updates remotely . . . 6
  2.6 Different approaches of recovery from update failures . . . 6
  2.7 Managing a fleet of devices . . . 7
  2.8 Monitoring . . . 8
3. Frameworks for OTA updates . . . 9
  3.1 Yocto Project . . . 9
  3.2 BalenaOS . . . 10
  3.3 SWUpdate and hawkBit . . . 12
  3.4 Mender . . . 14
  3.5 Alternative update frameworks . . . 16
4. Test environment requirements . . . 18
  4.1 Environment and use case description description . . . 18
  4.2 Hardware . . . 18
  4.3 Requirements and challenges . . . 19
  4.4 Beyond a proof of concept . . . 20
5. Comparison and analysis of the frameworks . . . 23
  5.1 Functional . . . 23
  5.2 Security . . . 27
  5.3 Scalability . . . 29
  5.4 Results . . . 30
6. Proof of concept . . . 32
  6.1 Previous work and goals for the proof of concept . . . 32
  6.2 Proof of concept setup and usage . . . 33
7. Results from proof of concept . . . 39
  7.1 Results compared to original requirements . . . 39
  7.2 Other notable findings . . . 39
8. Conclusion . . . 41
References . . . 43

Appendix A: local.conf 49

LIST OF ABBREVIATIONS

GPU	Graphics processing unit
IoT	Internet of things
OTA -updates	Over the air updates
SoC	System on a chip
SoM	System on a module
TPU	Tensor processing unit. An AI accelerator on the Coral dev board.
Yocto	A set of tools for creating embedded linux distributions

1. INTRODUCTION

IoT devices often run without much human interaction. This presents a challenge on delivering software updates to them. Over the air updates allow a device to be updated without interaction from the end user. OTA updates can vary from the delivery of new versions of pieces of software to completely replacing the firmware of a device. Installation of the updates can be triggered either by an end user or from a remote server.

Triggering the deployment of updates remotely allows for IoT devices to be placed in remote locations where constant physical access might not be feasible. It can also make pilot phases of projects easier. A pilot device can for example be tested in a real world environment, even while the software itself is still somewhat unfinished. As development progresses new versions of the software can be deployed to the device.

Many OTA update frameworks use a client server model. The client periodically checks with the server for updates and the server is responsible for providing those updates out to the appropriate devices. Some OTA frameworks make it possible to deploy updates to fleets of devices at once. These tools can also include ways of monitoring the devices.

Over the air updates present multiple challenges both on the device and the server side. The device might not be easy to reach physically, and as such failure in deployment of updates should not cause the system to be unable to function. Possible causes for such failures could be issues with the updated software itself and possible power loss during updates.

Additional issues might arise from the security of the update deployment pipeline. The client should be able to verify that the updates it receives are from trusted sources and the server should be able to separate different devices from each other. The updates might also have to be deployed to an encrypted device.

Monitoring the state of devices is required to know when an update can be deployed and if the deployments were successful. The device should be able to report it's state to the remote server.

The goal of this work is to compare three different update frameworks, BalenaOS, SWUpdate and Mender, for the purpose of performing OTA updates on an IoT device. These three frameworks were selected to be the focus of this thesis as they can be used for

updating the entire OS of a device while not risking the integrity of the OS if the update process fails. The comparison of the frameworks will mostly be based on relevant documentation regarding each framework. This is due to existing research, either comparing such frameworks or evaluating them on a use case, being rare and difficult to find. Instead existing literature will mainly be referenced when establishing the justifications and requirements for the use of OTA updates.

The most suitable tool will be further evaluated with a proof of concept that is specific for an use case based on a customer project. The work will attempt to answer the following research questions:

- Which of the three frameworks appears most suitable for implementing a remote update system for the use case? What are the shortcomings of each framework?
- With the proof of concept further evaluate, if the chosen tool can be used to produce a prototype that fullfills the needs of the use case.

This thesis will start with an overview of how OTA updates work and the related challenges. Section 3 will focus on introducing the different frameworks, their approaches to the challenges OTA updates and how they differ from each other. Section 4 will go through the use case and it's requirements. Section 5 will compare the frameworks to each other according to the requirements defined in section 4. After that the rest of the thesis will focus on a proof of concept to further evaluate one of the three frameworks in a real world use case.

2. BACKGROUND

2.1 OTA updates instead of firmware patches

Firmware updates that are done via physical medium have been used widely as a way of distribution. They can be done by the manufacturer updating each device separately or by the end user of the device performing the update themselves with the update medium provided by the manufacturer. However this approach doesn't scale well as the amount of devices in the field grows. Another problem with this approach is that the updates require reaching the device physically. If some devices in the fleet are not easily reached, it might leave them needlessly open to vulnerabilities that were otherwise easily fixed. With OTA updates found vulnerabilities could be patched for all devices en masse. [1, p. 2]

The networks, over which updates are delivered, can vary. One example would be a peer to peer system where devices are connected to each other directly. Alternatives might be networks with either connections to a central hub or a mesh like structure, where data is distributed between devices in the network. Delivering updates in methods other than via a direct connection is more challenging as a trustworthy connection must be established between two remote parties. [2, p. 7] The latter chapters in this work will focus on update systems, where there is a central server that provides updates to client devices over the internet.

2.2 Conventional IoT and edge computing

Conventional IoT systems have an end device and a cloud service, which does majority of the data processing for a system. The IoT device itself acts as a frontend to interact with the systems' environment via for example sensors and actuators. These type of cloud centric systems have various shortcomings that can be improved on with the use of edge computing. In edge computing parts of the responsibilities of the cloud can be moved either onto the IoT device itself or to nodes that are closer to the device. [3, pp. 3-5]

There are multiple benefits to offloading parts of the computing to the IoT device. Conventional systems are limited by the bandwidth between the IoT device and cloud, availability of a network connection and the latency of that connection. With edge computing for example in classification use cases the collected sensor data doesn't have to be relayed

onto the cloud. Instead the classification can be done on the IoT device and only the results have to be reported. Alternatively several individual IoT devices might be connected to a server, which does the computing, but is still a step away from the cloud and easier to reach by the device. [3, pp. 3-5]

Increasing the responsibilities of the IoT device also increases the attack surface and consequences of successful attacks by malicious parties. An edge node might handle data that was previously stored in the cloud. As a result in addition to all the sensor data that was collected by the device, the data moved from the cloud to the edge would be available to an attacker if they gained access to the device. [4, pp. 11] Such data might also include machine learning models that are used for data classification mentioned earlier.

As previously mentioned conventional IoT systems do majority of resource heavy computations in the cloud. [3, pp. 3] Updating the cloud components of such system is relatively straight forward as they rely on technologies such as virtual machines and containers. The amount of resources available in the cloud makes it possible to store multiple versions of for example container images. Replacing the currently running container image or migrating it to another datacenter is also possible. [3, p. 36]

Maintaining edge computing nodes is more difficult as some of the previous responsibilities of the cloud have been moved onto the edge devices. The edge devices are often more resource constrained than their cloud counterparts and as such deploying updates can be more difficult. Edge computing also introduces another challenge of discovering and categorising the devices to recognize which devices an update should be deployed to. The deployed software must be fit for the device in cases where there are multiple different devices with similar but somewhat differing capabilities. [3, pp.35-36]

2.3 IoT security risks

Compared to offline only devices, IoT devices are connected to internet and that makes them subject to additional attack vectors. Additional security risks are introduced by the complex nature of IoT systems. The importance of ensuring the security of such devices can be partially justified by their use cases. As an example IoT systems are common in healthcare applications, where the compromising on security is unreasonable. [5, pp. 1-3]

IoT systems face threats both on the frontend device and in the network it uses. Attacks on the physical device can vary from compromising a device to replacing it entirely. In both instances the attacker could gain some level of access to the wider system or simply feed malicious data to the system. Erroneous data could be used to affect the results the system produces or a malicious device could spam the network with data to create a

denial of service attack. [6, p. 59]

The network level threats of an IoT system can include eavesdropping, where an attacker could try to intercept unencrypted data from for example a wireless communication. Other attack methods can be man in the middle attacks, where the malicious actor tries to insert itself between a device and the node it's communicating with. While the devices themselves can be used to create denial of service attacks, the network level is also subject to them. Flooding a node with requests can affect the availability of that node to the other nodes in the system. [6, pp. 59-60]

An example of possible consequences an IoT device getting infected with malware is the Mirai botnet. The attack relied on a piece of malware propagating through unsecurely configured IoT devices. Each infected device scanned random IP addresses for possible targets and attempted to access those with a predefined set of credentials. When access to a new device was gained it was reported to a remote server. A separate central server then instructed the new devices to download a version of the malware suitable for the specific system. That server could also trigger infected devices to commence distributed denial of service attacks. [7, pp. 80-82] The significance of this botnet stems partially from it's size spanning hundreds of thousands of devices. In 2016 it was used to create, at the time, the largest recorded DDoS attack with a capability of around 1.2 Tbit/s. [8, pp. 1-2]

Updating the software of IoT devices frequently can be one method of keeping the device secure. [9, pp. 223-224] Additionally the functional requirements of a device might change over time, and updates allow for the software to keep up with the changes. [10, p. 55]

2.4 Risks of remote updates

Adding update capabilities to devices also brings some additional risks. The worst case scenario is that failures in the update process make a large amount of devices unusable. Some of the possible causes include the possibility of power loss while the update is being installed, the possibility of network connection failure during the update, or running out of storage space. Additionally some devices might be using different versions of the various pieces software on the device, which could lead to incompatibility issues when one part of the system is updated. [11, p. 133]

The update process also increases the attack surface of the device. For example if an attacker can make the device upgrade or downgrade the software at will, they might be able to switch the firmware to a specific version that is more vulnerable. [11, p. 131] The update system can also be vulnerable to eg. Man-in-the-Middle attacks. [12, p. 3] With those a device might be sent counterfeit messages to get it to act in the interests of the attacker. Another attack method might be spoofing, where a non trustworthy part of the

system attempts to gain access by posing as legitimate. [13, p. 87]

2.5 Challenges of delivering updates remotely

The challenges of delivering updates can be split roughly into two categories: data and service integrity. Data integrity means ensuring that the device cannot be updated with new software from an unauthorized party. Service integrity means that accessing the update provider, downloading and installing of the software is done without tampering. [12, p. 3]

Data integrity can be verified by checking that the update was created by a trusted party. This can be achieved with the usage of digital signatures and by checking the hashes of the incoming update files.[12, p. 4]

Additionally the updates have to be protected to avoid theft of intellectual property or other sensitive data. One method of achieving this at a large scale is using asymmetric cryptographic keys to encrypt updates. However this also introduces the issue of storing these keys securely on the end device. One such approach can be the usage of a trusted platform module or other hardware security modules to store the necessary keys. That way the encryption keys don't have to be accessible directly by the operating system. [14, pp. 3-5]

Other challenges include that the complexity of the software on an IoT device might require updates to be installed to various different components on the device. Updating a one component on the device might cause unexpected behaviour in others. To solve this issue individual component updates should be tracked independently to detect risks of updating only parts of the system. In the other hand updates might be delivered by different providers. As such a device might have to be able to verify updates from various sources. [15, p. 2]

The hardware resources taken up during the installation of updates might temporarily affect the operational capabilities of the device. In such cases it might be desirable to delay the installation until the load and availability requirements on the device are low. [15, p. 2]

2.6 Different approaches of recovery from update failures

Considerations must be made for recovery options in case of failed updates. The device should be able to recover back to a functional state in case an update has failed. Additionally installing updates might introduce some downtime in the operation of the device. This can happen both while an update is being installed or if the device needs to reboot to finalize an update.[10, p. 55-56]

Failure tolerance for updates can take multiple forms. For example in a Linux based system one option would be to only update individual files of noncritical applications and as such ensure that the core operating system remains untouched. An example of such method would be Mender's Update Module system which only updates individual files or directories, but doesn't touch the kernel or devicetree.[16] However in such case updating the entire system would not be possible.

To allow for safe updates to the core operating system, there has to be a foolproof method of recovering from a failed update. Such approaches can be either symmetric or asymmetric. Asymmetric systems have a regular operating system held on a one disk partition and an additional minimal recovery system on a separate partition. System updates are applied to the regular operating system, but in case of update failure, the recovery system remains functional and it can still repair the main operating system. [17, p. 4988]

Symmetric systems have two root partitions. Both contain a functional operating system, but only one is active at a time. An update is applied onto the inactive partition. After an update the labels of active and inactive partitions are flipped in the bootloader and the device reboots. If an update is successful, the device will boot from the updated partition. If the device fails to boot, the still intact older version can be used as a fallback. This system however will require twice the disk space of a regular operating system installation. [17, p. 4988]

2.7 Managing a fleet of devices

In addition to simply delivering updates to IoT devices the entire fleet of devices can change over time. As such the management system for those devices must be able to do many other things in addition to simply delivering new firmware versions. Such additional responsibilities include provisioning new devices or decommissioning them, monitoring and tracking the software running on the device and configuring the devices. [18]

Provisioning devices with unique certificates makes it possible to differentiate them from each other. This is necessary, as devices can have user specific needs that require modifying a configuration on device basis. The ease of configuration has to be streamlined to minimize work per device to allow for operation in a large scale. Some type of authentication is also necessary to prove that a device is trustworthy. It also helps with verifying that data provided by the device has not been tampered. [18]

Many of the update tools that are discussed later in this work not only implement the update management part of the IoT device management. Instead they include some or all of the above device management aspects. For example many of the tools include methods for provisioning new devices and methods of authenticating devices with a remote server.[19]

Additionally some update systems have tools for managing fleets of IoT devices beyond just triggering updates remotely. These can include phased rollout of updates, in which a group of devices is updated in consecutive batches. For example an update could first be deployed to a small group of devices. If the update is successful, it can then be automatically deployed out to the rest of the fleet. This way possible software bugs can be caught before they are delivered to many devices.[20] Alternatively some systems also allow for parts of a fleet to be held back on a specific version of the software. Depending on the environment this might also be an useful feature.[21]

2.8 Monitoring

A software that is capable of IoT device monitoring can provide information on the current state of the device. This can include the level of resource usage, firmware state or version and the state of applications that are running on the device. [22, p. 1163] The status of the device can also be used to monitor the success of update deployments. This can be beneficial for example when using a phased rollout of updates.

Monitoring can also be used to respond to attacks on IoT devices. By monitoring the device closely one can detect abnormal behavior and potentially respond to that by deploying security fixes. [22, p. 1163] Alternatively when it is detected that a device has been compromised, it can be isolated from the network. [2, p. 7] For example some update frameworks allow the access keys of devices to be revoked, so that they won't have access to future versions of the firmware. [23]

3. FRAMEWORKS FOR OTA UPDATES

This chapter will give an introduction on the three update frameworks, that were selected for comparison. Those frameworks are BalenaOS, Mender and SWUpdate. They were selected to be the focus of this thesis as they can be used to deploy OS updates to a device remotely. Additionally each of them has some mechanism for recovering from failed updates by using redundant file systems. [24] [25] [26] All of the frameworks are based on Yocto project, so that is introduced first to give more context on how the frameworks work. The last section also describes an alternative framework that could be used for deploying updates. However compared to the other frameworks it has some shortcomings that will also be described. The comparison of the frameworks will be done in chapter 5 after introduction of the use case and requirements in chapter 4.

3.1 Yocto Project

Yocto project is used as a base by several OTA frameworks. For example Both Mender and SWUpdate can integrate with an operating system image generated with Yocto. [27] [28] Yocto project provides tools for generating Linux distributions that are compatible with a broad variety of hardware architectures. Yocto is flexible in that software components can be reused and layered on top of each other. Another major benefit of using Yocto is it gives its user control on what features they want to add to the distribution build. For example a graphical user interface could be foregone if the device has no need for it. [29] Only installing the required software can save disk space and other hardware resources.

The tools of Yocto project can be used to produce a bootloader, Linux kernel and a root filesystem that are compatible with the target architecture. Poky is a subset of Yocto project that provides the tools for managing the various sources of software that are to be built. Additionally it includes a task runner Bitbake, which used to manage the build process. [30, pp. 6-9]

With Yocto, pieces of software are introduced to the operating system via recipes. Usually a single recipe will contain instructions on how to install a single application and how do the necessary configuration for that software. [31] Recipes are grouped together into repositories called layers. They are used to isolate irrelevant parts of software from each other to allow reuse and also simplify modifying existing software. An example of a layer is

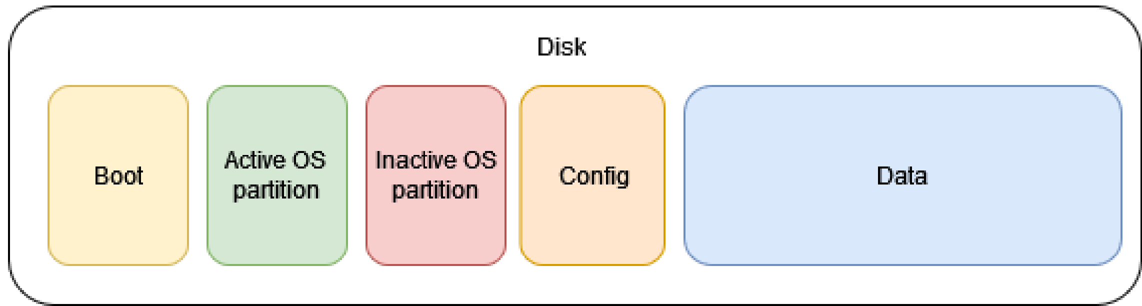

Figure 3.1. *Balena partitioning layout.*

a board support package (bsp), which can be provided by a hardware vendor. Such layer can include changes and configurations, that are needed to support a specific platform. Other software can be introduced via other layers and can be reused for multiple different platforms. [29]

Each of the frameworks, Mender, SWUpdate and Balena use Yocto project to provide layers to introduce necessary modifications for them to work with specific device types.[27] [28] [32] For example Mender provides a meta-mender layer, which makes modifications to the distribution's bootloader, kernel and partition layout. [27]

3.2 BalenaOS

BalenaOS is a combination of several tools that create a framework for delivering updates both for the operating system and user applications. User created software is meant to be run in docker containers, which isolates them from the operating system. This should also allow for containers to be reused on different platforms. [24]

The remote server is provided either by balenaCloud, which is a commercial service or via openBalena, which provides a more restricted version of the cloud platform for on premises hosting. [33]

3.2.1 Update mechanism

Updating BalenaOS is separated to updating containers and the underlying operating system. Operating system updates have no effect on the containers due to the system partition layout, which separates the OS and container images. The partition layout is described in figure 3.1 [24]

Operating system updates make use of a symmetric root partitions. When an update is available it's downloaded to the inactive partition and the bootloader switches to boot from that partition. This ensures that in case of a failed update, the system retains the previous OS version, which it can revert then back to. [24]

User applications are delivered to the device as docker container images. The operating system runs a Supervisor, which checks for and downloads new images onto the device when needed. When a container is replaced with an updated one, it can be stopped either before or after the new container image has been downloaded. This allows for either minimal downtime by having two images stored in the device at once or saving of storage space by removing the old one before fetching the newer one.[34] [24]

Balena supports persistent storage for data via docker named volumes. Inside the container a program can access the volume from the /data directory. Volumes can be shared between multiple containers and they are kept intact when container images are updated.[35] They are also kept in a separate partition, so operating system updates don't affect them either.[24]

3.2.2 Security and authentication

All connections to the remote server are initialized by the device itself. As such using BalenaOS doesn't require opening ports on the device for incoming network traffic. Additionally communication between the device and server is TLS encrypted.[23]

New devices are initialized with a provisioning key, each of which is only valid for a single use. That key allows them to request an API token from he backend server. The token then allows for the new device to initiate new connections with the server. All updates and software modifications are triggered by the server. Additionally each new device receives an unique API token, and they can be revoked remotely. As such a compromised device can be removed from the fleet.[23]

3.2.3 Cloud integration

There are two options for interacting with the balenaCloud services. The first is balena CLI, a command line interface, that can be used to push new images to the cloud. The second is the cloud dashboard, from which devices and updates can be managed with a web browser. [36]

Prebuilt operating system updates are downloaded directly from the cloud as they are both published and hosted by balena. Distributing new versions of docker images on to devices is also done by the cloud service. There are multiple ways of pushing new docker images to the cloud. [36]

First method is done by using the balena CLI to push a project directory to balenaCloud. The directory root has to have either a Dockerfile or a docker-compose.yaml -file in it. The cloud server will use these files to build a new image or a set of images. The second way is to build the images locally and push the built image to the cloud. The third way is to

add the balenaCloud server as a remote Git repository. After that git can be used to push the project files to the cloud build server. [36]

The cloud dashboard allows separation of devices into fleets. When an update is pushed to the cloud, it is automatically deployed to the target fleets. If a device or fleet should not be updated, they can be pinned to a specific software version. [21]

The balenaCloud also provides remote access to every connected device with a recent enough version of the operating system. The devices can connect to the cloud to create a reverse tunnel for a ssh connection. As such the connection doesn't require opening ports from the device's firewall. Alternatively a traditional ssh connection to the device can be made, but that requires the corresponding port to be open and adding the client's public key to the device. [37]

An open source but more limited version of the cloud service is available via openBalena. It allows the user to host their own instance of the remote server. However it's missing many of the features of the cloud platform. One of these is the web dashboard, which provides a browser management view for the server. Instead it can only be used via a command line utility. openBalena provides the core functionality of the cloud service, which allows the user to initialize new devices and deploy updates to them. [38]

3.3 SWUpdate and hawkBit

SWUpdate is an open-source project that provides a very flexible approach to updating Linux based devices. It's can perform updates to a device from either local mediums such as SD-cards or remotely over the internet. The flexibility of the system stems from the ability to create and replace modules for various parts of the update process. For example proprietary hardware etc. can be supported by creating a new update handler. These handlers are called by SWUpdate when needed during the update process. [39]

SWUpdate has two major parts that can be used to implement remote updates. The first is using SWUpdate in a mode called Suricatta, where the SWUpdate software will query a remote server for available updates. When new updates are found, they are downloaded and installed on the device. It's also responsible for tracking and updating the state of the bootloader to ensure that the system can recover from failures.[40] The default remote server for the system is hawkBit. It's a service by Eclipse, that provides the ability to roll out updates and monitor devices remotely. [41] hawkBit can also connect to a completely custom remote server, but that would require implementing the required interface for the backend or modifying the hawkBit to function with the new remote server[40]

SWUpdate supports the usage of both symmetric and asymmetric partition layouts. With the symmetric layout the system has four partitions: bootloader, primary operating system, secondary operating system and persistent data. By default the bootloader loads

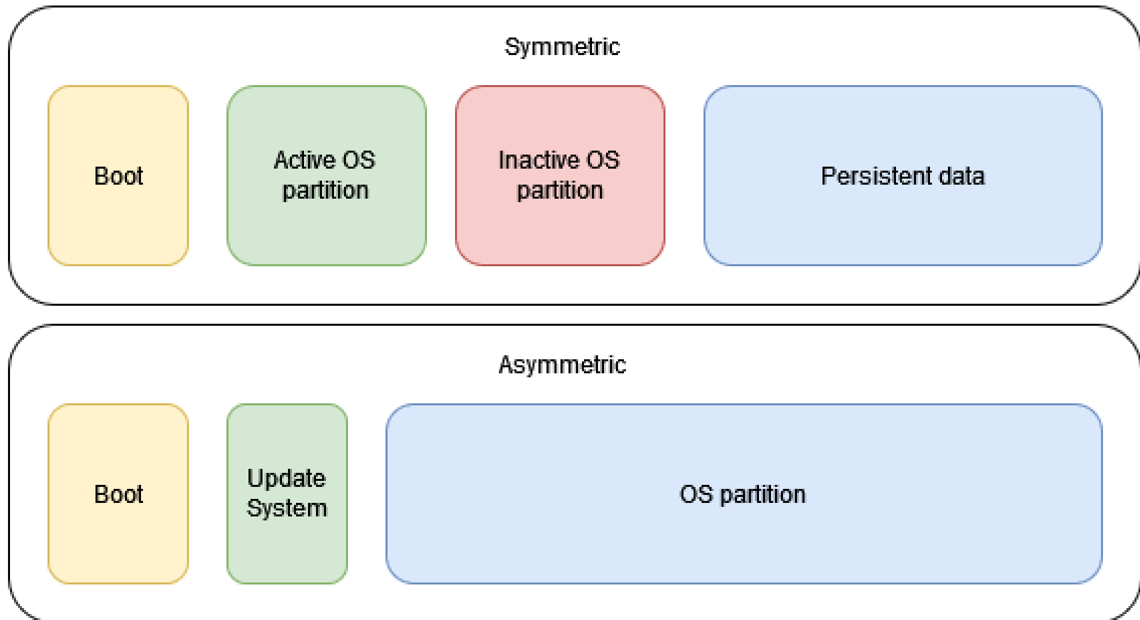

Figure 3.2. *The two available partitioning layouts on SWUpdate.*

the operating system from the primary partition. When an update arrives, it is installed on the secondary partition, and on the next reboot the bootloader will flip to read that one instead. In case of a failed update the bootloader can always revert back to the previously active partition. The last partition is for persistent data that is not modified by system updates. This approach is similar to those that both Mender and BalenaOS use for system updates. [26]

The possibility to use an asymmetric partition layout is one factor, where SWUpdate differs from other update tools. It might be needed for example when the amount of storage space on the device is not enough for hosting two entire filesystems at once. This approach uses three partitions: one for bootloader, one for an update system and one for the actual operating system. The update system is a minimal linux kernel, that includes the necessary software to update the operating system partition. By default the bootloader loads a kernel from the operating system partition. When an update is triggered, the system must reboot to the update system partition, which can fetch the update. After the update has finished, the system can again reboot to the new operating system. This approach has no hot swappable fallback in case of a failed update. However the system can still reboot to the update system, and wait for a new update.[26] Both partitioning approaches are described in figure 3.2

3.3.1 Backend (hawkBit)

The default backend server supported by SWUpdate is hawkBit. It's a completely separate project and agnostic to how to the client performs the update installation. Instead its purpose providing and managing updates and devices. [41]

hawkBit has functionality to to deliver updates to groups of devices at once. A feature that seems to be unique to hawkBit is the ability to perform cascading update rollouts. This means that after an update has been delivered to a group of devices and installed successfully, it can be automatically delivered to other larger groups of devices. This staged rollout can be configured to stop automatically in case of some milestones aren't met. For example if a certain percentage of devices in a group doesn't update successfully. This should allow catching erroneous updates before they have propagated to a large amount of devices. [20]

3.3.2 Security features

SWUpdate has a feature to determine that the received updates are from a trusted source. It's is based on signing updates with a private key before publishing them. The devices can then verify the authenticity of the updates with a public key. Signing entire update images might not be feasible, as the signature cannot be verified until the entire image has been downloaded. To solve this issue each update can contain a description file. If the update is provided in a set of smaller sub-images, the hashes of those images can be stored in the description file. That file itself can then be signed and verified. When each sub-image is downloaded, they can be hashed and compared to the known valid value. [42]

The SWUpdate documentation didn't state any authentication policy for differentiating between devices or how new devices are added to a fleet.

3.4 Mender

Mender is a remote update framework that uses a similar client-server architecture as balenaOS. The device runs a client program that polls the remote server for updates periodically. New updates are published to the remote server, which then informs the corresponding clients that a new update is available. It also supports updating devices from a physical medium without connection to a server. In such case the updates would be triggered by an user of the device. However this mode of operation is not relevant to the use case. [25]

For remotely managed updates, Mender supports two types of update mechanisms, application and full system updates. Application updates modify individual files or directories. System updates use a symmetric partitioning layout to update an inactive partition with a new version of the operating system. [25]

The standalone Mender client software can be installed on Debian based systems. The client contains tools that are necessary for converting an existing Debian based operating system image into one that has the redundant partitioning layout. Alternatively mender

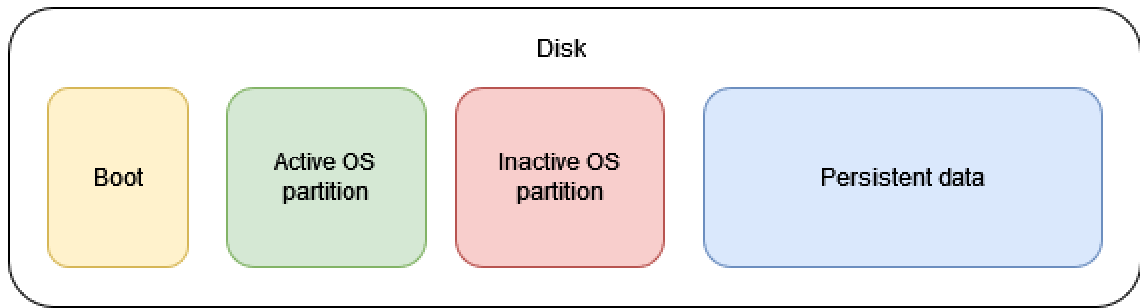

Figure 3.3. *Mender partitioning layout for use with system updates.*

provides a set of Yocto layers for integrating with a system or use case that cannot use a Debian derived operating system. [43] [44]

3.4.1 Update process

This work focuses on updating the entire system and not just parts of the software, so with Mender only the full system updates are relevant and described. Creating a system update with Mender requires a source system, which is in the state that the update should represent. The update consists of a snapshot of that system, which contains the state of the active OS partition. The snapshot is then compressed. The snapshot is specific to that single type of device, so a new shapshot will have to be generated for each platform separately. Each snapshot is generated for a specific type of device. This allows the server to determine which devices are compatible with the update. [45]

After the snapshot has been created, it's uploaded to the remote server. The user can then trigger the update on a specific device or a set of devices via a web interface.[45] After that the relevant client devices will download the update and place it on the inactive OS partition. After the update is finished, the bootloader is switched to read that inactive partition during the next boot. If the system fails to boot to the newly activated partition, the bootloader reverts back to using the old version and a fail safe redundant system is achieved. [25] The partition layout is visualized in figure 3.3.

The Mender client software can also run shell scripts in various stages of the update process. These stages include before and after downloading or installing of the update and in case of update failure. These scripts are delivered as a part of the update. The scripts can have multiple use cases. For example they can attempt to verify in more detail that the update was successful or they can request an end user for approval for the update to be installed. [46]

Mender also allows storage of configuration files and persistent data on a separate partition. This is not touched by the system update mechanism. It's purpose is similar to that of BalenaOS. However it is supposed to be accessed directly in the filesystem instead of using it via a docker volume. [45]

3.4.2 Authentication and security

When a new device is intialized, the Mender client will attempt to connect to a predetermined server. Authentication is based on public/private key pairs. Additionally a device is identified by it's MAC address and optionally other identifying keys. The authentication request is signed by the private key stored on the device during device initialization, and the server can then verify the request with the corresponding public key. [19]

After the device has authenticated with the server, the user must approve it in the remote servers web dashboard. An approved device receives an authorization token, which can be used for further communication. Each token has an expiration date, after which the device has to re-authenticate with the server. Alternative method of authentication is placing signed certificates on the device beforehand and using them to identify and approve the device automatically. [19]

Communications between the server and client are always initialized by the client software. As such using Mender doesn't require opening ports on the device for incoming traffic. Additionally all communications are using TSL encryption. [47]

3.5 Alternative update frameworks

Besides the three tools that were described, others potential options exist for Linux based OTA updates. However many of them have some shortcomings, which make them less suitable for the use in this comparison.

An example of such tool is AWS Greengrass. It is a set of tools provided by Amazon, that allow a Windows or Linux based IoT device to receive and install OTA updates from AWS cloud services. The updates consist of components which can be program binaries, docker images or AWS Lambda functions. If the components are individual programs, they can be updated independently from each other. [48]

AWS labels individual devices as things. It supports combining these things into groups which can be updated together or on device basis. By default Greengrass automatically rolls back the version of a software in case of a failed update. [49] Additionally Greengrass has various methods of monitoring individual devices. These include logging and health monitoring. Logs are stored on the device by default, but can be requested and read from the cloud interface. [50] Device health monitoring is achieved by each device sending update messages to the cloud whenever the device's status changes. These changes can include updates being installed or components going from functional to an erroneous state. This status can then be checked by the user from the cloud interface. [51]

The reason why AWS Greengrass isn't included in the compared tools is that it's meant to only update the software components and core client software instead of the entire oper-

ating system [48] However this approach doesn't have to deal with issues that introducing redundant software causes.

4. TEST ENVIRONMENT REQUIREMENTS

4.1 Environment and use case description description

The focus for this work is from a customer project, where there was a requirement for remotely updating an IoT system. The projects goal was to produce a new iteration of an existing IoT device on an embedded system. To prototype the system Google's Coral dev board was chosen as the initial platform..

The previous version of the device was a x86 based system. The devices are setup on remote premises without much interaction by the end user. They process data collected via sensors and a camera. The results are then sent to a cloud service, so the devices require a cellular internet connection. This iteration of the device requires remote access to adjust configurations and for installing updates or delivering new versions of the software.

The goal of the project was to produce a more scalable and secure solution for updating the devices. With the new iteration, each device still needs some initial configuration after it has been setup in the target environment. After that interventions should only happen when there is for example a hardware issue.

4.2 Hardware

The Coral dev board is a development board that is meant to be used for prototyping systems for on device machine learning purposes. The device itself is not meant for production use, but a System on a Module (SoM) is available. The goal of the customer project was to prototype the system on the dev board. If successful, a scalable product could have been created with the SoM by designing a custom board around the module with the required IO capabilities.

The main reason that the Coral dev board was chosen for prototyping was the tensor processing unit on the device. Offloading the classification of results onto the TPU and parts of image processing to the GPU allowed for much lower power device to be used. Another advantage of the Coral dev board was ready availability of hardware in comparison to other vendors, which had much longer order lead times.

The main features of the dev board include[52]:

- NXP i.MX 8M SOC with a 1.5GHz quad core ARM CPU and a Vivante GC7000Lite GPU
- 1GB of RAM
- 8GB of internal eMMC storage
- An edge TPU tensor processing unit

4.3 Requirements and challenges

Requirements for the system cover multiple areas. Those can be roughly split into functional, security and scaling.

Functional requirements include:

- Remotely triggering device updates
- Ability to recover from failed updates or power loss during updates without external intervention
- Unique data persistency between updates
- Resource usage
- Hardware compatibility

The first requirement is the original driving factor behind this work. The goal is to demonstrate a system where updates can be triggered remotely. The second requirement expands on that with the requirement of some type of redundancy or fault tolerance. If failure tolerance wasn't required, even something like the inbuilt package manager, might be suitable for updating the device. Also during the deployment of an update it should not be possible for the device to end up in a state, where it could not recover to a working state without external intervention.

The third requirement is related to the need to be able to configure devices independently of each other. The goal is for the devices to be running the same software, but as each device needs some environment dependent tuning, the related configurations must be saved on the device and not be altered between updates.

The devices are run in environments where they depend on a cellular connection for sending data between the device and cloud. As a result, the ideal situation is one, where the amount of data downloaded per update is minimized. The resource limitations also exist in the amount of available storage space, as the internal eMMC is limited to 8GB. As such the size required by the update framework is hopefully minimal to free disk space for useful payloads. Using a micro-SD card is preferably avoided, as that would make it way easier for a malicious actor to extract data from the device.

The last requirement is derived from the fact that the software that is used runs several

times slower on the Coral dev board in comparision to the earlier x86 based machine. To avoid resource starvation several optimizations have been done for the Coral version of the software. First, the program uses the GPU via OpenCL to do some image processing, which would take up too much CPU resources. Additionally some classification is done on the TPU. The required drivers and libraries would have to be present in an operating system, which integrates with the selected update framework. For example using OpenCL requires installing GPU drivers.

Security Requirements include:

- Authentication with both the device and remote server
- Secure transfer of updates

The first one requires that the server is able to identify different devices from each other. Also for example if the disk of one physical device is cloned to another, it shouldn't be able to identify with the remote server. Additionally the device should have some method of validating that the server is a trusted source for receiving updates from.

The latter requirement guarantees that the integrity of the updates is kept free from tampering. An example solution might include the usage of an encrypted transfer medium or signing of the update packages.

For physical security, encryption of the device is something that is desired by the customer. However such features go much beyond a prototype and from preliminary searches it seems that almost no update methods are designed to work with disk encryption without major modifications.

Scalability requirements include:

- Fleet division and updates

The goal of this requirement is to guarantee the possibility of scaling beyond a few dozen devices. With a large amount of devices updating each device one by one would not be desirable. Instead it should be possible to deploy updates to entire fleets. Division is also beneficial by allowing the use of multiple target architectures within a single update ecosystem. This is not relevant at the time of writing this work, but a need to deploy devices with differing architecture might arise at a later date. Additionally creating fleets of devices would allow testing of new versions of the software in specific pilot locations before updates are deployed more widely.

4.4 Beyond a proof of concept

The requirements that were described above are suitable for a proof of concept demonstration, but additional changes would have to be made for a production ready implemen-

tation. For example considerations would have to be made in regards to ensuring the security of the end device. For example hardware root of trust would have to be established to prevent the execution of unauthorized software and disk encryption could be used to prevent physical tampering of the device.

Such security considerations were mostly left out of the requirements that were introduced earlier. That also means the comparison of the frameworks will only take into account the needs of a proof of concept and not a production environment. However the iMX SoC of the Coral dev board does provide several features that could be used to implement security features.

For example Arm TrustZone and High Assurance Boot are mentioned on the dev board datasheet.[52] However the documentation doesn't state if the kernel and bootloader already support these features or if modifications would have to be made.

Arm TrustZone would allow for two separated software environments to be run on a single processor. One of the environments would be deemed as unsecure, and would have no direct access to the secure environment. Safety critical operations such as handling cryptographic keys could be offloaded to the secure environment to keep them out of reach from the non trusted software. Communication between the secure and unsecure environments happens via specific software to limit the exposure of the secure area. [53]

High Assurance Boot, or HAB, could be used to initialize the hardware root of trust on the device by verifying a bootloader before it is allowed to execute. For it to work the bootloader image is signed with a private key and the corresponding public key would be stored securely on the device by burning one time use fuses. During a boot the device can verify the bootloader signature and only allow the execution of trusted bootloaders. [54] In the customer project this feature was implemented as a standalone feature to demonstrate that its use is possible on the Coral dev board. Even though support for the feature was listed in the device datasheet, the bootloader had to be recompiled to support some required functionality. To be usable with an update framework additional work would most likely be needed. This is because some update frameworks make their own changes to the bootloader. Changes made by both would have to be integrated together. [25]

After the bootloader has been verified the chain of trust would have to extend to the operating system. This could potentially be achieved with a U-boot feature called Verified boot. It gives the bootloader the ability to check if a kernel image has been signed similarly to how HAB works with the bootloader.[55]

To account for the physical security of the device disk encryption could be implemented. However some of the frameworks won't support encryption out of the box and implementing support might require lots of work. Additionally there would have to be a secure

place to store the encryption keys on the device to make boot without requiring user input possible.

5. COMPARISON AND ANALYSIS OF THE FRAMEWORKS

This chapter compares the three frameworks according to the requirements described in chapter 4. The goal of this chapter is to find out which of the three best meets the requirements and why. Each section is focused on one of the requirements.

5.1 Functional

5.1.1 Remotely triggering updates

Triggering the update process remotely is the most important requirement for the use case and all of the three make it possible. [36] [25] [41] BalenaOS and Mender provide a browser based dashboard, from which the user can pick which devices are to be updated and select which update is to be delivered.[36] [25]. SWUpdate doesn't provide the backend for the system, but Eclipse hawkBit is responsible for providing the interface to upload the update to the server and push it out to devices.[41]

One advantage of BalenaOS is that it uses a supervisor program to start and stop containers when needed during the update process. This makes it possible to automate the startup of programs after an update has finished and when the device is booting up.[34] SWUpdate and Mender documentation do not mention similar functionality, so similar control of the applications would have to be implemented separately. However Mender allows for an update to include shell scripts that can be run before or after various stages of the update process. These could be used to achieve similar functionality, but those scripts have to be embedded into each update, whereas with BalenaOS it should be possible to completely automate the process. [46]

5.1.2 Recovery from failed updates

Balena divides updating the system into two separate issues. One is updating the operating system and the other the docker container images. Operating system updates use the symmetric partitioning as a recovery method from failed updates. Always keeping a functional system on at least one partition should make it possible to recover from an update

failure in a timely manner. However rebooting the system will introduce some downtime [24]

Updating the docker images can be done with different policies. If minimal recovery time is desired, the hand-over mode will guarantee that the new container will already be running before the old one is stopped and removed. However this will require the end user to define a policy in the new and old containers to determine when handover is acceptable. If no instant handover is required, other approaches can stop and remove the old container for example either before or after the replacement has been downloaded. [34] This gives some flexibility for balancing storage use and uptime.

The balena cloud service provides remote access to each device from the web interface. [37] This might be useful in debugging issues where the device can boot and connect to the internet, but there is some other problem with the updated software. Additionally this could be used for configuring new devices. The use case requires setting some parameters that change based on the device's environment. However the feature requires access to the dashboard, and as such is not usable on openBalena version.[37]

With Mender the system updates also require rebooting the device after each update. This introduces downtime both in case of a successful and failed update. Longer in the latter, as the system will attempt to load the updated version before reverting back to the old one. [25] Rebooting cannot be avoided unlike with BalenaOS, where the application containers can be updated on the fly. Additionally file updates cannot substitute the container updates, as the changes would be replaced with the next system update.[25]

Mender can also provide a remote terminal to connected devices via the web interface. This requires a mender-connect addon to be installed on the target device. The addon communicates with the remote server via a websocket connection forwarding user inputs to a shell and passes the outputs of the sell to the remote server. [56] As with the remote access feature of BalenaOS, the remote terminal with Mender might be useful when configuring new devices or debugging issues after updates. The feature also doesn't seem to be restricted to any commercial version of Mender, as it's available from the regular web UI. [56]

With SWUpdate the introduced downtime depends on the partitioning layout. With symmetric partitioning the functionality is very similar to Mender, where after an update has been written to the inactive partition, the system reboots and attempts to boot from that. In case of failure, the system will revert back to the old partition. If the recovery partition is used, the system will have to wait until a working operating system has been downloaded and installed, before the system will again be functional.[26]

5.1.3 Unique data persistency between updates

All three of the frameworks use a separate partition for storing persistent data. Such partitions are not modified by the update process. [26] [35] [45] With SWUpdate the persistent partition is only mentioned when referencing the symmetric partitioning. [26] The remote terminal features of BalenaOS and Mender could be used to modify the persistent data on the device for example when configuring devices.

5.1.4 Resource usage

Bandwidth usage was defined as an aspect of resource usage. One method of minimizing it is only downloading parts of an update that are different from what is already on the device. The frameworks have somewhat different approaches to this.

Mender supports delta updates, which attempt to only deliver the necessary data for updating the system from the previous version to the next one. To create a delta update, the user must have an existing update artifact for the older version and use a newly generated artifact to parse which parts of them are different from each other. As an update is being deployed, the remote server will check if the target device's inactive partition matches any of the delta updates and deploys one if possible. Multiple delta updates can be created for updating from various older versions and a full image can be used as a fallback. However this feature is restricted to the commercial version of Mender. [57]

Updating software on BalenaOS takes advantage of the layer based architecture of a Docker image. During an update only modified layers are downloaded. Additionally recent versions of BalenaOS add support for their own version of delta updates, where different versions of the Docker images are compared on the content level. These update deltas can be created on demand by the remote server based on what software a target device is running. [58] The delta update feature is only available with the cloud server. [38]

SWUpdate documentation mentions a few methods of delivering partial updates. The client software can use various handlers to support these features. For example an rdiff handler can fetch binary delta patches generated with librsync.[59] Alternatively a Delta update handler can fetch updates using a zchunk format. With the latter an update is split into chunks, and only those not already present on the device have to be downloaded. [60] The SWUpdate documentation doesn't seem to focus on how these update deltas are created. Instead the handlers focus on fetching already generated deltas.

Another aspect of resource usage is the disk space that is required for the usage of the framework. Mender requires two root partitions to be present. [44] As the entire software stack has to be present on both partitions, the total required disk space is almost doubled in comparison to not using any update tool. To save space one could move some parts of

the software stack to the persistent partition. However that would require managing them more difficult as they would have to be updated via the file update strategy, which lacks any type of versioning system.

SWUpdate with the symmetric partitioning suffers from the same drawbacks with Mender. Avoiding two separate filesystems can compress the update and recovery partition down to under 8MB. [26] However this has it's drawbacks that were described previously in regards to the recovery time from failed updates.

BalenaOS also uses similar partitioning layout for the main operating system. However the root partitions are mounted as read only and they only store the core operating system itself. That allows the duplicated partitions to be small in comparison to the rest of the system. Applications are stored as docker images in a separate partition that can automatically expand to fill up rest of the available disk space. [24] The savings in storage space can only really be achieved, if an old version of a docker image is removed from the device before a new one is downloaded. In such case a failed update would require the system to refetch the older version to revert back to it. Another noteworthy finding is that according to a BalenaOS developer it should be possible to flash BalenaOS the internal eMMC of the Coral dev board as an alternative to using a microSD card. [61]

As the use case is heavily resource dependant, one possible shortcoming of BalenaOS is the containerization of software running on the device introducing some overhead. Comparisons between native and containerized applications have previously been done in research papers. For example an IBM research paper from 2015 which compared database performance when docker versus a native environment. It presented results where I/O performance was around 2% worse with Docker in comparison to a native Linux. It's mentioned in the paper that I/O and OS level actions are affected by containerization and CPU performance and memory usage should be minimally affected. [62, pp. 171-172]

5.1.5 Hardware compatibility

This requirement concerns the Coral dev board support of each framework. Additionally it covers the support of board specific features that are required by the use case.

It should be possible to integrate each of the frameworks to the device. Mender uses Yocto to introduce the necessary modifications for Coral dev board compatibility. [63] In the previous reference, support for the internal eMMC memory is mentioned to be untested, but it has been demonstrated separately. [64]

The Mender build uses a meta-coral Yocto layer to provide support packages for the board. [63] This layer includes various libraries for the TPU support. [65] Additionally the meta-coral layer depends on a meta-freescale layer which provides gpu drivers with the imx-gpu-viv -package. [66]

BalenaOS provides a prebuilt operating system image and a guide for initializing the dev board with such image. The guide also describes usage of the TPU with an example project and as such the necessary libraries can be presumed to have been included in the base image. [67] GPU drivers might not be included in the prebuilt image, but building a custom image is possible by using board specific repositories. [32] The Coral specific repository includes the same meta-coral and meta-freescale layers as with Mender. As such GPU drivers should be available to a custom built operating system image.[68]

SWUpdate documentation doesn't specify Coral support explicitly. However it's supposed to be integrated with Yocto via a layer. [28] This could make it possible to integrate SWUpdate with a Yocto build, that supports Coral, via the meta-coral layer or otherwise. However this approach would presumably take much more work than the other two, as hardware specific aspects of the system updates haven't previously been demonstrated.

5.2 Security

5.2.1 Authentication with both the device and remote server

The authentication of a Mender device is done based on a key pair that is stored on the device. Additionally other identifying parameters are used by the server to distinguish between devices. These can include eg. the mac address of a device. The authentication requests are singed with a private key and the corresponding public key is included in the request. A server keeps track of these requests until an user approves the device. An approved device will receive an authentication token as a response for the next authentication request it makes. The client includes the received token in future communications with the server. After the token expires the device will request a new one from the remote. [19]

The registration can also be preauthorized by generating the key pair beforehand and submitting it to the remote server. When a new device is initialized, the same key pair is stored on the device. When the device then makes an authentication request, the remote server can automatically approve it and return a authentication token. [19]

The device determines the trustworthiness of the server based on the root certificate authorities that are available on the operating system. Alternatively the user can use for example self signed certificates. [69]

BalenaOS uses a provisioning key for initial authentication with the remote server. The key is preloaded on the device and it allows the device to join a predetermined fleet. After joining each device receives an API key, which is included in future requests. The API key can be revoked remotely and as such a compromised device can be removed from the fleet. [23]

The update images that are deployed, can be built locally before they are published to the cloud server or alternatively they can be built by a separate build server. When building an image, the build server can fetch the required base image either from Docker Hub or a private docker registry. In such instances the server might need to authenticate with the registry. The required credentials have to be set on the build server, before it can reach those registries. [36]

SWUpdate based devices can interact with the hawkBit remote server in multiple authentication modes. The most open policy is one, where any device is allowed to query and download updates. Such approach might not be feasible for the use case of this work, but hawkBit documentation states it as a possible option for situations, where the updates themselves are encrypted to protect the contents and anonymous downloading is not an issue. Other possible authentication policies use tokens. The token can either be provided by the device within a request or it can be added by a gateway that sits between the device and server. [70]

5.2.2 Secure transfer of updates

Mender and BalenaOS use TLS for secure communication between the remote server and the device.[23] [47] SWUpdate supports TLS connections with some additional configuration. [71] Additionally Both Mender and BalenaOS always initiate the connection from the device and as such no open ports are required for incoming traffic on the device itself. [23]

Some of the frameworks support signing the updates to make it possible for the target devices to verify their authenticity. For example Mender supports using key pairs to sign update artifacts. A new artifact is signed using a private key and the client software can use a corresponding public key to verify it. [72] Additionally each update artifact includes metadata with a checksum of the update contents. This allows the client to check if the delivered artifact has been corrupted during transit. [73]

SWUpdate also supports signing the update images with a private key. The feature is similar to Mender in functionality. [42] Additionally SWUpdate supports symmetric encryption for update images. [74]

For BalenaOS there wasn't any documentation found on the subject of signing updates. Docker itself provides a feature called content trust, which allows the creator of the image to sign it with a private key. When the image is downloaded from the container registry, the recipient can verify the integrity and publisher of the image with an corresponding public key. [75] However it seems that currently BalenaOS is not capable of checking whether the images match the public key. [76] Docker has a builtin feature where it verifies fetched layers using checksums. [77] When not using delta updates, BalenaOS downloads new

container images this way, so they can be protected from being corrupted in transfer. [34]

5.3 Scalability

5.3.1 Fleet division and updates

Separation of devices into groups is important for example in situations where parts of the fleet are used for piloting new versions of software. All of the frameworks provide some type of support for separating devices into groups.

Mender supports dividing devices into groups using the management UI or via a separate API. The simpler division into groups can be done by manually assigning each device into a static group. A device can only belong to a single static group at once. Alternatively devices can be added automatically into dynamic groups based on some attributes. [78] The groups can be leveraged when creating new deployments of updates. When a deployment is made to a static group, it is deemed to have finished after each device in the group has been updated. With dynamic groups the deployment will remain in progress even after each device has been updated. The deployment will only finish after being stopped by the user, or after they meet a specified criteria of number of updated devices. [79]

The balenaCloud management view also supports creation of new fleets of devices. [80] The fleets can be updated and managed independently of each other. [21]

hawkBit is responsible for the remote management of devices using SWUpdate. It should also support splitting devices into groups. While the workflow for doing that is not explicitly defined in the documentation, the rollout management still describes the ability to install updates in cascading stages of groups. Additionally these groups can be chained together for a cascading roll out. In such configuration an update is first delivered to a group of devices. If enough devices update successfully, the update is then rolled out to a growing number of devices. The feature supports configurable limits for how many devices have to succeed in either percentages or number of devices. In case the limit is not met with a certain group, rest of the roll out is stopped. [20]

The commercial versions of Mender support a similar feature where the user can use the management view to split the update process into phases. The phases include attributes for which percentage of target devices should be updated by each phase, and how long the wait should be between phases. The user can monitor the progression and halt it if problems arise. [81] Additionally the regular Mender deployment allow the delivery of updates to both static and dynamic groups of devices. [79]

By default BalenaOS rolls out updates automatically to all devices of the fleet whenever the update is available. To prevent updates from being installed on a fleet, it can be pinned

to stay behind on the current release version. A pinned fleet can later be set to update to a specific newer release version or to revert back to tracking the latest available version. Pinning devices can also be done on per device basis or in groups based on arbitrary device tags. [21] These features could be used to achieve a similar cascading release as with the other tools, but it requires manual work during each update to set a specific version for each device group.

5.4 Results

These results only evaluate the frameworks based on the requirements set for the proof of concept. A production ready device would have a more expansive list of requirements especially concerning the security of the end device. Some methods for securing the device were introduced in section 4.4. Such features would most likely have to be implemented by the developer of the IoT system. For example no documentation on any of the compared frameworks seem to mention any type of possibility for the use of disk encryption in an updateable system. Other features such as achieving hardware root of trust might be possible by modifying the linux distribution that is integrated with each framework.

The amount of required configuration for a functional installation in addition to the uncertainties with hardware compatibility make SWUpdate unsuitable for the use case. Mender and BalenaOS provide a clearer path for implementing a working system on the Coral platform.

The documentation of both Mender and BalenaOS describe a feasible method of delivering updates to the Coral platform. Both use a similar approach for ensuring recovery from failed operating system updates. The main drawback of Mender is that the entire system besides bootloader must exist twice on the disk. This restricts the available storage space heavily if using the internal 8GB eMMC. BalenaOS somewhat avoids this issue while maintaining a functional versioning system for application updates. Additionally the performance impact of using containers should be minimal.

Hardware compatability of both Mender and BalenaOS should be nearly to identical. This is due to the compatibility being provided by the very same Yocto layer. Both will likely require customization of the default configuration to add GPU drivers and other dependencies.

Ease of scaling the number of devices is should be similar with both Mender and BalenaOS. Both support division of devices into groups and groups can be updated independently from each other. Devices can still be managed one by one. Both also include monitoring tools for keeping track of state of devices.

The one major advantage Mender has is the ability to host the update server on premises instead of the cloud interface. The openBalena server is much more limited than the cloud

Table 5.1. Results for comparison of update frameworks

Requirement	SWUpdate	Mender	BalenaOS
Functional			
Remotely triggering updates	+	+	++
Recovery from failed updates	+	+	+
Data persistency	+	+	+
Resource usage	+	-	++
Hardware compatibility	-	+	+
Security			
Authentication	+	+	+
Transfer	+	+	+
Scalability			
Fleet division and updates	++	++	+

version.

The benefits and shortcomings of each framework in regards to the use case requirements are presented in table 5.1.

6. PROOF OF CONCEPT

Of the three compared frameworks, BalenaOS was selected to be studied further with a proof of concept. The reason that BalenaOS was selected is that generating updates as container images can be much more straight forward than having to create a snapshot of a running system. Additionally compared to the other frameworks, the additional storage requirements presented by BalenaOS are smaller due to the OS installations being minimal and the applications being stored in a single partition.

6.1 Previous work and goals for the proof of concept

Before much research was done into alternative options, some testing had already been done in the customer project. During that work it was demonstrated that the required OpenCL platform could be configured on a Mender compatible Yocto image. However as mentioned in the previous chapter, Mender mirrors the entire operating system installation, and as such the usable space available on the eMMC was heavily constrained. Additionally deploying updates was quite cumbersome. Creating an update artifact required setting up a complete device from which the artifact was generated. Every update would have been a full system update. Remote file updates could be done, but those were overwritten by the next full update. Finally there were some issues with the built in update handler erasing the relevant bootloader environment variables. This would make the bootloader lose track of active and inactive partitions and the state of the update process. This made it possible for a device to be rendered unusable after an update unless the variables were set again manually.

The goal of this proof of concept is to try to replicate what was previously attempted with Mender. The goals include demonstrating that the OS can be installed on the Coral dev board and that both OS and container updates can be deployed to the device. Additionally it is necessary to demonstrate that GPU drivers, a OpenCL platform and the libraries related to the TPU can be setup correctly. Finally the prototyping should demonstrate that deploying updates to BalenaOS requires less work than when using Mender, as this was one of the main reasons BalenaOS was deemed better for the use case.

Figure 6.1. Creating a new fleet in balena cloud.

6.2 Proof of concept setup and usage

6.2.1 Initializing a fleet in balenaCloud

To start prototyping an account was created to balenaCloud. After that a new fleet was added as shown in figure 6.1. Next we could move forward to creating an operating system image that will connect to the fleet.

6.2.2 Generating a custom image for adding extra features

The prebuilt operating system images, provided by Balena, didn't include the required GPU drivers or OpenCL related utilities, so an attempt was made to build a custom version with the additional features present. The following steps were taken to generate an flasher image, which could be used to install the system on the device.

Yocto documentation lists compatible versions of host operating systems, which can be used for building the OS image. The compatible operating systems are dependent on the version of Yocto that is used for building the image. For use for Coral dev board Yocto 3.0, or zeus, was required. This is because it was the latest version that is supported by the meta-coral layer, which provides board specific support. The latest host OS version compatible with Yocto 3.0 was Ubuntu 18.04, so that was used. [82] An attempt was made to use a more recent 20.04 version, but newer versions of Ubuntu have for example an updated version of the GNU C library, which has deprecated some features still required by Yocto 3.0.

The build process mostly followed the README that could be found in the Coral board specific BalenaOS repository but some changes were required to add the necessary dependencies and libraries. The process for building a flasher image was as follows.

Cloning the source repository:

```
git clone --recursive https://github.com/balena-os/balena-coral
```

Initializing the build system:

```
./balena-yocto-scripts/build/barys --remove-build --dry-run
```

`build/conf/local.conf` was modified to add the necessary additional dependencies. The edited file is included in appendix A.

Recipes related to OpenCL had to be manually copied into the Yocto directory structure, as they were not available by default.

Repository: `https://github.com/openembedded/meta-openembedded`

Recipes from folder: `meta-oe/recipes-support/opencl/`

Into target folder within the Yocto project directory tree:
`meta-oe/recipes-support/opencl/`

To include the Vivante GPU driver in the kernel changes were made to the kernel defconfig at:

```
meta/coral/recipes-kernel/linux/linux-coral/defconfig
```

By including: `CONFIG_MXC_GPU_VIV=y` at the end of the file.

The root file system size limit had to be increased to make room for the additions. The following changes were added:

```
IMAGE_ROOTFS_EXTRA_SPACE = "100000"
```

```
IMAGE_ROOTFS_MAXSIZE = "530000"
```

At the end of the file in:

```
layers/meta-balena/meta-balena-common/recipes-core/images/balena-image.bb
```

After that the flasher image could be built with the following commands:

```
source layers/poky/oe-init-build-env
```

```
MACHINE=coral-dev bitbake balena-image-flasher
```

After the build was finished, the flasher image could be found from the following path:

```
build/tmp/deploy/images/coral-dev/<imagename>.balenaos-img
```

6.2.3 Configuring custom operating system image to connect with fleet

After the BalenaOS image had been created, it had to be configured to authorize it for connecting to the correct fleet. The same balena-cli tool, which is used to deploy updates, was used to configure the image. To configure the image a balena-cli tool was installed to the Ubuntu development system. The version was 13.6.0. The following command was used to authenticate with balenaCloud via a browser prompt.

```
balena login
```

After that the flasher image was renamed to `balena.img` and placed into the root directory of the cli-tool. The following command was run to configure the image. The fleet parameter tells the system to connect to the fleet that was configured in the cloud dashboard. The version parameter was necessary for the configuration to pass. It was set to match the latest version available in the balena-coral repository.

```
balena os configure balena.img -fleet coralfleet -version 2.98.33
```

After these steps the image could be flashed to a microsd card and the system was installed on the device by booting it with the card. After the initialized device had booted up, it was visible in the cloud dashboard.

After the custom built operating system image had been installed, the availability of the OpenCL platform could be verified by running `clinfo` on the device console. The tool reported the available GPU and OpenCL platforms correctly.

6.2.4 Deploying application updates

Testing the deployment of software was done by using a project, which runs a sample image classifier on the device's onboard TPU.

The relevant project files can be found from:

`https://github.com/milanfin/balena-coral-tests`

The test application was packaged into a docker image according to a `Dockerfile.coral_env` file. The file ending describes the target platform for the Balena build server. Other file endings could be used for other platforms to allow a single project to be deployed to several platforms at once.

Docker compose is used to make it possible to package multiple docker images to a single deployment. Each image is introduced as a service in a `docker-compose.yml` file. Finally a `balena.yml` file is used to specify additional metadata such as a list of supported devices.

The balena-cli tool was again used to authenticate with the cloud service.

```
balena login
```

The tool was then used to deploy the test project into the fleet, by running the following command in the same directory as the `balena.yml` file.

```
balena push coralfleet
```

This pushed the project contents to a build server and the update was deployed automatically after the build was finished. The state of the update process could be followed in the cloud dashboard, which displayed device states and devices specific logs.

From the logs it was possible to verify that the sample image classifier had been run successfully using the onboard TPU. Updated versions of this software could be pushed by modifying the project and redoing the deployment. The device could also be held back in a specific version of the software by pinning it to a release from the cloud dashboard.

6.2.5 Shortcomings of the custom operating system image

It was discovered that currently it's not possible to push custom operating system updates directly to the cloud server. Instead only prebuilt operating system images can be selected from the dashboard. There is documentation that states that third party image builders should contact the provider for their custom image to be included in the build system. This would presumably work by creating a repository of the project files and having a separate build server build the OS image based on the contents of the repository. This image would then be available for deployment from the cloud dashboard. [32] However usage of a possibly commercial service to include a custom operating system build onto the dashboard goes beyond the scope of this prototype.

Prototyping things with the custom image was somewhat difficult, as all modifications have to be included when the image itself is built. When using a regular Debian based operating system on the Coral dev board, it was possible to easily test install and remove libraries and drivers after the system had been initialized. To emulate this a package manager could be added onto BalenaOS, but installing packages and doing modifications on a live device is difficult due to the OS using a read only root filesystem. However this isn't an issue in production environment, where generating updates should require minimal manual work.

6.2.6 Operating system updates with a prebuilt OS image

To be able to test the operating system updates in any capacity, the prebuilt OS images were used. A prebuilt image could be configured in balenaCloud as shown in figure 6.2.

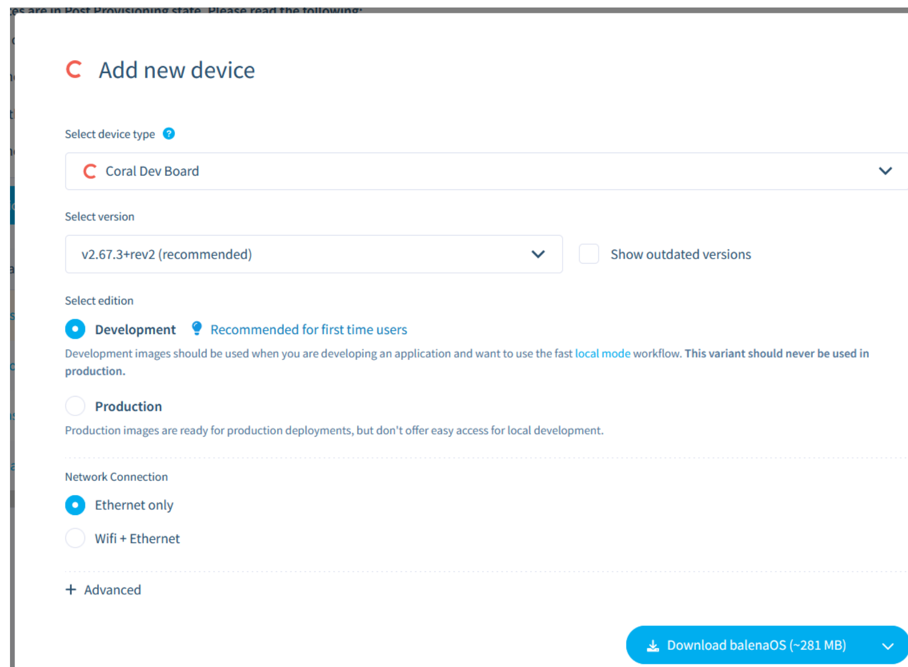

Figure 6.2. *Configuring a new device.*

The result was an image that could be flashed to a microsd card. Installation of the OS was done as with the custom built image. After a new device had been flashed, it was visible in the corresponding fleet in the cloud dashboard.

An older version, in this case 2.48.0+rev5 was purposefully installed to make it possible to trigger an update to a newer OS version. Triggering an update to a newer version was done from the cloud dashboard. Only a single device was updated, there was an option to update the entire fleet, if more devices were available. The target update version was 2.56.0+rev1, which was the next available version. The update process first updated the base operating system and after rebooting the supervisor. At the same time that this test was run it was also verified that files that were manually placed on the persistent data partition were not touched by the update process.

Testing recovery from failed updates was done rudimentally by removing power from the device during the update process. The status of the update was followed from the dashboard and on the first time power was cut was during the operating system update phase. In this scenario the device recovered to the previous version of the operating system correctly. Restarting the update didn't happen automatically but instead it had to be triggered again from the dashboard.

The second scenario was removing power after the operating system update had finished, but during the supervisor update. In this instance the device booted up to the new version of the operating system successfully. After it had reconnected with the cloud service, the supervisor update was retried automatically.

Currently the latest available version is 2.67.3+rev2 from January 2021, so first party updates don't seem frequent unless more recent versions are available under commercial licenses.

6.2.7 Logging, remote access and management tools

As determined in the comparison of the frameworks, the cloud dashboard provides a method of remotely accessing the device terminal. The terminal was available when an individual device was being viewed. It allowed for direct access to a root shell on the device. Additionally the remote terminal could attach to docker containers that were running on the device.

From the same device specific view an user can also view the state of each docker container and the logs entries that they had generated. The logs could indicate for example the process of individual containers being updated.

The cloud interface allowed for revoking the access of devices that have previously been added to a fleet. Also location data was available for each device. Apparently the location was determined from the public IP address, so could be somewhat inaccurate. [83]

7. RESULTS FROM PROOF OF CONCEPT

7.1 Results compared to original requirements

When the proof of concept is assessed in regards to the original use case requirements the results mostly correspond to those that were found in chapter 5. Almost all of the functional requirements were met with the current prototype. Creation and delivery of application updates was straight forward and the updates could be deployed from any device that is capable of using the balena-cli tool. Also recovery from a failed installation of updates was demonstrated. The only issue was that delivering OS updates to a customized OS image would require collaboration with the vendor of the framework.

Security aspects of the requirements are also met. For example the remote server can distinguish individual devices from each other. Additionally the devices could be rebooted, shut down or completely removed from the fleet via the remote server. The transfer of updates happened over a TLS encrypted connection.

The scalability related features of the cloud service are robust. Devices could be managed individually or a fleet could be created. A shortcoming in regards to scalability is that most of the related features are dependent on using the cloud service. This introduces significant vendor lock-in when using BalenaOS.

7.2 Other notable findings

It was possible to install the required software dependencies on the custom OS image. However prototyping such system was cumbersome, as the OS image had to be regenerated any time a change had to be made to the libraries on the core OS. This also applies to any security related features that might be desirable in a production ready device. For example on Mender the OS could be modified after installation and changes could be delivered as an operating system update. On BalenaOS the changes would have to be introducing during the building of the OS image.

In regards to security the device is very vulnerable physically. The system was installed in the eMMC which is soldered onto the board, so that should make accessing the file contents a bit more difficult than if the system was installed on a microsd card. However

booting the device from a microsd card is still possible and in that instance the user can still access the internal memory. Some work was previously done in the customer project looking into integrating High Assurance Boot with Yocto on the Coral dev board, and it seemed to be possible to block the execution of untrusted bootloaders. However including other features such as full disk encryption is difficult as BalenaOS and neither of the other two frameworks are designed to work with an encrypted system. Even more considerations would have to be made for an unattended boot with disk encryption, as the devices would be in environments where there would not be a person to input a password every time the device had to reboot.

Much of the documentation that is needed to make some libraries or security features work with Yocto seem to be platform specific. For the Coral dev board or the Coral SoM much of that documentation doesn't seem to exist. As a result there is a lot off guesswork on what needs to be done to make various parts of the system work. This makes prototyping with Yocto slower than with for example the default operating system that is provided to the Coral dev board by Google.

8. CONCLUSION

Conventional IoT devices rely heavily on the cloud for processing data. Updating the cloud part of the system is relatively straight forward. However moving parts of that data processing onto the end device makes deploying updates to those same features more difficult. Updating these devices causes concerns with the reliability and security of the update process. Many reliability focused approaches store two versions of an operating system on a single disk, a primary and a backup. An alternative approach can rely on a minimal recovery operating system installation, that is used as a backup in case of failed updates. Making a device updateable also introduces challenges on how the chosen update approach functions with the desired security features. For example updating an entire operating system of a device that relies on disk encryption is difficult.

Three update frameworks were compared to determine the most suitable option for remotely updating the entire operating system of a Google Coral Dev board. Of those frameworks Mender and BalenaOS both had previous examples of their use with the target platform. Unlike Mender the approach used by BalenaOS didn't require mirroring the entire system installation on two partitions. Additionally BalenaOS uses containers, which make it possible to update only parts of the system at once. Mender requires the entire system image to be replaced each time a new release is deployed to the device. A table describing how each of the frameworks fulfills the requirements of the test scenario can be found at the end of chapter 5.

A proof of concept of an updateable system using BalenaOS was implemented on the Coral dev board. Container image deployment and recovery from failed updates could be demonstrated. The main issue with the proof of concept was that adding for example device drivers onto the system installation proved difficult. A custom Yocto based image could be generated, on which the required libraries and drivers could be installed. However prototyping was made difficult by a read only file system not allowing changes at runtime. Additionally a major shortcoming of BalenaOS turned out to be that operating system updates can only be pushed from the provider's own repositories. As a result publishing OS updates on the custom built version couldn't be tested at this time.

Requiring the entire operating system to be updatable made the configurement of new systems much more difficult than if a less comprehensive approach was used. Having to

integrate the entire filesystem with the update framework removes a lot of flexibility that is useful, when testing and developing new prototypes. Additionally using security features such as disk encryption and verifying the bootloader and kernel is much more difficult if those features have to be integrated onto a redundant system.

REFERENCES

[1] A. Qureshi, M. Marvi, J. A. Shamsi, and A. Aijaz, “Euf: A framework for detecting over-the-air malicious updates in autonomous vehicles”, eng, *Journal of King Saud University. Computer and information sciences*, 2021, ISSN: 1319-1578.

[2] S. El Jaouhari and E. Bouvet, “Secure firmware over-the-air updates for iot: Survey, challenges, and discussions”, eng, *INTERNET OF THINGS*, vol. 18, pp. 100 508–, 2022, ISSN: 2542-6605.

[3] *Fog and edge computing : principles and paradigms*, eng, 1st edition, ser. Wiley series on parallel and distributed computing. Hoboken, New Jersey: John Wiley Sons, Inc., 2019, ISBN: 1-119-52506-3.

[4] A. M. Alwakeel, “An overview of fog computing and edge computing security and privacy issues”, eng, *Sensors (Basel, Switzerland)*, vol. 21, no. 24, pp. 8226–, 2021, ISSN: 1424-8220.

[5] K. Kandasamy, S. Srinivas, K. Achuthan, and V. P. Rangan, “Iot cyber risk: A holistic analysis of cyber risk assessment frameworks, risk vectors, and risk ranking process”, eng, *EURASIP Journal on Information Security*, vol. 2020, no. 1, pp. 1–18, 2020, ISSN: 2510-523X.

[6] A.-R. Cazacu, “Iot security: Threats and possible solutions”, eng, *Informatica economica*, vol. 26, no. 2, pp. 57–64, 2022, ISSN: 1453-1305.

[7] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, “Ddos in the iot: Mirai and other botnets”, eng, *Computer (Long Beach, Calif.)*, vol. 50, no. 7, pp. 80–84, 2017, ISSN: 0018-9162.

[8] M. De Donno, N. Dragoni, A. Giaretta, and A. Spognardi, “Ddos-capable iot malwares: Comparative analysis and mirai investigation”, eng, *Security and communication networks*, vol. 2018, pp. 1–30, 2018, ISSN: 1939-0114.

[9] N. Dejon, D. Caputo, L. Verderame, A. Armando, and A. Merlo, “Automated security analysis of iot software updates”, eng, in *Information Security Theory and Practice*, ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 223–239, ISBN: 3030417018.

[10] H.-N. Nguyen, T.-T. Nguyen, T.-N. N. Thi, M.-D. Tran, and B.-H. Tran, “Proposed methods t o rollback a failed update of iot devices”, eng, *International journal of engineering and advanced technology*, vol. 11, no. 2, pp. 55–62, 2021, ISSN: 2249-8958.

[11] A. Gilchrist, *IoT security issues*, eng, First edition. Boston, [Massachusetts] ; De G Press, 2017 - 2017, ISBN: 1-5015-0562-9.

[12] G. Kim and I. Y. Jung, “Integrity assurance of ota software update in smart vehicles”, eng, *International journal on smart sensing and intelligent systems*, vol. 12, no. 1, pp. 1–8, 2019, ISSN: 1178-5608.

[13] Y. Ashibani and Q. H. Mahmoud, “Cyber physical systems security: Analysis, challenges and solutions”, eng, *Computers security*, vol. 68, pp. 81–97, 2017, ISSN: 0167-4048.

[14] R. Petri, M. Springer, D. Zelle, I. McDonald, A. Fuchs, and C. Krauß, *Evaluation of lightweight tpms for automotive software updates over the air: Paper presented at 4th escar usa, the world’s leading automotive cyber security conference, detroit, mi, june 1-2, 2016*, eng, 2016.

[15] J. L. Hernandez-Ramos, G. Baldini, S. N. Matheu, and A. Skarmeta, “Updating iot devices: Challenges and potential approaches”, eng, in *2020 Global Internet of Things Summit (GIoTS)*, IEEE, 2020, pp. 1–5, ISBN: 9781728121710.

[16] “Create a custom update module”. (), [Online]. Available: https://docs.mender.io/artifact-creation/create-a-custom-update-module (visited on 02/18/2022).

[17] L.-C. Duca, A. Duca, and C. Popescu, “Ota secure update system for iot fleets”, eng, *International journal of advanced networking and applications*, vol. 13, no. 3, pp. 4988–4992, 2021, ISSN: 0975-0290.

[18] “Why device management is the core compatibility of an iot platform”. (), [Online]. Available: https://medium.com/softweb-solutions-inc/why-device-management-is-the-core-compatibility-of-an-iot-platform-88e8f8c821de.

[19] “Device authentication”. (), [Online]. Available: https://docs.mender.io/overview/device-authentication (visited on 02/25/2022).

[20] “Rollout management”. (), [Online]. Available: https://www.eclipse.org/hawkbit/concepts/rollout-management/ (visited on 03/18/2022).

[21] “Release policy”. (), [Online]. Available: https://www.balena.io/docs/learn/deploy/release-strategy/release-policy/ (visited on 03/04/2022).

[22] S. Yoon and J. Kim, “Remote security management server for iot devices”, eng, in *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, vol. 2017-, IEEE, 2017, pp. 1162–1164, ISBN: 1509040315.

[23] “Security”. (), [Online]. Available: https://www.balena.io/docs/learn/welcome/security/ (visited on 02/25/2022).

[24] “Architecture overview”. (), [Online]. Available: https://www.balena.io/os/docs/architecture/ (visited on 02/18/2022).

[25] “Introduction”. (), [Online]. Available: https://docs.mender.io/overview/introduction (visited on 02/25/2022).

[26] “Software management on embedded systems”. (), [Online]. Available: https://sbabic.github.io/swupdate/overview.html#software-management-on-embedded-systems (visited on 03/04/2022).

[27] “Overview”. (), [Online]. Available: https://docs.mender.io/system-updates-yocto-project/overview (visited on 02/18/2022).

[28] “Meta-swupdate: Building with yocto, Overview”. (), [Online]. Available: https://sbabic.github.io/swupdate/building-with-yocto.html (visited on 02/18/2022).

[29] “2 introducing the yocto project, 2.2 the yocto project layer model”. (), [Online]. Available: https://docs.yoctoproject.org/overview-manual/yp-intro.html (visited on 02/18/2022).

[30] S. Angolini Daiane, *Embedded Linux Development Using Yocto Projects*. Packt Publishing, 2017.

[31] “4 yocto project concepts, 4.1 yocto project components”. (), [Online]. Available: https://docs.yoctoproject.org/overview-manual/concepts.html#yocto-project-components.

[32] “Building your own”. (), [Online]. Available: https://www.balena.io/os/docs/custom-build/ (visited on 04/08/2022).

[33] “Controlling the update strategy”. (), [Online]. Available: https://www.balena.io/open/docs/getting-started/ (visited on 02/18/2022).

[34] “Controlling the update strategy”. (), [Online]. Available: https://www.balena.io/docs/learn/deploy/release-strategy/update-strategies/ (visited on 02/18/2022).

[35] “Communicate outside the container, Storage”. (), [Online]. Available: https://www.balena.io/docs/learn/develop/runtime/#storage (visited on 02/18/2022).

[36] “Deploy to your fleet”. (), [Online]. Available: https://www.balena.io/docs/learn/deploy/deployment/ (visited on 03/04/2022).

[37] “Ssh access”. (), [Online]. Available: https://www.balena.io/docs/learn/manage/ssh-access/ (visited on 03/04/2022).

[38] “Balena-coral repository”. (), [Online]. Available: https://www.electronics-lab.com/openbalena-iot-management-platform-linux-devices/ (visited on 04/14/2022).

[39] “Swupdate: Software update for embedded system”. (), [Online]. Available: https://sbabic.github.io/swupdate/swupdate.html (visited on 03/04/2022).

[40] “Suricatta daemon mode”. (), [Online]. Available: https://sbabic.github.io/swupdate/suricatta.html (visited on 03/04/2022).

[41] “Eclipse hawkbit, Overview”. (), [Online]. Available: https://projects.eclipse.org/projects/iot.hawkbit (visited on 03/04/2022).

[42] “Update images from verified source”. (), [Online]. Available: https://sbabic.github.io/swupdate/signed_images.html (visited on 03/04/2022).

[43] “Device support”. (), [Online]. Available: https://docs.mender.io/overview/device-support (visited on 02/25/2022).

[44] “Overview”. (), [Online]. Available: https://docs.mender.io/system-updates-yocto-project/overview (visited on 02/25/2022).

[45] “Deploy a system update”. (), [Online]. Available: https://docs.mender.io/get-started/deploy-a-system-update (visited on 02/25/2022).

[46] “State scripts”. (), [Online]. Available: https://docs.mender.io/artifact-creation/state-scripts (visited on 02/25/2022).

[47] “Security”. (), [Online]. Available: https://docs.mender.io/overview/security (visited on 02/25/2022).

[48] “How aws iot greengrass works”. (), [Online]. Available: https://docs.aws.amazon.com/greengrass/v2/developerguide/how-it-works.html (visited on 04/29/2022).

[49] “Create deployments”. (), [Online]. Available: https://docs.aws.amazon.com/greengrass/v2/developerguide/create-deployments.html (visited on 04/29/2022).

[50] “Monitor aws iot greengrass logs”. (), [Online]. Available: https://docs.aws.amazon.com/greengrass/v2/developerguide/monitor-logs.html (visited on 04/29/2022).

[51] “Check greengrass core device status”. (), [Online]. Available: https://docs.aws.amazon.com/greengrass/v2/developerguide/device-status.html (visited on 04/29/2022).

[52] “Dev board datasheet”. (), [Online]. Available: https://coral.ai/docs/dev-board/datasheet/ (visited on 04/08/2022).

[53] “Arm trustzone explained”. (), [Online]. Available: https://www.microcontrollertips.com/embedded-security-brief-arm-trustzone-explained/.

[54] “I.mx high assurance boot (hab)”. (), [Online]. Available: https://www.variscite.com/blog/434-what-is-hab-and-why-do-we-need-it/ (visited on 05/06/2022).

[55] “Verified u-boot”. (), [Online]. Available: https://lwn.net/Articles/571031/ (visited on 05/06/2022).

[56] “Remote terminal”. (), [Online]. Available: https://docs.mender.io/add-ons/remote-terminal (visited on 04/22/2022).

[57] “Robust delta update rootfs”. (), [Online]. Available: https://hub.mender.io/t/robust-delta-update-rootfs/1144 (visited on 04/08/2022).

[58] “Delta updates”. (), [Online]. Available: https://www.balena.io/docs/learn/deploy/delta/ (visited on 04/08/2022).

[59] “Handlers”. (), [Online]. Available: https://sbabic.github.io/swupdate/handlers.html (visited on 04/08/2022).

[60] “Handlers”. (), [Online]. Available: https://sbabic.github.io/swupdate/delta-update.html (visited on 04/08/2022).

[61] “Balenaos on coral dev board’s emmc”. (), [Online]. Available: https://forums.balena.io/t/balenaos-on-coral-dev-boards-emmc/180029 (visited on 04/08/2022).

[62] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers”, eng, in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2015, pp. 171–172, ISBN: 9781479919567.

[63] “Google coral dev board”. (), [Online]. Available: https://hub.mender.io/t/google-coral-dev-board/1711 (visited on 02/25/2022).

[64] “Boot from emmc on coral dev board”. (), [Online]. Available: https://hub.mender.io/t/boot-from-emmc-on-coral-dev-board/2256/3 (visited on 04/08/2022).

[65] “Meta-coral”. (), [Online]. Available: https://layers.openembedded.org/layerindex/branch/master/layer/meta-coral/ (visited on 04/08/2022).

[66] “Meta-freescale”. (), [Online]. Available: https://layers.openembedded.org/layerindex/branch/master/layer/meta-freescale/ (visited on 04/08/2022).

[67] “Get started with coral dev board and python”. (), [Online]. Available: https://www.balena.io/docs/learn/getting-started/coral-dev/python/ (visited on 04/14/2022).

[68] “Balena-coral repository”. (), [Online]. Available: https://github.com/balena-os/balena-coral (visited on 04/14/2022).

[69] “Security considerations for remote management of software in iot devices”. (), [Online]. Available: https://mender.io/user/pages/05.resources/04.whitepapers/iot-device-security/iot-device-security.pdf (visited on 04/14/2022).

[70] “Authentication”. (), [Online]. Available: https://www.eclipse.org/hawkbit/concepts/authentication/ (visited on 04/14/2022).

[71] “Config for hawkbit under ssl/tls using private ca / sub ca”. (), [Online]. Available: https://sbabic.github.io/swupdate/hawkbit-setup.html (visited on 04/22/2022).

[72] “Sign and verify”. (), [Online]. Available: https://docs.mender.io/development/artifact-creation/sign-and-verify (visited on 04/14/2022).

[73] “Artifact”. (), [Online]. Available: https://docs.mender.io/overview/artifact (visited on 04/14/2022).

[74] “Symmetrically encrypted update images”. (), [Online]. Available: https://sbabic.github.io/swupdate/encrypted_images.html (visited on 04/14/2022).

[75] “Content trust in docker”. (), [Online]. Available: https://docs.docker.com/engine/security/trust/ (visited on 04/22/2022).

[76] “Docker content trust/container signing enforcement”. (), [Online]. Available: https://forums.balena.io/t/docker-content-trust-container-signing-enforcement/5860/8 (visited on 04/22/2022).

[77] K. Rajendran. “How to check the authenticity of container base images”. (), [Online]. Available: https://www.suse.com/c/how-to-check-the-authenticity-of-container-base-images/ (visited on 05/06/2022).

[78] “Device group”. (), [Online]. Available: https://docs.mender.io/overview/device-group (visited on 04/22/2022).

[79] “Deployment”. (), [Online]. Available: https://docs.mender.io/overview/deployment (visited on 04/22/2022).

[80] “Fleet types”. (), [Online]. Available: https://www.balena.io/docs/learn/manage/app-types/ (visited on 04/22/2022).

[81] “Managing fleets of connected devices with phased rollout”. (), [Online]. Available: https://mender.io/blog/managing-fleets-of-connected-devices-with-phased-rollout (visited on 04/22/2022).

[82] “Yocto project reference manual, 1.1. supported linux distributions”. (), [Online]. Available: https://docs.yoctoproject.org/3.0/ref-manual/ref-manual.html#detailed-supported-distros (visited on 05/06/2022).

[83] “Visualising the location of your devices”. (), [Online]. Available: https://www.balena.io/blog/visualising-the-location-of-your-devices/ (visited on 06/17/2022).

APPENDIX A: LOCAL.CONF

```
BALENA_STORAGE = "overlay2"
OS_DEVELOPMENT = "1"
BB_NUMBER_THREADS ?= "${@oe.utils.cpu_count()}"
PARALLEL_MAKE ?= "-j ${@oe.utils.cpu_count()}"
DISTRO ?= "resin-systemd"
RM_OLD_IMAGE = "1"
USER_CLASSES ?= "buildstats image-mklibs image-prelink"
PATCHRESOLVE = "noop"
BB_DISKMON_DIRS = "\
    STOPTASKS,${TMPDIR},1G,100K \
    STOPTASKS,${DL_DIR},1G,100K \
    STOPTASKS,${SSTATE_DIR},1G,100K \
    ABORT,${TMPDIR},100M,1K \
    ABORT,${DL_DIR},100M,1K \
    ABORT,${SSTATE_DIR},100M,1K"
CONF_VERSION = "1"
HOSTTOOLS += "docker iptables"
ACCEPT_FSL_EULA = "1"
IMAGE_INSTALL_append=" imx-gpu-viv imx-gpu-viv-tools imx-gpu-viv-
   demos clinfo opencl-icd-loader "
OS_DEV_UBOOT_DELAY = "1"
DISTRO_FEATURES_append= " wayland"
```