Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Measurement-based QoS Trade-off Analysis for Over the Air Software Updates

Ingo Schwendinger

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Floriment Klinaku M.Sc, Marcel Weller M.Sc
Commenced:	November 2, 2022
Completed:	May 2, 2023

Abstract

The thesis will concentrate on Quality of Service (QoS) trade-offs of Over-the-Air (OTA) updates with a special focus on the use case of updates on a car fleet.

Updating software of a car fleet efficiently, requires to have OTA updating techniques that can cope with various problems, like the car fleet's size and its spatial distribution. It makes sense to compare different techniques for their QoS trade-offs to find the best technique for given circumstances.

We wanted to understand the key principles of OTA updates. By researching different techniques for OTA updates and understanding their QoS trade-offs, we have developed a decision model for the best OTA update technique in different cases that appear regarding the update of car fleets. Finally, we wanted to implement some of those techniques for the purpose of analyzing them on QoS aspects.

Researching different OTA techniques to find out about their characteristics was a main part of the methodology. Furthermore, we needed to identify which properties were especially important for updating a car fleet. After that, those techniques were analyzed on their QoS aspects. This was achieved by making assumptions on their behavior and by checking them in a simulation environment.

In our results we showed 31 techniques that we grouped into four categories and registered them as Architectural Decision Records (ADRs). Amongst those techniques we argued about the trade-offs that are involved and also showed some trade-offs directly in our experimental setup. There we focussed on code dissemination approaches. We found trade-offs between the quality an administrator observes in update completion time and the response times a car driver observes.

From the results we concluded which technique fits which use case in updating car fleets.

Kurzfassung

Diese Arbeit befasst sich mit Quality of Service Trade Offs von unterschiedlichen Techniken, die bei Over-the-Air Updates eingesetzt werden. Unser Fokus liegt hierbei auf dem Updaten von Autoflotten.

Es ist wichtig Software in Autos effizient auf dem neuesten Stand zu halten. Dies wird nur von OTA Updates ermöglicht. OTA Update Techniken sollten in der Lage sein mit unterschiedlichen Problemen umgehen zu können, wie zum Beispiel mit verschiedenen Flottengrößen oder der räumlichen Verteilung der Autos. Um sich für die beste Strategie in verschiedenen Use Cases zu entscheiden, macht es Sinn OTA Update Techniken auf ihre QoS Trade-Offs zu untersuchen.

Wir wollen die entscheidenden Prinzipien von OTA Updates verstehen. Durch die Recherche von unterschiedlichen Ansätzen und ihrer Analyse auf QoS Trade Offs wollen wir Use Case abhängige Design Entscheidungen erleichtern. Manche dieser Techniken wollen wir für Experimente auf QoS Ebene nutzen, um unsere Argumentation zu verstärken.

Ein entscheidender Teil unserer Methode war das Finden von unterschiedlichen OTA Update Techniken und das Verstehen ihrer Bestandteile. Zusätzlich wollten wir verstehen welche Aspekte besonders wichtig sind, wenn es um das Update einer Autoflotte geht. Dannach mussten diese Techniken auf ihre QoS Trade Offs analysiert werden. Dies ist auf Basis von Annahmen geschehen, die mit Simulationen bestätigt wurden.

Wir haben 31 Techniken und Ansätze in Kategorien gruppiert. Pro Kategorie haben wir die Trade Offs aufgelistet und manche dieser Trade Offs experimentell bestätigt. Die Techniken haben wir als ADRs registriert. Bei den Experimenten haben wir uns auf Code-Verteilungsstragien konzentriert. Dabei wurden Trade Offs zwischen der Qualität die ein Administrator (Zeit bis das Update der ganzen Flotte abgeschlossen wurde) und der Qualität, die ein Autofahrer bei einem Update wahrnimmt (Downloadzeit des Updates im Auto), beobachtet.

Unsere Ergebnisse haben wir genutzt um zu argumentieren, welche Techniken zu welchem Use Case passen.

Contents

List of Figures

List of Tables

Acronyms

ADR Architectural Decision Record. iii

ECU Electronic Control Unit. 2

FOTA Firmware-Over-the-Air. 13

IoT Internet of Things. 7

OS Operating System. 10

OTA Over-the-Air. iii

PKI Public key infrastructure. 16

QoS Quality of Service. iii

RPC Remote Procedure Calls. 12

SOTA Software-Over-the-Air. 13

1 Introduction

In times in which systems become increasingly interconnected, Quality of Service is a vital part of networking tasks. Many applications would not work without QoS. Since updating tasks are also performed over networks, QoS is there involved too. But apart from those classical Quality of Service aspects that are from the networking realm, service quality is also defined in software engineering to define the overall quality of a provided service as part of non-functional requirements.

For many years software in cars was only updated in workshops, plugged into a service computer [20]. This comes with a lot of disadvantages. Such physical interfaces are costly in most cases, which makes it difficult to update many cars at the same time in one workshop. Also, the ever-growing use of software in cars even makes it safety critical to update that software on a regular basis [ABG+17a]: For instance, firmware of sensors has to be free of any kind of malfunction. This can only be achieved by frequent updates that ensure the reliability of those components. The best methods to tackle these problems are over the air updating techniques, since they are able to update many cars in less time and do this also much cheaper. It is estimated that car manufacturers can save up to $35 billion by using OTA techniques [ABG+17a]. To ensure a trouble-free updating process QoS is an important aspect of OTA updates on cars. Therefore, it is necessary to find a technique that can cope with different car fleet sizes and their spatial distribution. For example in high traffic areas, where potentially many cars that have to be updated are on a very limited space, OTA frameworks have to find solutions for ensuring QoS (i.e. reducing packet loss or latency) [ABG+17b]. But also more general: When many cars update at the same time, how can a server cope with that load. Also, other qualities like fault resistance or recoverability are very important to be considered.

To tackle that, we want to find solutions for such cases. Analyzing different state-of-the-art OTA methods to learn about their QoS trade-offs is a key step in finding a method that fits different use cases. It makes also sense to look into other domains of IoT and their use of OTA techniques, to maybe adapt some ideas one can find there. We have grouped different approaches into categories and tried to find trade-offs that can be observed there. For the implementation of measurements we chose certain architectural and design decisions to measure their influence on QoS.

We have found trade-offs which we grouped by categories. We argued about the use cases where those techniques can be applied best. Moreover, we showed some trade-offs of different dissemination techniques by measurements on our prototype. With that we were able to argue in more depth about their trade-offs than in other categories, where we relied on assumptions and literature. In our measurements we found out and argued that the Push approach (push images directly to the cars over the broker without the possibility for the car driver to decline the update) is most effective for safety critical updates. The Optional approach (notification is sent to the cars and drivers can decide if and when to install the update) is most suitable for infotainment updates. And the last of the three main dissemination techniques that we covered was the Polling Intervals approach (cars poll frequently if a new update is available). It is applicable to many use cases,

especially if certain updates are prioritized with a higher polling interval to increase also the quality observed by a fleet manager, when a safety critical update should be installed on every car as fast as possible.

The contribution of this thesis is a QoS aware analysis of OTA Updates. With that it will become easier to choose the best architectural and design decisions for certain use cases as a software architect or developer. Also, fleet owners and car manufacturers get important information in a compact form about design decisions for Electronic Control Units (ECUs) (car manufacturer) or choosing the best code dissemination approach if more than one is offered by a framework (fleet manager).

Thesis Structure

Here, give an overview of your thesis structure.

Chapter 3 – Survey of OTA update frameworks: Here, we give an overview of existing OTA update frameworks and techniques, their trade-offs and how to categorize them.

Chapter 4 – Prototypical Implementation of OTA update strategies: Here, we introduce the strategies used for QoS measurements and how we implemented them in a test prototype.

Chapter 5 – Measurements: Here, we provide information about the undertaken measurements and the used test scenarios.

Chapter 6 – Evaluation: Here, we discuss the observed results and the study design.

Chapter 7 – Conclusion We conclude our thesis by summarizing key aspects, discussing limitations and benefits as well as giving inside on possible future work.

2 Foundations and Related Work

2.1 Foundations

To understand the context of the thesis, basically two topics have to be explained.

Over The Air Updates

As stated before OTA updates are a convenient way to update a fleet of cars. But also in other domains of IoT OTA enables a wide range of possibilities, whether in smart home, industry or smart cities. To analyze OTA updates on certain aspects one must understand OTA updates in general.

Over The Air updates can basically be divided into two parts: Module Management and Rollout.

Module Management concentrates on the compilation of the update modules, their compatibility with already installed modules and a verification step like a simulated update on a digital twin network [BRG+20].

Rollout focuses on security, dissemination, installation and activation of the update. Security is a critical aspect of OTA, as data from the developer of the update has to be authenticated. Otherwise, attackers could inject poisonous code into critical infrastructure leading to safety issues and all kinds of different problems. However, since the focus of this thesis is on QoS trade-offs we will not cover much about security. We will only make assumptions on the influence of different security measures on QoS aspects, without getting into details how they work exactly.

Different techniques for dissemination of the data are likely to have the highest influence on QoS. Coping with different situations regarding the spatial distribution of the car fleet or network congestion because of large updating packets is one of the problems that has to be tackled. But nevertheless it makes sense to have a look at Module Management techniques to, since the way how the modules have been compiled may also impact the completion time of the rollout.

However, Villegas and Astudillo state the there is a lack of standardization for OTA updates [VA20]. So it can be said that there is no blueprint on how to create a suitable OTA update. Rather the update has to be developed according to its use case until a standardization is available.

Quality of Service

Quality of Service is a term mostly related to networking [22c]. It describes the performance of a specific service on a network with a special focus on what can specifically be observed by the user. Inspecting the service quality of a software product requires addressing different categories. Steve Tockey describes software's service quality as part of non-functional requirements that a software

product should fulfill [Toc19]. Different levels of performance can be described by measures like response time, throughput, reliability, scalability etc. Rupp and Pohl define the quality of a software or Service analogously [RP21]. Moreover, they refer to a standard that describes different quality dimensions that define the quality of a software or service [Sta].

One vital part will be Performance Efficiency [Sta]. Here lie the more classical QoS aspects. As already stated OTA updates have to cope with potentially large car fleet sizes. Making sure that even in high traffic areas updates work well, is very important. OTA techniques should strive to achieve high throughput, as higher throughput means also a smaller completion time. How such metrics, like completion time, response time and throughput, behave under certain car fleet scales will be very interesting (Elasticity).

Other classical QoS aspects that can be observed are for instance Packet Loss, Latency and Packet delay variation. All of those network related aspects are helpful to look at, since all of them influence the overall performance of the OTA technique. Only looking on completion time would raise questions about the reason for a slow completion time. That is why it makes sense to take the other metrics into account too.

Also, it is vital to take into account how compatible an OTA technique is with other services that work on the same network. For instance, how are other remote car services affected by a large update? Will they still work without any performance loss due to high congestion of the network cell? If other services fail this leads to inconveniences for the car driver which should be avoided. However, this is more a networking and scheduling related topic which is not covered in this thesis.

Other important questions are: How does the update behave in case of connection loss, server shutdown or other problems that can appear? Knowing about the OTA update's recoverability is important, because it could force the user (driver of the car) to take action himself, which would be a major inconvenience and therefore a reduction in service quality. But even without a complete failure, to which degree can specific OTA updates hold up their intended function [Sta] regardless of having minor faults like high packet loss for instance? All of that can be measured by simulating faults or even complete failures of the system and measuring the time needed to rebuild the connection and to restart (or to continue) the updating process and even the time needed to finish it. Also, other metrics can be measured in this case, like the throughput after restarting and how long it takes to reach full capacity again. Another important measure in this realm is availability since an update should be able to be performed at any desired time. Also, this should be invariant from the position of the car which undoubtedly requires a stable internet access at every place of usage.

Security is also an important aspect of software quality and since it leads to major problems for the user if not applied, also an important aspect of QoS. Only imagine the impact of poisonous firmware code in a car's sensor. Especially in case of self-driving vehicles this is a major issue, which could lead to accidents. Even though security is very important the thesis will only concentrate on the impact of different security measures on the overall performance, while assuming that they work as intended.

Under often changing circumstances OTA techniques are needed that can easily be modified to meet new requirements. For instance, let's say a new networking technique is used to distribute the data, how easy is it to change the OTA technique to work with the new technology. As this is a part of Maintainability it makes also sense to take this into account.

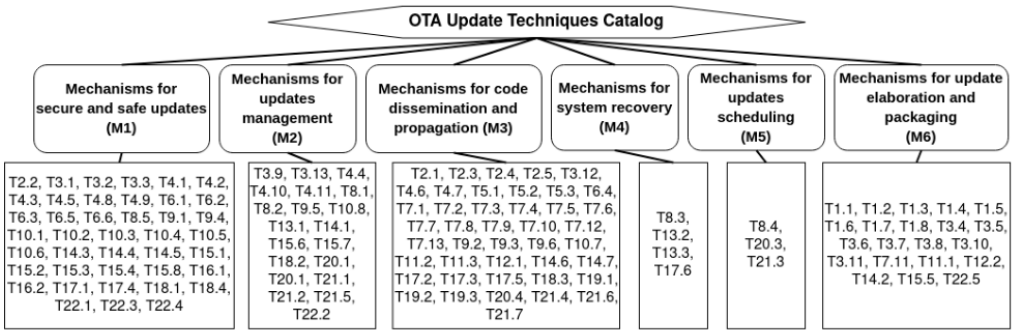

Figure 2.1: Different OTA Update Techniques categorized by their purpose [VA20]

A dimension that we completely dismissed is usability. Since the user only communicates directly with the OTA update when confirming that an update should be performed, usability seems to be negligible. The only role the profits from usability is the fleet manager, which will be covered to a certain extent in this thesis

Analyzing on functional suitability was also not very important, since all the proposed techniques are peer-reviewed works that have most likely already been checked on their functional suitability. But it is necessary to check on the suitability for our given context of car updates especially when taking ideas from different IoT domains.

In the thesis we mainly focused performance observed by mean response time and completion time observed by the fleet manager. We used that to argue about QoS for those different roles. In other realms of OTA updates apart from code dissemination we argued with trade-offs that are known or can be implied by certain characteristics of the observed strategies.

2.2 Related work

Most work so far concentrates on specific issues of OTA updates. Numerous techniques are proposed to ensure the security of the update [SSLK21] [KKC+16] [NL08]. Also, other aspects are considered, like improving the data dissemination technique to speed up updates on cars [MSAA21]. But comparative studies seem to be very rare. The problem seems to be that all techniques only concentrate on a certain aspect while a complete OTA update consists of more than that specific aspect. Also, the already mentioned lack of standardization is a problem which leads to many use case specific ideas.

One study that concentrates on more than only one technique by Villegas et al. [VA20] develops a taxonomy to categorize different OTA techniques in Internet of Things and Cyber-Physical Systems. The techniques have been categorized according to the type of issue that they tackle (see Figure 2.1). Besides security approaches, there are also techniques for update management, the code dissemination, scheduling approaches and mechanisms that concentrate on packaging and elaboration. The paper contributes to the decision process of developers, which approach they should use for their specific use case. It also states in the conclusion that further work has to be done to help developers to compare alternatives. Moreover, it complains about the missing

standardization in OTA updates. Since, most approaches only concentrate on a certain aspect or are specialized to a specific scenario, there is no real blueprint on how to develop a OTA technique for every case. The study definitely provides an interesting overview on some techniques in different disciplines of OTA updates, which helped in starting our own research.

Another highly related work is from Palm et al. [PB22], which evaluates different OTA techniques on deployment time and update rollback functionality. It especially focuses on the question, if different file structures have an effect on the deployment time and if OTA techniques have rollback functions. The underlying context of the work is to find a reliable Over The Air update technique for a recycle station in Internet of Things. Since this work does not concentrate on QoS aspects and the overall focus is on a different scenario it is still different from our work.

As already quoted in Foundations, Bauwens et al. [BRG+20] surveyed the main principles of OTA updates. This is also an example for a paper that has a more general look on OTA update techniques. While Khurram et al. [KKC+16] concentrates on issues regarding OTA updates, Bauwens et al. defines different phases of OTA updates. However, the study does not compare specific techniques. But it will still be helpful for understanding the main principles of OTA updates and make it easier to research for techniques about different issues in OTA updates. But the work gives interesting insights in how to evaluate OTA techniques which helps to do our own thesis.

Aurora Labs [Lab] created a calculator for OTA update costs. As input, it takes fleet size, image size and number of ECUs per car. Also, the number of updates a year are considered. As output, it shows the cost using certain OTA update approaches. It only compares three image compression techniques. Full Image update, binary update and Line-of-Code update. They conclude that their proposed Line-Of-Code update performs best, considering cost. However, the work only concentrates on a certain aspect of OTA while we concentrate on a general overview of QoS trade-offs.

Other than that studies do not seem to concentrate on comparing different OTA techniques and rather on proposing a specific approach.

3 Survey of OTA update frameworks

In this chapter we show the frameworks that we found which provide OTA update capabilities. Most of them are Internet of Things (IoT) management frameworks. Since cars are also IoT devices many approaches are also applicable in the car context.

3.1 Methodology

We surveyed different OTA update techniques by finding frameworks which support OTA updates. Frameworks, which are well documented, support OTA updates and that are already available in a containerized form were the first targets we were looking for. After realizing, that it is more effective to emulate certain approaches in an own setup, we also considered frameworks which were not already containerized. A good documentation became the main metric we used to choose the frameworks. We also tried to cover as many different approaches as possible. We were also using literature to find techniques there. The goal was to find certain trade-offs in QoS. We registered the techniques we found in ADRs. Those ADRs can be found on our GitHub Repository. This is also used to refer in chapter 4 to the techniques we used in our prototype. An ADR is marked by "(ADRi)" with i representing the ID of the ADR. Every ADR label references to the appendix, where all the ADRs can be found.

3.2 Frameworks

In the following we present the frameworks we used for our survey.

3.2.1 Edgehog

Edgehog is an open-source framework for IoT management written in Elixir [Eli] and based on Astarte [Ast]. It offers features like OTA-updates, device status information, geolocation etc.

Basic concepts OTA updates with Edgehog [Srla] are possible with any device which implements certain interfaces provided by Edgehog.

They distinguish between managed and manual OTA updates. In managed OTA updates cars are updated automatically according to their specific group and their System Model. A System Model is defined as a group of devices with the same functionality for users. In case of cars, an example for that would be a car of a specific type which is used in one use case as a police car and in another case as a private car. Even though the cars have the same hardware, they may have different

software installed and are by that in different System Models. Groups are devices that fulfill certain conditions (AD8) defined by the administrator. That means a device group can be every logical group of devices.

One key concept is the base image collection which is tied to a specific System Model. It connects all images that are associated with a System Model and by that with specific use cases.

The next important concept is the update channel. This channel offers base images to the devices. Every device is automatically assigned to a default update channel which can be changed by adding the device to a group.

To start updating devices an update campaign is necessary. It tracks the distribution of a base image to all devices that belong to a specific update channel. Only one campaign per update channel is possible. Also, constraints can be defined for the update, for instance a certain minimum version that is necessary to perform the update on a device. When starting a campaign, the devices that are effected are "snapshot", so only devices subscribed to the update channel at campaign start are supplied with the new image.

Two rollout mechanisms are provided. Push mode (AD3) sends images directly without confirmation to the devices. Optional (AD2) mode waits for confirmation from the device's user.

And finally there is the concept of an OTA operation. It tracks the update process of a specific device.

Also, devices can be manually updated via the Edgehog server's front-end (AD9). But here it is not checked if the system model of the updated campaign fits the system model of the device. This leads to more flexibility in how to update the devices while increasing the risk of incompatibilities if done carelessly.

The concepts are partially applicable to car fleets. Groups do not have as many variable constraints in case of a car fleet. Car fleets are defined in a more static way like a company's fleet or a fleet of a whole manufacturer. Subgroups in fleets could be defined in the way proposed in Edgehog, since the management of specific car types of a certain purpose would then be automated. It would be easier to change a cars group membership, since already the change of a certain attribute of the car would change its System Model and by that how it is effected by updates.

Architecture Edgehog uses Astarte as main backend (figure 3.1). Astarte [Ast] is an open-source platform for IoT data management. Edgehog offers an API for administrators with a frontend and also a GraphQL interface for third party applications. Since the frontend also uses the GraphQL API the same operations are possible for third party applications. As database PostgreSQL is used. The communication with the devices is fully managed by Astarte using VerneMQ as broker. Data is exchanged between Edgehog and Astarte via REST-API and Astarte Triggers, which define under which conditions data is sent to a specific component or device. For the communication with the devices the MQTT (AD10) protocol [mqt] is used. It is very useful for IoT tasks, due to its low packet size, which makes MQTT very effective in low bandwidth environments. Moreover, MQTT can carry all possible data types. It is reliable in providing QoS (in this case network related) by guaranteeing delivery.

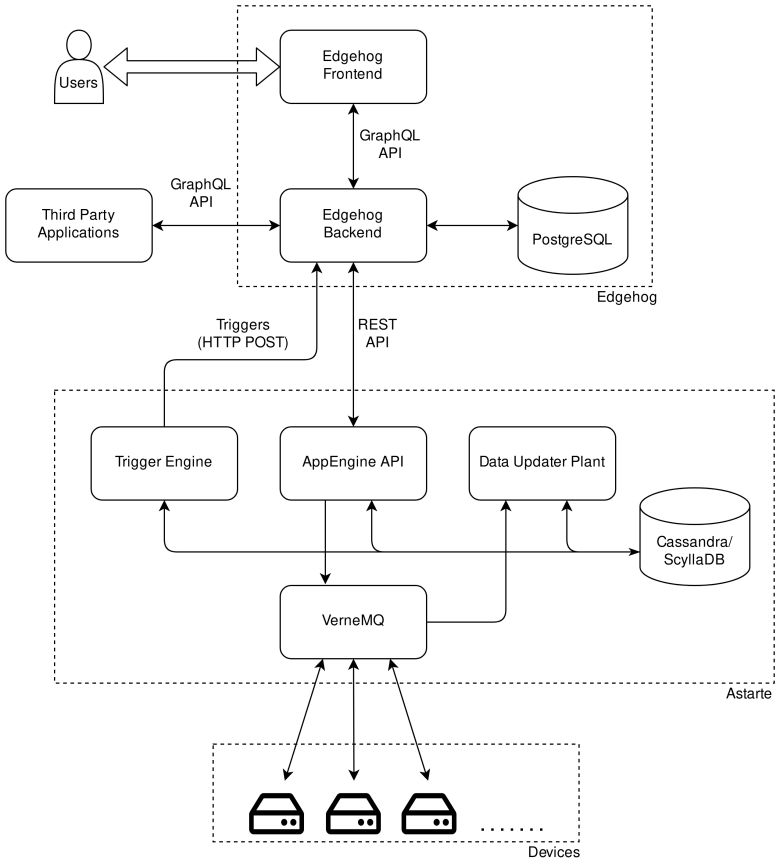

Figure 3.1: Architecture of Edgehog [Srla]

3.2.2 Mender

Mender [Men22] is an open source OTA software updater for embedded Linux devices. Mender proposes its simple client / server architecture.

Its focus lies on ensuring robustness i.e. if power is cut during an update, rollbacks should be ensured. It uses two server side APIs: Device and management (AD11). The Device API is for device originated requests and the Management API for users that manage devices, and updates.

Mender provides two modes of client operation: In managed mode devices check for updates automatically themselves. Installation, confirmation of the update, reboot of the device and other tasks are fully managed by the Mender client. In standalone mode updates are initiated locally (not OTA). They support full filesystem or application updates.

Security and Robustness In order to ensure a secure and robust update process, Mender needs additional metadata alongside the raw bits of the update payload. For that, information like the update version and a checksum of the payload. The update version ensures an easier rollback to older versions, since the version is directly attached to every image. A checksum ensures that the image file was not modified after delivery (AD12). Also, it provides authentication by comparing

the checksum to an original checksum to ensure that the image was in fact sent by the Mender server. The device type with which the update is compatible is also covered in the metadata, to prevent compatibility issues during installation. Additionally, information about provided fields from the mender artifact, that is fields that are saved on the device during the process, are provided, as well as depending fields that must be present on the device for a successful update. Mender achieves robustness by providing a dual redundant scheme (AD13), which ensures that a device is always able to return to a working state after a failure. For that, two partitions are used for the device's Operating System (OS). One, where the OS runs on and a second, which is inactive and used to start the new updated OS. If anything fails in the boot process, the device can still rollback to the previous version, since that version is still actively running on the second partition. The only downtime when using this approach is during reboot to the new OS version. When an update was successful, it gets committed, that is setting a flag in the bootloader to indicate the update was successful. The filesystem needs to be stateless in that approach, since the whole partition is overwritten in every update.

When applying application updates, Mender uses Update Modules which represent any type of software used in the OS. In an update module also the installation process is directly defined. One key element there is the installation of the software itself, but also rollback instructions or other actions can be defined there.

To provide another layer of security, Mender offers a gateway, which can serve as a proxy between Mender server and client (AD14). Additionally, the gateway helps in balancing load more easily to different server instances. Another advantage is the improved proxy compatibility i.e. Mender works more easily in private networks.

The Mender client does not use any open ports. Every communication with the server is started by the client (AD4). This is an advantage in security, since connections are harder to be compromised, by offering fewer endpoints. Mender also uses device groups and artifacts to define which devices need which update image.

Mender uses a REST-API to communicate with the devices. Every communication with the server is started by the devices. The connection is secured with TLS (AD15). The system's root certificate authorities are used to verify the server identity. Devices are authenticated by pre-shared RSA keys with which they sign all their requests.

Device Groups Mender supports two type of device groups. The first one are static groups (AD7), which are lists of device IDs. A device can only be part of one group. Groups only exist as long at least one device belongs to them. The other group type is the dynamic group (AD8), which is defined over a set of filters. Devices that match the defined constraints are effected. Such groups can change over time when the devices change their properties.

Deployments Deployments are used to perform updates on groups. A deployment consists of a release, the devices that are effected (generally a group) and the number of retries if an update fails. When updating a group the update is performed as snapshot. That is, only cars that are part of the group at deployment start are supplied with the update (AD5). Deployments to a dynamic group never finish, since every car that matches the constraints of the group at any time will be supplied with the update (can be compared to the continuous approach in section 3.2.5 and the

continuous approach used in our prototype). There are different states a deployment can live in. After deployment start, it is in the state “pending”. After the first device starts requesting a new image, the deployment reaches the state “inprogress”. When all devices finished their update process, the deployment reaches the state “finished” no matter if the process failed or not. A deployment on a dynamic group behaves differently. It remains in the “pending” state until the administrator explicitly stops the deployment or until the number of supplied devices matches the optional parameter “maximum number of devices”.

Delta Updates Mender uses delta updates automatically in all supported artifact versions (AD6). That is, Mender sends only the data of changed blocks of the old compared to the new update image. This approach is especially helpful in reducing bandwidth needed, since fewer data needs to be sent out to the devices.

3.2.3 Thingsboard

Thingsboard [thi] is an Open Source IoT management system. Since version 3.3 it also provides OTA update support. Thingsboard notifies devices about new available updates and provides a protocol specific API to download the firmware. The platform tracks status of the update and stores a history of the updates.

They propose five main qualities:

- Horizontal Scalability
- Fault Tolerance, since there is no single point of failure in the architecture
- Efficiency and Robustness: They propose that a single server node can handle tens of thousands of devices and a whole cluster millions of devices
- Customizability: new functionality is supposed to be added easily
- Durability

The proposed qualities refer to the overall IoT functionality. All but the third point are also included in the OTA functionality.

Architecture One major component of the Thingsboard architecture is the Transport Layer. The transport layer provides APIs for four different protocol types: MQTT, HTTP, CoAP and LWM2M. With MQTT also Gateways are supported as in Mender (section 3.2.2), which can represent a whole group of devices. With message queues between the Transport Layer and the Core Services (figure 3.2), it is ensured, that all messages arrive. Devices only get an acknowledgement after the messages are acknowledged by the message queues. The Thingsboard core services are responsible for handling REST-API calls and WebSocket subscriptions. They also manage data, like device state and sessions. The Rule Engine is responsible for handling incoming messages. Also, a Web UI is provided for administrational use, which uses the REST-API and WebSockets. The message queues are proposed to be crucial for managing high load. Thingsboard uses topics to communicate over the queues, for instance if a message is sent from the transport layer to Thingsboard core, it is published to the topic “tb_core”. Before Thingsboard used message queues for communication, they

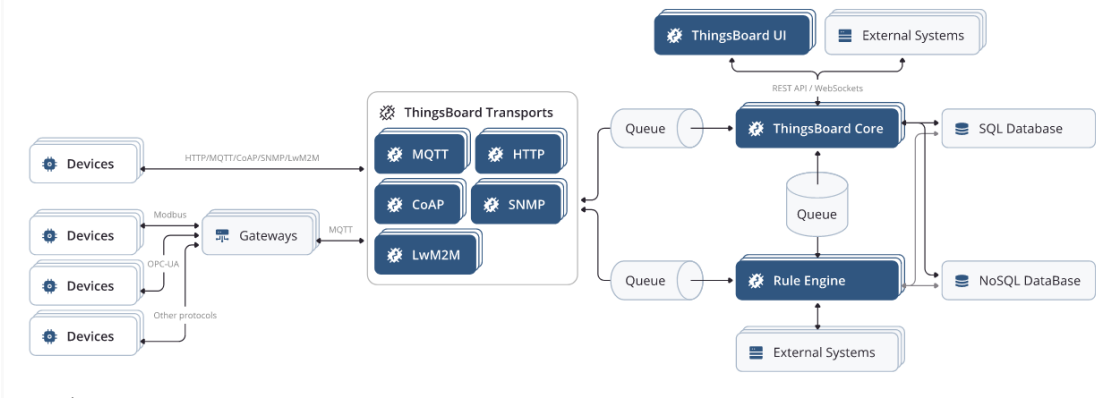

Figure 3.2: Thingsboard's architecture diagram, taken from the official docs [thi] All blue components are proposed to be scalable and fault-tolerant.

used gRPC [Aut]. While Remote Procedure Calls (RPC) (AD17) is generally more effective when it goes to low-latency and real time interaction, Message Queues (AD16) provide better reliability and fault-tolerance. If one component fails for a certain time, the message is not lost. When it goes to production, Thingsboard offers two modes of deployment: A monolithic architecture (AD18) where all components are coupled tightly together in one deployment. This makes it easy to maintain and run also in small setups with less hardware resources. The downside lies in less scalability and availability of a monolithic architecture, managing large device fleets, while also maintaining availability becomes impossible with a monolithic architecture. For that purpose they also provide the microservices architecture (AD19) provided in figure 3.2.

OTA-Updates To start an OTA-update an administrator needs to upload an update package to the OTA repository. This update package can be assigned either to a device profile or to specific predefined devices. That means that the administrator either needs to give a list of device IDs, to describe which devices they want to update or give a device profile which is updated with that package. A device profile is defined in Thingsboard as set of characteristics that devices that are part of that profile fulfill. Which devices are part of such a profile is automatically detected by Thingsboard. So the great advantage of using OTA-updates with device profiles, is that the administrator does not need to select specific devices he wants to update (AD8), but can simply decide to update devices with certain characteristics. All of that increases the usability for the administrator. When a device is to be updated, it receives a notification (AD2). This notification consists of a link to a protocol-specific API for the package download. The update state, that is the percentage of devices that already received their update, can be observed from a dashboard.

An update package consists of a title, version, corresponding device profile and a type. The device profile mentioned here is only for checking purposes. When starting an update for a device profile, the administrator can only choose packages, that are compatible with the profile. Devices in that profile only get updated after a direct assignment from the administrator. The type of the update can either be Software or Firmware. Apart from those mandatory parameters, there are also optional ones like a checksum algorithm and the respective checksum of the package (AD12). If the administrator does not set these two, SHA-256 is chosen as checksum algorithm.

There is only a minor difference in the Firmware-Over-the-Air (FOTA) and Software-Over-the-Air (SOTA) implementations in Thingsboard core. This difference is not further described in Thingsboard Docs. While Software in this case refers to applications or other software running on an OS, Firmware refers to the whole OS. FOTA updates are more complicated than SOTA updates, since the device needs to be restarted during the process and validation of the firmware image needs an additional bootloader at the device.

The update process consists of several states:

- QUEUED: The update notification is in the queue but not yet pushed. The queues are released in a controlled way. This is to reduce peak loads. The default release rate is 100 device notifications per minute. This can be configured by the administrator.
- INITIATED: The notification is already pushed to the device.
- DOWNLOADING: The device started downloading the update package.
- DOWNLOADED: The device finished the download of the update package.
- VERIFIED: The checksum of the package was verified by the device.
- UPDATING: The device started the actual update process.
- UPDATED: The device finished the update process successfully.
- FAILED: Either verification or the update process itself failed.

Supported protocols MQTT [mqt], HTTP [Gro], CoAP [SHB14] and LWM2M [All] are supported by Thingsboard. The protocols have certain trade-offs which we will describe in the following.

- MQTT (AD10): MQTT is a lightweight, pub/sub protocol. It is efficient regarding bandwidth and power consumption. This is very helpful when managing resource constrained devices. But MQTT has no built-in support for device management or security, this needs to be implemented separately.
- HTTP (AD20): HTTP is very widely used and well-supported. It offers good support for security and device management. The downside is its reduced efficiency in bandwidth and energy consumption, resource-constrained devices will struggle with.
- CoAP (AD21): CoAP is a lightweight RESTful protocol especially made for IoT management. It already offers important IoT device management functionality. One example for that is an extension for CoAP, which enables group communication via multicast. A device group can be directly accessed without further implementation. Other than that, it is similar to HTTP but uses UDP instead of TCP, making it more suitable for low-power devices. CoAP is not as largely supported as HTTP, which is one downside.
- LWM2M (AD22): LWM2M is also lightweight and specifically made for IoT applications. It has even more built-in IoT functionality, like device registration, software updates and remote management. It has even less support than CoAP. Working with existing IoT devices or platforms may require additional implementation.

3.2.4 SWUpdate

SWUpdate [SCL+21] is an open-source OTA software update framework. It proposes a flexible and robust way of updating a variety of Linux-based devices.

The way that images are sent from server to client is not much different as in the already described frameworks (SWUpdate also uses an approach with polling intervals (AD4)). That is why we concentrate on installation strategies proposed in SWUpdate to provide fast and robust updates.

Update Strategies (Client Side)

Single copy - running as standalone image The new firmware is created as a standalone image (AD23). This image can run on the device independently. When a new update image is ready the device switches into a special mode, where it is allowed to accept the new image. The new firmware is downloaded and stored on the device, replacing the old firmware. This is completed with a reboot. The whole approach is most helpful, when having devices with limited memory. By creating a self-contained firmware image that includes both the bootloader and the application firmware, this approach simplifies the update process and reduces the risk of errors or complications.

Double copy with fall-back This approach involves creating two copies of the firmware installed on the device (AD13). One version of that firmware is updated to the new version. If something goes wrong with the update, the bootloader can choose the old working firmware. For that, it is necessary to split the memory into multiple partitions. This enables a quick and reliable fallback option. The downside is the increased memory consumption.

Double-copy with rescue system This approach involves the same steps as in the "Double copy with fall-back" approach. In case none of the copies can be started, the bootloader will start a rescue system, which is stored on a different memory or partition (AD24). This third copy is helpful as a further layer of protection against any issues that may arise during the update process. On the other hand, it increases the memory consumption even further.

Split system update with application update If an update can be split into an OS update as well as an application update, the update can be performed in two phases (AD25). The first phase involves the update of the OS, where a certain downtime is created, due to the need of a reboot. The second face covers the application update, where no reboot is needed. This approach has the advantage of offering the possibility to verify the OS update as well as the application update separately. Another advantage is the reduction in downtime of the device, since the OS update can be performed while the device is still running. Only for the activation, a reboot is necessary. After that the application update can be performed, with no further downtime. If application and OS update would be in one package, the device would need to shut down for the whole update process.

Note that SWUpdate is customizable, other strategies than the already mentioned can be implemented and integrated into SWUpdate.

Delta Approaches

SWUpdate has no delta support yet, but in the docs certain approaches are discussed for their compatibility with SWUpdate. The general setup and the goals of a delta update are already explained in section 3.2.2. We will not analyze the delta approaches for their applicability to SWUpdate but to OTA updates in general.

Librsync Librsync [Poo] uses a set of checksums for the files to calculate a delta file (AD26). If a checksum of a block in one file is equal to a checksum in the other file, the two blocks are likely the same. The blocks that do not have a matching partner then get further analyzed for their differences, which are written into delta blocks. This works fine if the differences are very small. If the differences reach a certain threshold, the delta file size is even larger than the target file size. One advantage is, that the algorithm only needs a small amount of memory on the server. Librsync is most effective when having small images and a resource constrained server.

Xdelta Xdelta [Mac] (AD27) is in general very similar to Librsync. Instead of a rolling checksum algorithm it uses binary diff operations to detect redundancies between two files. Also, the compression is handled with a different algorithm. Overall Xdelta is also most suitable for image files with only minor changes to their previous version.

Zchunk ZChunk [Die] splits every update in chunks (AD28). These chunks are stored separately on the server. With metadata files it is defined which chunks are part of an update image. If only certain parts of an image are changed, only a few chunks need to be sent to the devices, reducing the bandwidth needed by a lot. Chunks are verified with a hash. The creation of the deltas may take longer than in the already mentioned approaches, due to the fragmentation of the data into chunks, which may be stored in different locations. Also, ZChunk needs more memory on the server than other approaches. The compression ratio is very high. For OTA updates it depends on the use case if Zchunk is helpful. When having a server with lower memory capabilities, ZChunk is not the right choice. If the time saved by reducing the amount of data to be sent and also reducing the installation time is lower than the time lost by performing the delta algorithm, ZChunk is also not the right choice.

3.2.5 AWS IoT OTA library

AWS offers a library in its IoT management system which enables OTA updates [Ama]. The key element in AWS are Jobs. Jobs are administrative tasks, effecting a device, that can be performed remotely. One possible type of task are OTA updates. A job normally consists of a JSON file which describes the actions that should be undertaken by the device. In case of an OTA update job, a link to an S3 storage containing the update image is sent via MQTT to the devices [AD10, AD2]. The devices are part of a device group, which is defined in the same way as in Mender (section 3.2.2). Devices install the new update image after downloading and sending out an acknowledgement to the server that the job is finished. For an administrator the job only finishes completely if it was started as a snapshot. That is that only devices, which were part of a device group when the job was started, get the notification about the job (or about the OTA update in our case). Jobs can also be

executed continuously (AD5), that is, if devices are added later to a group, they still need to execute the job. This technique of sending out notifications over a broker and referring to a link where the update file can be downloaded, will be used in chapter 4 and 5 for the QoS measurements. Also, the idea of distinguishing between a continuous and a snapshot update will be used.

3.2.6 Uptane

Uptane [KML+; Upt22] is a standard for secure OTA updates of car fleets (AD29). It consists of a basic architecture that needs to be in place. Further details in implementation are left open.

Goals of Uptane

Uptane aims to provide a secure way of delivering OTA update images to cars. One important goal is to make sure that an image that is sent cannot be tampered with during transition. Also, Uptane provides resistance against typical attacks like rollback, man-in-the-middle or denial-of-service attacks. Uptane only defines the security measures that need to be in place, the rest of the implementation is up to the individual developer. Uptane strives to be a unified standard for secure OTA updates in automotive industry. This shall lead to an overall increased security and safety of vehicles by reducing the risk of cyberattacks. Securing only the connection with TLS for example is not sufficient, since only man-in-the-middle attacks are prevented by that. The aim is to become compromise resilient, so that the repositories are more difficult to compromise, while not losing the ability of doing customized updates for different vehicles. So Uptane overcomes the need to choose between on-demand customization and compromise-resilience. This is explained further in the following.

Basic Architecture

Uptanes architecture consists of the following components:

- Software Repositories
  - Image Repository that holds binary update images and their metadata
  - Director Repository which has access to an inventory database. It can sign metadata for images in the Image Repository.
- Tools for generating Uptane-specific metadata
- Public key infrastructure (PKI), which supports the metadata production and signing.
- A secure time-server for the ECUs.
- One ECU per car that is capable of downloading images and metadata from the servers (called Primary).
- One ECU per car that is able to verify signatures of metadata and images (can be the same as the Primary). It also needs to be able to verify images and metadata for all secondary ECUs in the car, that are not directly connected to the server. The metadata and images are then distributed by the primary ECU to the secondary ECUs of the car.

- Secondary ECUs that are either able to perform full or partial verification of metadata.

Metadata types

Uptane consists of different metadata types, which serve the aim of providing a hierarchical structure, which describes an update.

- Root metadata: Includes the public keys of the manufacturers, suppliers and other trusted entities that are involved in the update process.
- Targets metadata: Describes the software packages available for update and their corresponding hashes and signatures. Also, it may contain additional information like compatibility or dependencies between different update versions for different ECUs.
- Snapshot metadata: Snapshot of the current state of the target metadata. It is signed by the root metadata and used to verify that the targets metadata is valid and not outdated.
- Timestamp metadata: Provides a timestamp for every version of the snapshot metadata. It is used to verify that a system gets the latest version of an update.

There are also metadata types which focus more on the transition of signing rights to other roles, which is useful, if a manufacturer wants to give signing rights to companies for their car fleets.

Update Process

In the following the update process of one vehicle is described. Any group management as in aforementioned frameworks can be implemented on top, but the following steps need to be applied for every car that is updated.

The Director repository identifies the vehicle either by a manifest that is sent by the vehicle's primary ECU or with other mechanisms like a two-way TLS handshake with unique client certificates provided by the PKI that is in place. With the vehicle identifier the Director repository queries the inventory database to check for relevant information about the vehicle's ECUs. The provided manifest by the vehicle needs to be checked with the inventory database if all information is correct. If any check fails the update of the car has to be stopped. The minimal checks that should be involved are the following:

- Set of ECUs that are part of the car are also registered on the database for that car.
- The signature of the manifest is valid when using the primary ECUs key.
- Signatures of secondary ECUs are also valid with their respective keys.

ECUs use a nonce for requests to the server to prevent from replay attacks. This nonce has to be checked if it was used before. If that is the case the request needs to be dropped. The Director Repository extracts information about installed images on the car's ECUs to check for new updates available and provides a set of images that are to be sent to the car. Another important step for the Director is to check for dependencies and conflicts between the images used in the different ECUs. Images can be encrypted if needed, otherwise it is sufficient to send the image in an authenticated

way. Finally, the Director creates new metadata representing all the images that need to be installed on the car's ECUs. The car then receives the metadata with which it can verify the images that it receives additionally.

The ECUs verify the update (images and metadata) either with full or partial verification. The full process of validating images and metadata will not be explained her, since it involves a plenty of steps, but we will break down full and partial verification to a certain extent.

Partial Verification The only metadata verified with partial verification (AD31) is the target metadata, which is information about the image to be installed e.g. filenames, hashes of the image or file sizes. In detail, the ECU checks if the image version in the target metadata matches the version number of the image in the snapshot metadata. Snapshot metadata consists of information about all software installed on the different ECUs of the car. Also, it is checked if the previous installed version number is equal or lower, to prevent rollback attacks. To prevent freeze attacks (attacker wants to block update action by spamming old update requests, which update to the same version again) an expiration timestamp is used. If the current time is higher than the expiration timestamp, the update is aborted.

Full Verification With full verification every part of the metadata is checked for validity, that is validating all signatures (AD30). The main focus is to compare the metadata of the director repository with the metadata provided by the image repository. This adds another layer of protection, since an attacker would need to compromise two servers to be able to successfully temper with the update process. Also, if one key is compromised, the attacker is still not able to do anything harmful since on other levels of the signing hierarchy are also keys that are needed to sign the update.

The main trade-off here is the use of resources against increased security. Verifying all the metadata in full verification takes a certain amount of time and also needs memory for all the metadata to compare the signatures. With certain resource constraints, partial verification is the right choice, since it needs less bandwidth and less memory and computational capabilities. For highly safety critical updates it makes sense to use full verification to prevent any threat to the car and its driver. It is sufficient if only a primary ECU does full verification.

Main advantage

The main advantage lies in the distribution of keys. Data is not only signed by one set of keys stored in one server, but also by keys that are stored offline and are used on a second server for signing. In case of Uptane the second server is the image repository, where all metadata about those images is signed by offline keys that are not directly stored on the server. This makes it much more difficult for attackers, even if they are able to compromise one server, they are still not able to temper with the update process. This two server principle allows compromise resilience while still preserving on demand customization. If only one server would be used e.g. one server that signs with a key that is stored in the server, the whole system is not compromise resilient, since an attacker only needs to compromise one server, to have full control. Having only one server that provides signing with an offline key, that system would be compromise resilient, since keys are not stored on the server, but in an external memory like a USB stick. The downside is the loss of on-demand customization, since in this setup metadata is signed only ones, and it takes a certain effort to sign new metadata.

If for instance for the same car type but with two different purposes an update is planned, it is desired to have different installation configurations for the different cars. This would be much more complicated and time-consuming in a setup with only one server with offline keys. That is why Uptane proposes an approach with two servers, since with the combination of offline key signing for the image itself and on demand signing of configuration metadata, the system is compromise resilient and still supports on demand customization.

OTA Connect

One implementation which follows the standard provided in Uptane is OTA Connect [HER].

Device Provisioning OTA Connect follows in its security policies and in its architecture the guidelines that Uptane provided. Devices need to go through a provisioning step, which ensures a secure communication. This is done by receiving a provisioning key from the server, which is then written into the initial disk image that is installed on the ECU. This key is then used when connecting the device to the server for the first time. The server creates a new key pair for the device and sends it together with a signed certificate to the device. The device saves its key pair and the server deletes the private key of the device from its memory.

Update Process Devices poll periodically if a new update is available (AD4). This process is effectively the same as already described in Mender (section 3.2.2). The device receives a metadata file, which is checked for validity with either full or partial verification. If the checks were successful, the image is pulled. To offer an approach similar to a delta update, OTA Connect offers OSTree combined with TreeHub as a way of only downloading the changes applied to the image. OSTree is a git-like repository for managing and delivering versioned file systems (AD32).

Installation process OSTree is also a key part in the installation process if it is chosen. Its git-like structure allows the download of only certain files and binary diffs that are actually needed for the update. For that, a metadata file, containing the respective commit identifier, is sent to the device which then pulls the necessary files from the TreeHub. Another approach is the dual bank approach (AD13), which is similar to the Double copy with fall-back approach in SWUpdate (section 3.2.4) and the approach used in Mender for installation (section 3.2.2).

The OSTree approach combines both the advantages of the dual bank approach while also providing a way to perform delta updates. The dual bank approach has the trade-offs described in section 3.2.4.

3.3 Observed QoS Trade-Offs

Here we will categorize the techniques and list the trade-offs, which we observed in the aforementioned frameworks.

3.3.1 Technique Categories

The found techniques can be categorized into certain groups. We use a similar structure as provided in the related work of Villegas et al. [VA20] or the work of Bauwens et al. [BRG+20]. We propose the following structure:

- **Update Management**: Containing all techniques, that concentrate on managing fleets and update versions.
- **Update Security**: Techniques, that concentrate on the security and authenticity of the update process.
- **Code Dissemination**: Containing techniques, that focus on the dissemination of the update files.
- **Update Installation**: Techniques that provide a way of installing the update image on the car's ECUs

3.3.2 Trade-Offs

Update Management

One main difference we have observed in the frameworks was the handling of device fleets. Mainly they can be distinguished if a device fleet or group is defined statically (AD7), or if it is defined along certain constraints. Having a fleet described by constraints (AD8) makes update management easier when managing multiple fleets with a certain purpose. Switching devices between fleets only requires changing parameters associated with that devices, and their automatically change their group affiliation. It may be the risk involved, that if an administrator decides to change a certain parameter and is not aware of its consequences it could lead to situations where devices (or in our case cars) get updated in a wrong way. If the constraints are correctly defined this should not be a problem. The overall usability can be increased with dynamic groups, but groups are more easily observable if they are statically defined by an administrator. But it is important not to forget that an administrator also needs to put more effort into fleet management when fleets are defined statically. In general, it makes sense to argue, that having multiple fleets where devices often change their purpose, it makes sense to use dynamic groups, while in environments where devices typically remain in one fleet it makes sense to use static grouping.

Another important aspect is the continuity of updates. When a group is updated, will updates only effect the devices that were part of the group at the exact time the update started? Or will also devices be served, that were added later. A main use case for a continuous update (AD5) are important security updates that should be applied when being part of a certain fleet. Imaginable is also software, which should be installed on every device of that fleet. Examples for that can be found in section 6.3. When having more static fleets, where devices change not as often their group affiliation, or in case of optional updates, snapshot updates make more sense to be used, since it reduces the management effort, that is necessary on the server-side.

Update Security

Regarding Update Security, we observed typical approaches to secure the update channel, like TLS (AD15) or signing of update files (AD12). Those are approaches, which are well-known and easy to integrate. However, Uptane showed the need of having a higher security standard (AD29), when it goes to updating car fleets. Since secure connections only reduce the risk of a connection compromise, it can be fatal if an attacker is able to compromise a whole server or a set of keys. To prevent this Uptane proposed its approach of making OTA updates compromise resilient. If OTA updates are used in a large scale in the automotive industry, it is crucial to have at least equivalent measures into place that secure the update process properly. The trade-off involved in Uptane is the selection of the correct verification approach. Choosing full verification (AD30) leads to the highest possible security, by involving a whole set of different keys and signatures that are part of a hierarchical PKI in validating the update metadata. Even if the attacker is able to compromise a certain server or a subset of keys, they are still not able to temper with the update. When having less resources it is the right choice to use partial verification (AD31), which is only checking the target metadata. The overall security is reduced, but less resources are needed. If there is a primary ECU which does full verification and forwards the metadata and images for his secondaries, it should be sufficient to do partial verification in the secondaries. With that an attacker has only one possibility left: Compromising the primary ECU. But even then they are only able to temper with that one car. We see a typical trade-off in having higher security, but for the cost of needing more resources. Also, it can be assumed that a full verification has a higher overall time consumption. Comparing different security approaches on their impact on efficiency of OTA updates can be the aim of future work.

Code Dissemination

Our overall focus, especially in the implementation of tests was on code dissemination. We have seen certain approaches, which can be listed as follows:

- Frequent polling of the devices for new updates (AD4).
- Sending out a notification, which either automatically leads to the download of a new image file, or the device user (in our case car driver) can choose when to install the update (AD2).
- Offering the image directly over a broker (AD3).
- Using delta updates, to reduce the amount of data to be sent (AD6).
- Usage of different protocol types

Comparing some of those approaches is a main part of our measurements in the following chapters. But there are also certain trade-offs that are described in the docs of certain frameworks.

Frequent Polling When devices poll for new updates frequently (AD4), it adds a layer of protection, if this is the only communication with the server. When only devices initiate the connection with the server and do not have any open ports, the surface of exposure of a potential attacker is reduced. A downside is a reduction in responsiveness. Important information, like a new software update will be delayed by a certain time, depending on the size of the polling interval.

Also, load can be controlled in a certain way, since increasing the interval leads to fewer requests per second and by that to a reduction in load on the server. All of this is elaborated in more detail in chapter 6.

Notify then download This approach (AD2) is often used across different frameworks. Depending on the use of any backpressure or load control mechanisms, its behavior can be very different. If car drivers can decide for themselves when to download the image, the load is hard to project. If cars immediately start downloading, a peak in load is created which can be difficult to cope with for the server. Even though a whole fleet may be updated faster that way, the mean response times should increase for the individual driver. When it goes to important security updates this may be the right choice because the fleet would be more secure at an earlier time, compared to a scenario where the throughput is reduced with a load control mechanism. On the other hand there may be no choice for the server then to limit the load that is applied. One advantage of that approach is that cars typically are notified earlier than with an approach using polling intervals. More on that can be found in chapter 6.

Image over broker Depending on the protocol and the chosen broker, this may be the right choice (AD3). Especially with smaller images that can make sense. Having larger images the typical approach is to offer a certain API, where to download the image from. Sending it over the broker reduces the number of calls to the image database, which can reduce the overall update completion time for the fleet. This is also elaborated further in chapter 6.

Delta Updates A typical way to reduce installation and dissemination times is the use of delta updates (AD6). We observed this in nearly every framework we looked at. Different approaches were already described in section 3.2.4 and 3.2.6. It mainly depends on the size of the images and how similar they are compared to each other. Many delta approaches struggle to cope with high differences, leading to even larger image sizes than the original image size. Also, it is important to note that a delta approach is especially effective if only one delta image for a whole fleet needs to be created, so every device / car needs to be in the same update state. In the worst case of having every client on a different software, one image per client would be needed, leading to much more time needed for delta creation. For further analysis see chapter 6. The resources that a server has available are also important to consider. Certain approaches have a higher memory consumption, others need more computational resources.

Protocol Types The trade-offs between different protocols were already discussed in section 3.2.3. The main trade-off lies in the already built-in fleet management functionality against the compatibility of the protocol. For instance LWM2M (AD22) supports device registration and OTA updates but is not as compatible with typical software used in IoT as for example HTTP (AD20) or MQTT (AD10). Also, the resource and network constraints have an impact on the selection of the correct protocol, since some protocols are much more lightweight than others.

Update Installation

The overall trade-off that was observed here was between memory demand and higher fault-tolerance / recoverability. Working on only one partition (AD23) is cheaper and easier to handle, but in case of power cut-off or any issue with the installation the ECU can be broken, which can in the worst case lead to the replacement of a whole part of the car. In most cases at least two partitions (AD13) should be used for recoverability. In a few cases, where the ECU can be cheaply replaced it can be fine to choose a one partition approach. The different approaches were covered in the following sections: 3.2.4 3.2.2 3.2.6.

Architectural Styles

Apart from specific strategies chosen for the OTA updates, there are certain architectural decisions made, that have an effect on the updates but are also generally helpful in different use cases. We have seen the trade-offs between monolithic (AD18) and microservices architecture (AD19) (section 3.2.3): A monolithic architecture is easier to maintain and suitable for smaller setups. When needing large scale fleet management a microservices architecture makes more sense, since it can be scaled. On the other side it needs more expertise to maintain it.

Another trade-off we saw in using RPC (AD17) against using message queues (AD16). RPC offer a better performance in low latency environments, while message queues have advantages in reliability and message persistence.

4 Prototypical Implementation of OTA update strategies

In this part of the thesis we will explain which strategies we chose for QoS measurements and how we implemented them.

4.1 General Information

For the measurements certain dissemination techniques were chosen from aforementioned frameworks. Previously it was planned to deploy open implementations of frameworks directly on a Kubernetes cluster and carry out the measurements. It turned out to be not trivial to do that, since all observed frameworks use different customer side software. Especially in case of general IoT management frameworks, OTA-updates are only a small part of the overall functionality, which would have had an impact on the performed measurements. This wide range of functionality leads to extensive back-ends, which would claim a substantial amount of resources. Documentations of those frameworks also do not fully clarify which detailed actions are involved in various dissemination approaches, which makes it difficult to compare them. The simulation of clients for load tests would have been far more complicated since the processes for registration and authentication are different among different frameworks. To simplify the measurements it was chosen to mock the described techniques in a simple implementation.

4.2 Terminology

For the implementation certain concepts were chosen to be covered and simplified appropriately. We defined the following terms according to the frameworks we observed without covering every aspect of their characteristics.

Group / Fleet An overarching scheme in many IoT management systems was the use of groups of different devices [22d; Man22; Men22]. In case of car fleets, a car group can represent a fleet of a specific car type or a fleet that for example belongs to a specific company or cars that are grouped by a certain purpose. Also, cars could be grouped by location or proximity, to distribute updates more easily, since with that, data has a shorter path to its clients. A group is in our test framework defined by an ID, by the cars that are part of it and the current as well as the previous image that was installed on those cars (AD7). It can be argued that it makes sense to save more than only one previous image ID for all cars to enable rollbacks, but for our simplified construction one previous image is enough, which will be explained later. Groups can be changed by adding new cars, or cars

can leave a group and join another. A transition of a car between groups is backed by use cases like cars that switch from one purpose to another. For instance, if a car gets sold by a private owner and a company decides to add this car to its car fleet, then the company can decide to take over the management of the car's software from the manufacturer and install certain third party software like a communication system between the company's cars. In our setup the car type is the lowest common denominator of cars in a group, since we assume that one image is sufficient to supply a car, which directly assumes a certain technological similarity. However, in reality there are many more logical layers that can be updated in a car.

Rollouts Another term used is the rollout, which is the process of updating cars. It is defined by an image file, a respective car group and the used rollout strategy. A rollout is started at a certain time for the defined car group with the underlying strategy. Also, there exists a special rollout version called a continuous rollout, which not only updates cars that are in a group at the moment of rollout start, but also after the rollout has been started. With that it is possible to bring all cars that are added to a group automatically to a certain update state, without the need of creating a new rollout every time.

Car A car consists of an ID, a corresponding group and the previous as well as the currently installed image represented by their IDs. Moreover, it is assumed that a car only belongs to one group. In reality cars can also belong to more than one group, but this is not relevant for the overall performance. For that to work in general, a decision algorithm would be needed to decide which image is installed in case of multiple rollouts effecting one car. That is why we chose to simplify that. Initially the value of corresponding car group is -1 which implies that the car does not belong to a group. Those are the only properties covered, since we only concentrate on the behavior of different dissemination techniques. Real life cars would be modeled much more detailed, since a car normally not only receives a single image file, but also images for secondary ECUs which can lead up to even double-digit number of images per car. All of that is neglected due to simplicity reasons.

4.3 Chosen OTA Update Techniques

For the measurements the following techniques are used, defined by their mode, continuity and image type. They were extracted from certain IoT management or OTA update frameworks.

- **Mode**
  - **Polling-Intervals** [ASb]

    The server is called regularly at a defined rate to find out if a new update image is available (AD4). Cars are then updated automatically if a new update is available. The poll can also be used for notification only and only after the driver confirms his will to update the car, the update is performed. In our case we assume an automatic update.

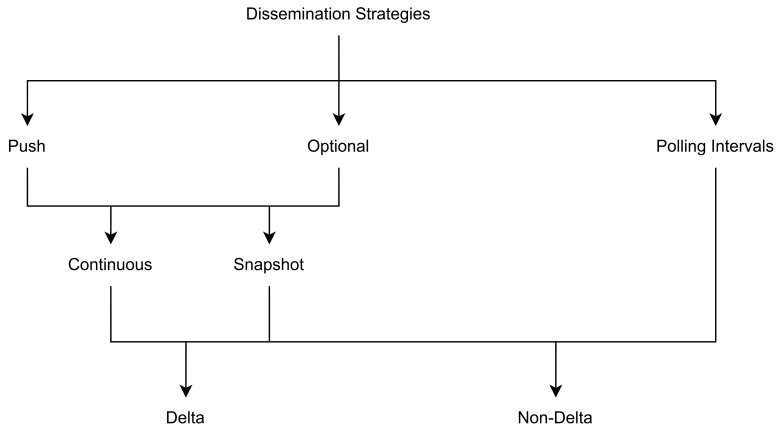

Figure 4.1: Combination of introduced dissemination techniques

  - **Push** [Srlb]

    Images are directly pushed to the cars if a new update is available and are installed without a possibility for the driver to intervene (AD3).

  - **Optional** [Srlb]

    Cars are notified about new available updates by listening to a queue (AD2). Those notifications are forwarded to the driver, who can choose if and when he wants to perform the update.

- **Continuity**
  - **Continuous** [Ama]

    Cars that are added to a group after a rollout will also receive the new image in one of the aforementioned ways (AD5).

  - **Snapshot** [Ama]

    Only cars that are in a car group at the time of starting a rollout receive the new image.

- **Image Type**
  - **Delta** [ASa; ASb; Lim]

    Images are not sent as full image update, but are created as a delta file of the previous installed image, that is only sending the changed sections in image two compared to image one (AD6).

  - **Non-Delta** [ASa; ASb; Lim]

    Images are sent as full image update.

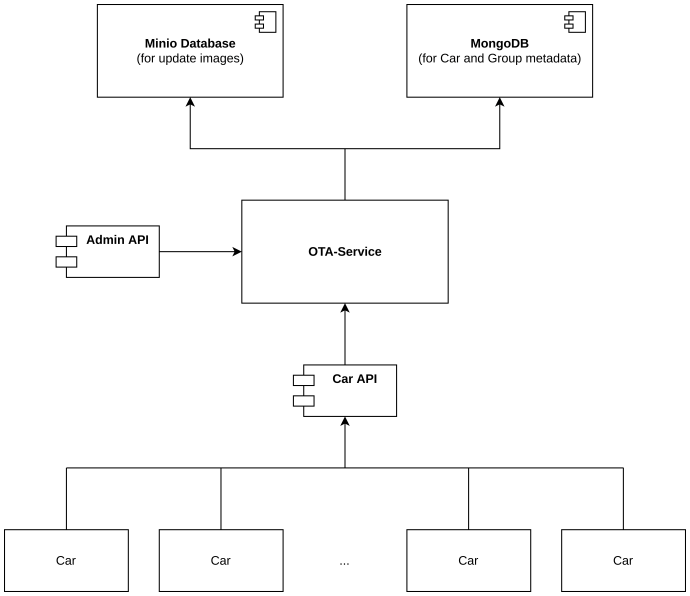

Figure 4.2: Basic architecture of OTA-tester

Those techniques can be combined. Those combinations are not all actively used in covered frameworks but can be combined logically. Push and Optional can be combined with Continuous or Snapshot approach. Polling-Intervals is excluded from that, since cars are always actively polling for a new image. If a new image gets assigned to a group every car is updated regardless if a continuous or snapshot rollout is used.

All the resulting combinations can also be combined with Delta or Non-Delta approach, since the selection of the used image type is independent of all other parameters. This can be visualized in the map shown in figure 4.1.

4.4 Used technologies for implementation

For the implementation of the web-services Spring Boot is used [VMwc] due to its easy and straightforward use. For JSON-like data MongoDB is used, [Incc] since data, like information about cars, rollouts and car groups had to be saved persistently. Since also Blob data, in our case the images, had to be saved, MinIO was chosen for that purpose [Inca]. As a broker we used RabbitMQ [Incd] which utilizes the AMQP protocol, which was needed for all techniques that require the server to send data proactively to the cars.

4.5 Details of implementation

For the implementation a simple architecture was used displayed in figure 4.2 and 4.3. Two different versions were implemented, since some techniques require a broker. If all techniques were implemented in one service, this would influence the performance of all techniques that do not

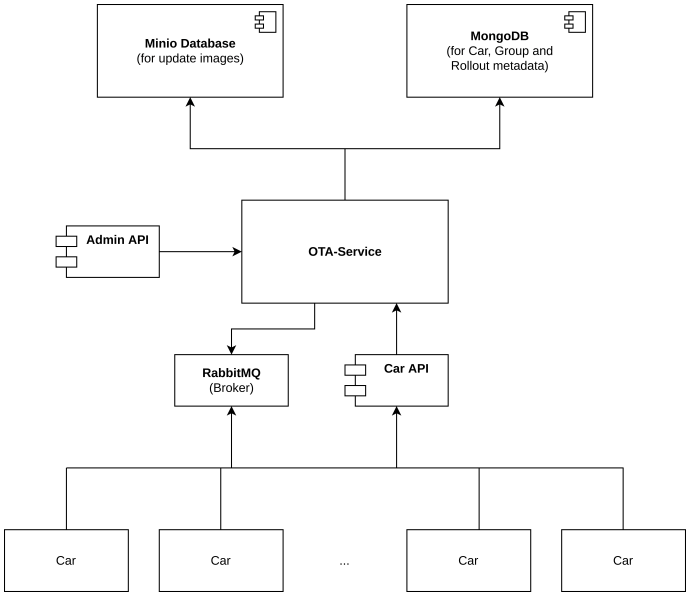

Figure 4.3: Architecture of OTA-tester with broker

require a broker to work. For instance there are differences in group creation when using a broker since there it would be necessary to create an exchange at the broker. As already explained there are two storage types:

MongoDB for basic JSON data and MinIO used for Blob data in our case binary image data. There are two interfaces in form of REST-APIs (AD11). One is the car API, used for direct requests from the cars to the server, which can be seen in table 4.1. Endpoints are provided like the registration of the car. For that the car needs to create a unique ID. Normally the server would do this, but it is easier to handle in simulations when the ID can be directly assigned in the HTTP-Request. A car must also be able to poll for the availability of a new image and retrieve that image. All of that is done via that API using the ID of the car.

To create groups and rollouts another API is needed, which will be used by a user with administrational capabilities 4.2. All of those endpoints are essentially self-explanatory. What has to be explained is how to encode a car list. This is done by a String which lists the car Ids that are added to a group, sticking to the following template: $id_1, id_2, id_3, ..., id_n$.

To define which strategy is used for the rollout, one must create a String according to the strategy tree defined in figure 4.1. For example if the admin wants to have a rollout in Optional mode, as Snapshot and Delta image delivery, he has to use the following String: “optional,snapshot,delta”. When using a Polling Intervals approach the second parameter is empty and thus replaced by “-”, for instance: “polling_intervals,-,non-delta”. Traversing that map from top to bottom helps in finding the right strategy string. All names are in lowercase.

Type	URL	Parameters	Returns
POST	/car/register/{carId}	Integer carId	-
GET	/car/imageAvailable/{carId}	Integer carId	True if new image available, else false
GET	/car/getNewImage/{carId}	Integer carId	Image that should be installed on that car

Table 4.1: REST-API used by the cars

Type	URL	Parameters	Returns
POST	/admin/createGroup/{groupId}/{carIds}	Integer groupId, String carIds	-
POST	/admin/addCarsToGroup/groupId/{carIds}	Integer groupId, String carIds	-
POST	/admin/rollout/{imageId}/{groupId}/{strategy}	String imageId, Integer groupId, String strategy	-
GET	/admin/cars	-	List of all cars registered
GET	/admin/carsInGroup/{groupId}	Integer groupId	List of all cars in that group
DELETE	/admin/deleteGroup/{groupId}	Integer groupId	-
DELETE	/admin/deleteImage/{imageId}	String imageId	-
DELETE	/admin/deleteCar/{carId}	Integer carId	-
DELETE	/admin/deleteRepos	-	-

Table 4.2: REST-API that provides management functionality for an administrational user

4.5.1 Main functionality

In the following every step needed to provide the main functionality is explained. All approaches do also involve the broker handling, which obviously is neglected in the server version without broker.

A typical work flow in that setup is the registration of a number of cars (the car fleet), the creation of a group with those cars, the upload of an image file and the start of a rollout with a specified strategy.

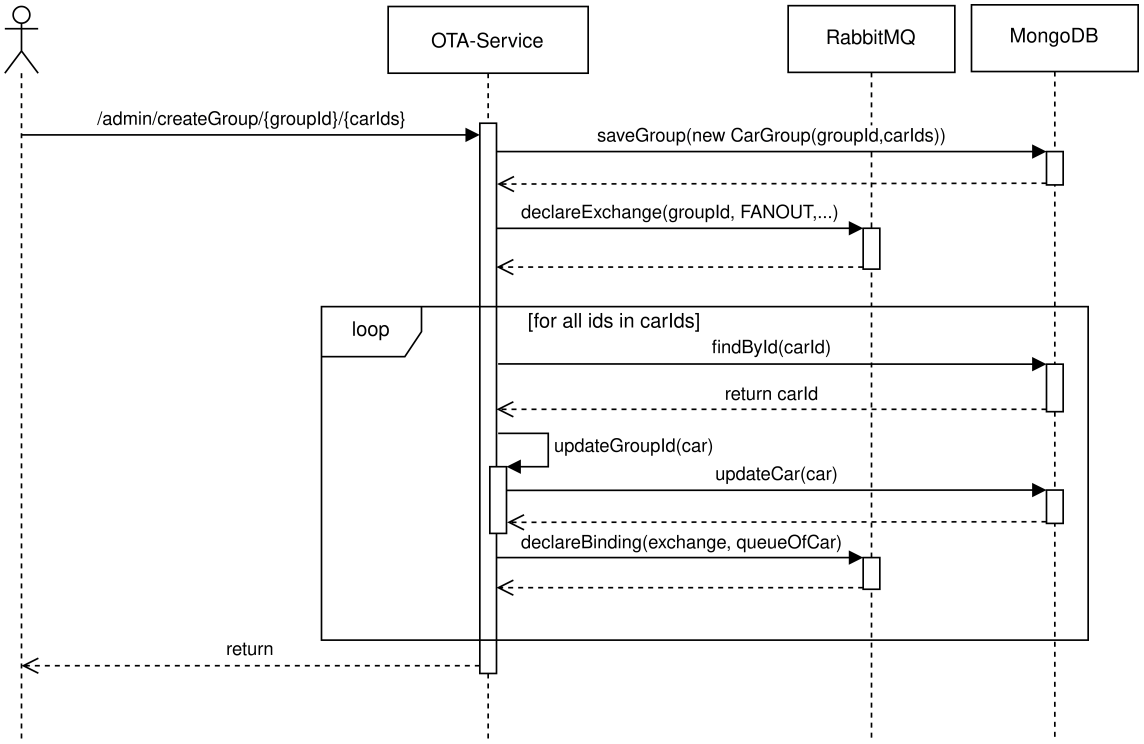

Figure 4.4: The process of group creation

When a car is registered it is stored in the database. This is done via the Spring Mongo Template [VMwb]. For simplicity reasons it is not checked if the ID is unique, but since the cars in our case are modeled by load drivers ensuring a unique ID is trivial. When registering a car, it has to be assured that the car also declares a queue at the broker. For that, we use the convention that the queue's ID matches the car's ID.

Creating a car group involves adding an entry to the group repository of the database. The next step is declaring an exchange at the broker. This is done by using the AMQP Admin [VMwa]. Here we also applied the convention that the group exchange's ID is the group ID. Fanout is used as exchange type since cars will be supplied unconditionally when a rollout is started. A Fanout exchange does exactly that: It forwards messages to every queue of the exchange no matter which routing key is used for the specific bindings or from the message itself. After that, the entry of every effected car in the database is changed by changing the corresponding group ID. Also, the car queues have to be bound to the exchange. The whole process is depicted in figure 4.4.

The next interesting step is creating a rollout. What actions are involved when creating a rollout is mainly determined by the chosen strategy.

The first strategy of which we will explain its implementation is the Optional strategy (figure 4.5). When a Snapshot rollout in Optional mode is started, at first the effected car group needs to be changed, so that the new current image is the image that is defined by the rollout. After that a notification is sent over the broker to all the cars in the group that a new image is available. In case of a continuous rollout, the whole process starts by adding a new continuous rollout to the database, since this is the only option to keep track of cars that need to be updated after adding them to a group after rollout. When running in Delta mode, a delta image needs to be created, which is done

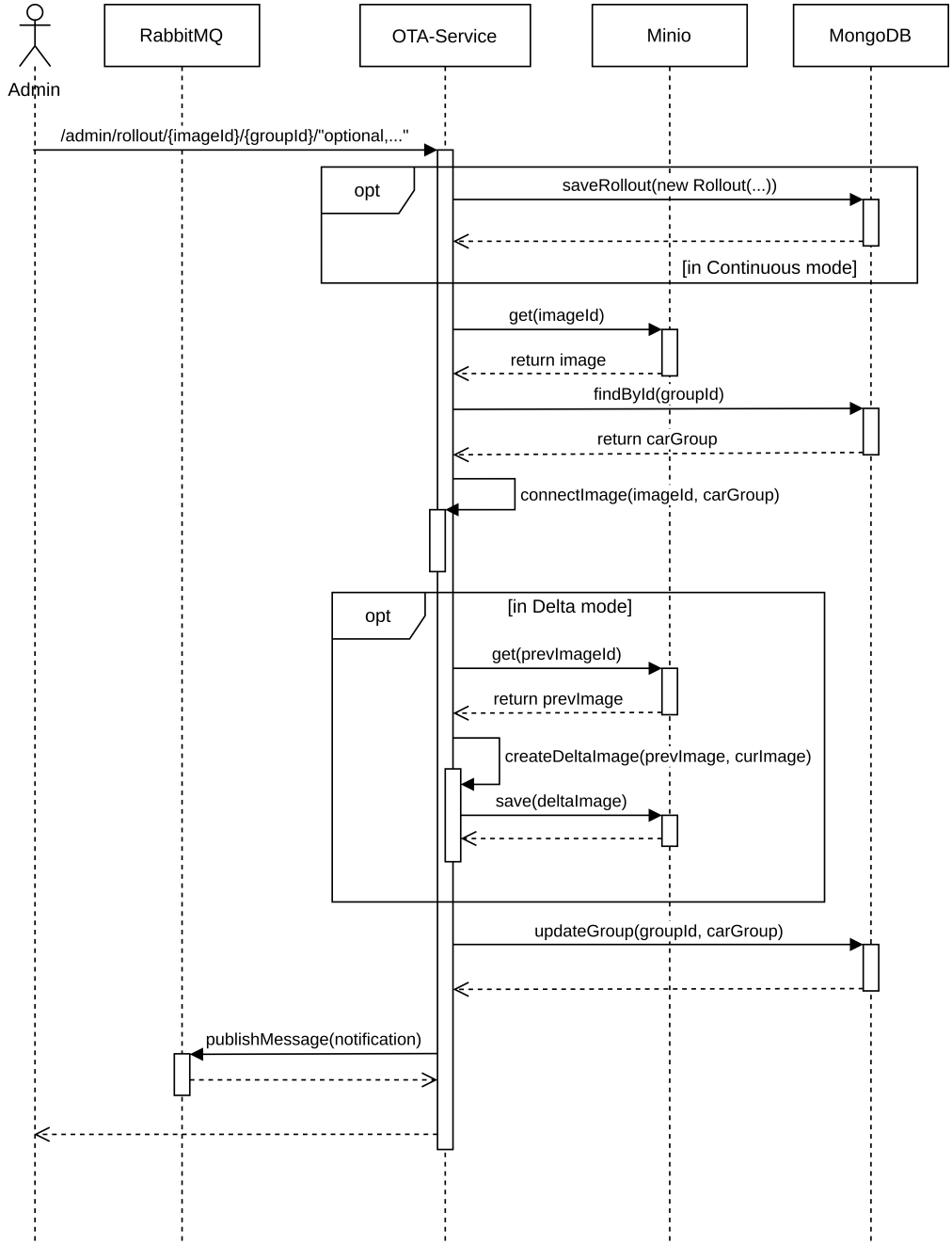

Figure 4.5: Optional sequence with all possible variations

by requesting the previous image from the database. Now in a real world setup a delta image would be created by evaluating the differences between those two binary images. As described in section 3.2.4 this is a non-trivial process, why we decided to mock this by creating an image that has a certain predefined size compared to the original images and depending on their similarity which is set via environment variables. For that it is assumed that both images have the same size. The new resulting image is created with the RandomAccessFile constructor [Ora]. We mock a perfect delta algorithm which is able to reduce the data needed to be sent by exactly the amount of similar information. Also, the time needed for that can be mocked by a waiting time, which is defined in the environment variables too. Another simplification made, is that we assume that every car has the same update state at start of the rollout. Otherwise, every car would require an own specific delta image, since not all cars had the same image installed before. However, in general the assumption is justified, especially when car fleets are more static, that is cars are not switched between groups at a regular basis. If car fleets often change their size, but the cars have not belonged to a group before our construct is still reasonable. In other cases our construction is oversimplified.

Something that is not shown in the sequence diagram is the setting of the Delta mode. When a rollout is started in Delta mode, the whole service switches into Delta mode. In reality this would need to be modeled differently, since by that, all rollouts would need to run either completely in Delta or completely in Non-Delta mode. But since we only want to compare different modes, which results in running only one rollout at the same time, this simplification is appropriate due to its reduction of administrational overhead.

In Push mode the same sequence is processed. The only difference lies in the data that is sent by the broker. Instead of only sending a notification, the whole image is published to the queues of the car group. Additionally, the entries of the cars need to be changed. In Optional mode this is done with every `getNewImage` call.

When executed in Polling Intervals mode the server behaves as explained in figure 4.6. The update image is pulled from MinIO as well as the group from MongoDB. The group entry is updated such that the new image is now the designated image of the group. In Delta mode a delta image is created and stored in the database. When a car now polls for a new image by calling `newImageAvailable`, it receives `true` if the car's current image does not match the designated image of the group (figure 4.7). Then the car automatically calls `getNewImage` and receives the new image. In Delta mode the image ID found in the car group is not directly used, but the delta image is found in the database by querying the image ID in the following way: "PrevImageID_CurrentImageID.delta". Under that ID the delta image was saved before in the rollout call.

When cars are added to a group after creation, the actions involved differ when a continuous rollout is connected to that group, compared to if not. If no continuous rollout is connected to the group, the group is pulled from the database and the car IDs are saved in the group instance. Also, every car gets its corresponding group ID value updated. The dedicated sequence diagram can be found in figure 4.8. When a continuous rollout is found, the following actions are performed (figure 4.9) after the aforementioned ones. For the newly added cars a new exchange will be created, which is used to supply the cars with the already installed image of the rest of the group. Here also a fanout exchange is used. Depending on the chosen rollout mode the newly added cars either receive a notification that an image (or delta image) is available or the image (or delta image) is directly send to the cars via the broker. After that the car entries in the database get updated. Here this is done unconditionally, in a real world setup one would need to assure that every car has received the image by some confirmation that is sent by the car. The distinction between Delta and Non-Delta

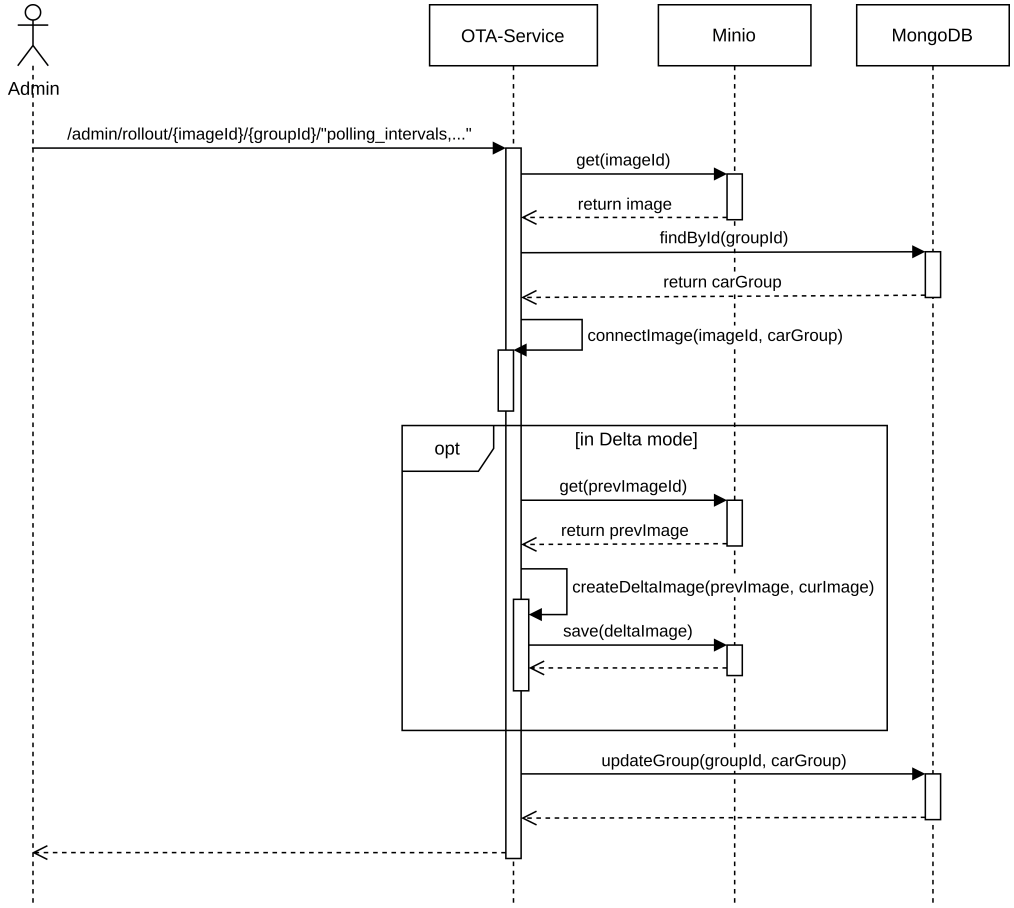

Figure 4.6: Polling Intervals sequence

mode is not shown in the diagram in figure 4.9 for better readability, since the differences are not significant. Instead of a normal image the delta image would be pulled from the database, with the aforementioned ID template. Also, the notification sent out by the broker is slightly different, than in Non-Delta mode. After supplying the newly added cars with the already deployed image the alternative exchange is deleted.

With those described endpoints different OTA update strategies can be tested on their performance under different conditions.

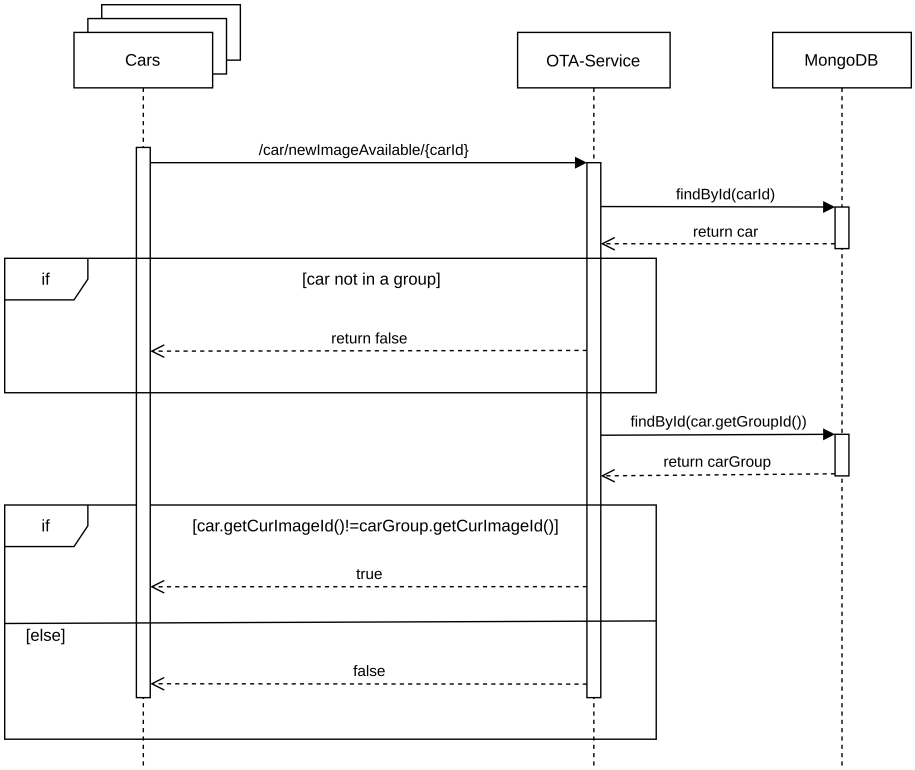

Figure 4.7: Actions involved in a `newImageAvailable` call

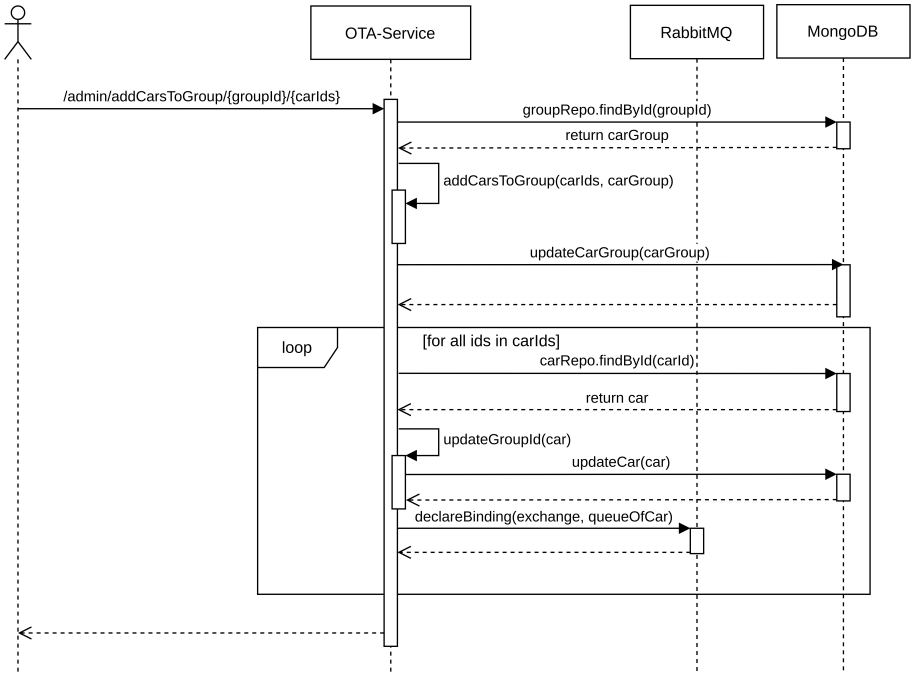

Figure 4.8: Sequence diagram of `addCarsToGroup` in case the group has no continuous rollout running

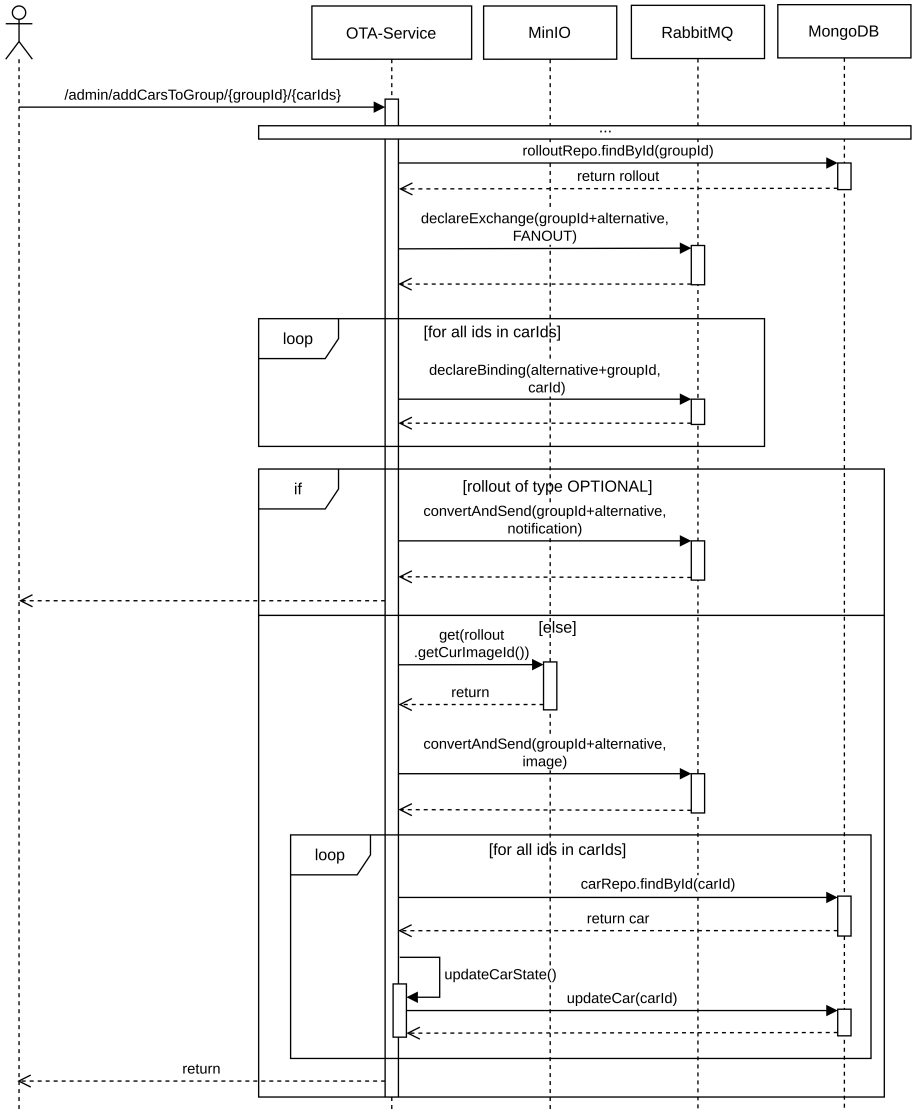

Figure 4.9: Continuation of the sequence diagram of `addCarsToGroup` if a continuous rollout is found

5 Measurements

In this part of the thesis we explain the process of measurement and the used metrics.

5.1 Simulation of Clients

Clients are simulated by Gatling [22b]. Gatling is an open-source load testing tool that is able to simulate a predefined demand at web APIs. It automatically visualizes test results and reports which makes testing much simpler.

To compare different techniques, different scenarios where chosen. At first there are all the basic scenarios that simply describe registration of a car fleet, group creation and starting of a rollout as shown in figure 5.1. In case of a delta update a second rollout is started after a predefined time, but now the image files are sent as delta (see figure 5.2). Another case in which the base scenario deviates is the continuous rollout strategy. For that, cars are added after rollout start (figure 5.3). To compare the performance of continuous against snapshot version another scenario is used in which

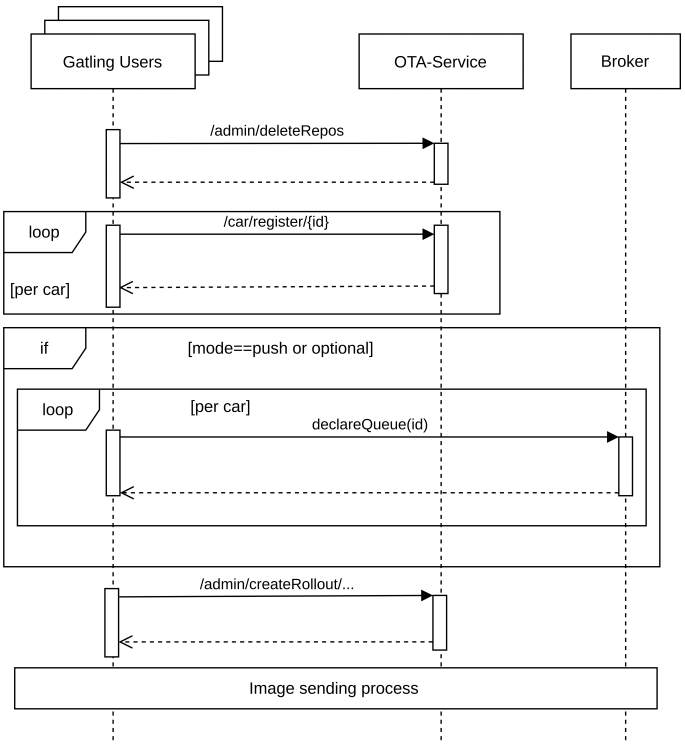

Figure 5.1: Basic scenario setup of Gatling users

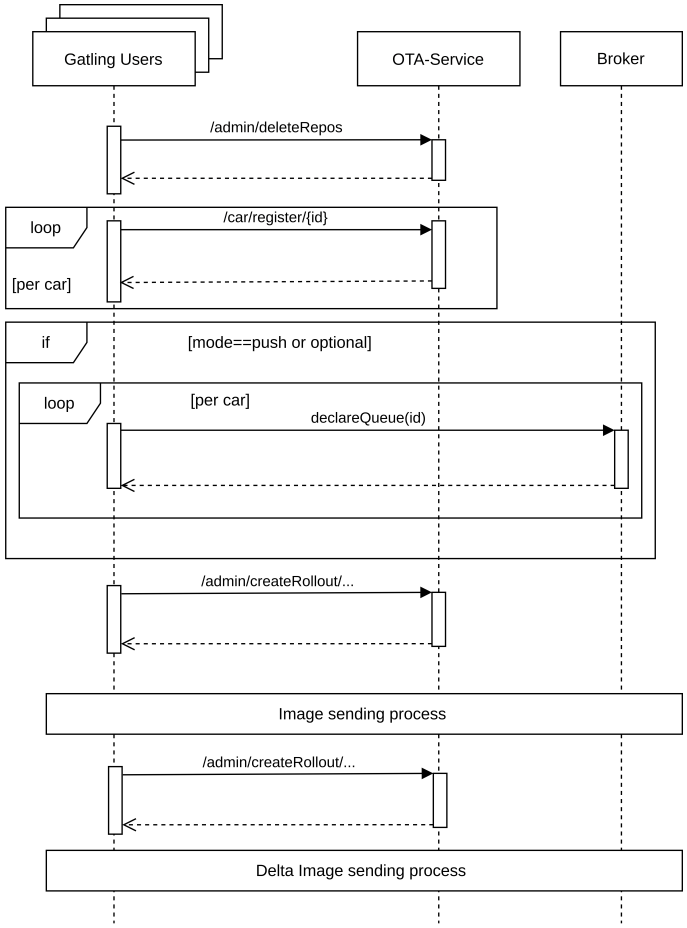

Figure 5.2: Basic scenario setup of Gatling users in delta mode

later added cars are served by creating a new rollout for the whole car group (figure 5.4). In all scenarios response times are measured and also the completion time of the rollout which can be determined either by the help of diagrams created by Gatling that describe the temporal distribution of made calls, or by the time needed for the scenario itself. In case of scenarios that need a broker to work, the latter can be applied.

The image sending process is different depending on the chosen strategy, as explained in section 4.5. This includes either polling for a new image regularly and pulling the new image if available in case of polling intervals as strategy chosen (figure 5.6), listening to the broker until a notification is received, to then pull the image via the REST-API in case of optional (figure 5.7), or in case of push, receive the image directly from the broker (figure 5.8).

To determine the elasticity of the used technologies, scenarios were created that do not represent a logical state in real life but strive to create specific load patterns on certain components. For instance, to create a specific number of requests per second it is more difficult to represent this in a real life scenario. To realize this, cars would need to register and receive the information that a new image is available and subsequently receive that image by calling the REST-API. To shape specific traffic patterns cars would need to be deleted from a group, new cars added to the group and a new rollout would need to be started. This can not be run in one joined scenario since Gatling needs to

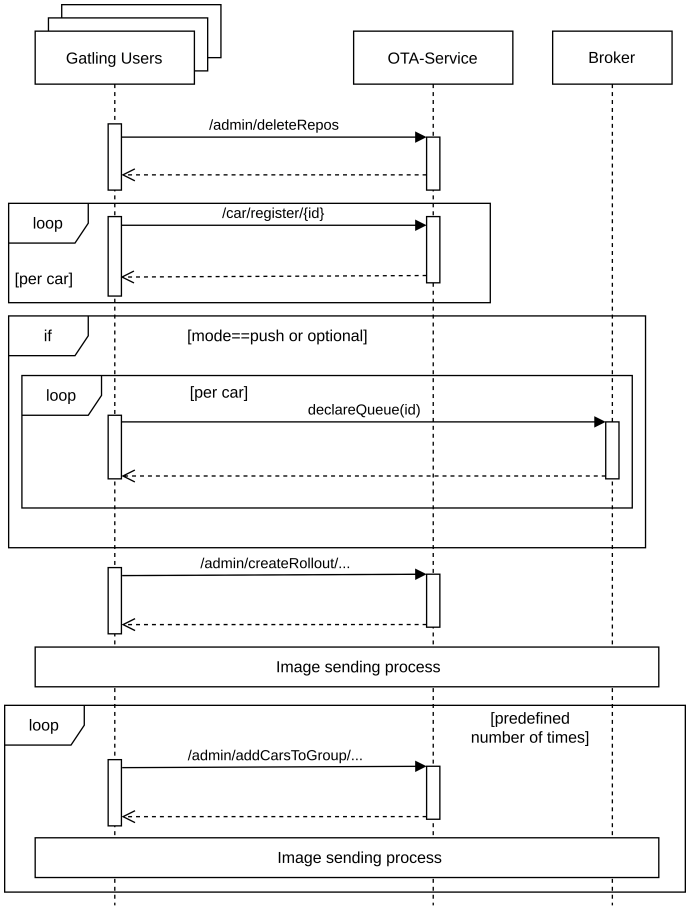

Figure 5.3: Basic scenario setup of Gatling users in continuous mode

simulate different numbers of users for the particular sub routines. Many users are needed for the registration, one for creation of group and rollout, or the deletion of entities and many users again for the process of image reception. All of this makes matters a lot more complicated, so we decided to simplify the scenario. In order to get rid of this administrational overhead the REST-API is frequently called with the same car ID which leads to an easier controllable traffic. Rollout creation and the management of cars with different IDs is skipped to only focus on the impact of a lot of cars pulling a new image. By that only one car has to be registered and added to a group and the rollout has to be started once (see figure 5.5). When executing those simulations, elasticity can be observed by using an auto-scaling policy and another run of the simulation with only one replica of the OTA-service to see differences in response times and throughput.

To sum up the goals of the scenarios:

Polling intervals strategy is compared with the other strategies involving a broker. This is done by measuring the response time when calling the REST-API or receiving the image directly from the broker. The completion time of an update with a certain strategy will be observed by simulation time, or time of the last `getNewImage` call. Delta and non delta approaches are also compared by

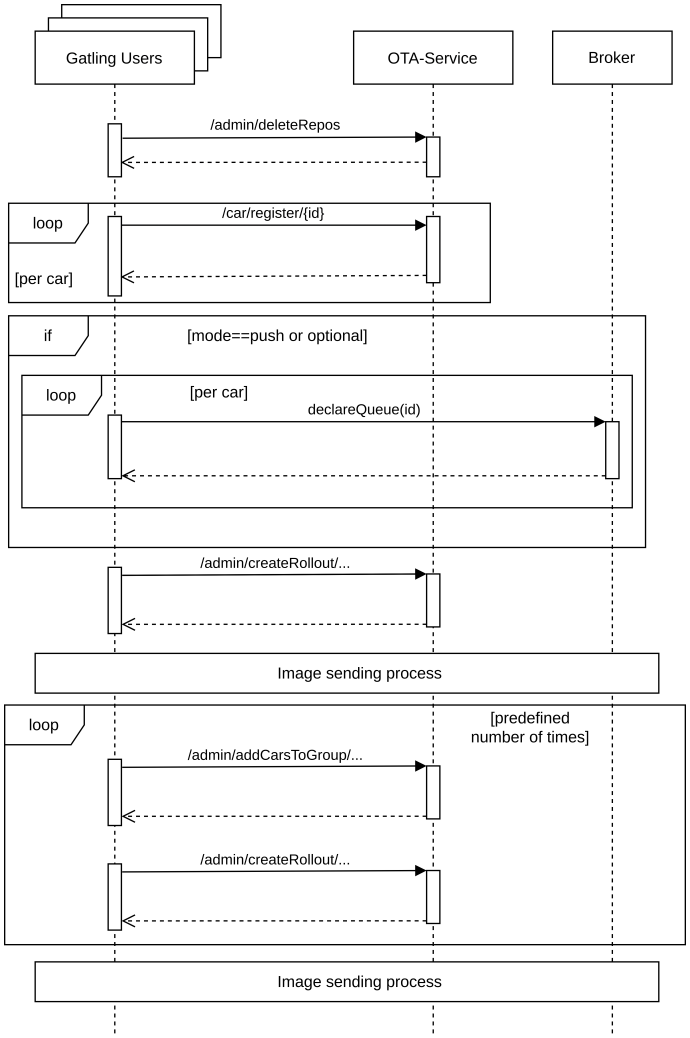

Figure 5.4: Basic scenario setup of Gatling users using new rollouts in comparison to continuous mode. Here for every bunch of cars that is added later a new rollout is created for the whole group.

completion time and mean response time of involved requests. Continuous and snapshot approaches are compared with alternative scenarios that start a new rollout to serve cars that are added later to the group. Here the completion time is the metric which determines the effectiveness.

Testing approaches that involved a broker, required a plugin for Gatling. We used the AMQP plugin [Tin23] to provide a way to listen to queues. Normally, this plugin is used to create load on a broker by inducing a certain number of messages on an exchange and then listening to the respective queues. We misused this construct in a way that it is only listening to queues that are connected to the exchange that is used by our update service. The messages that are originally meant to create load are published to a dead exchange, where no queues are connected to.

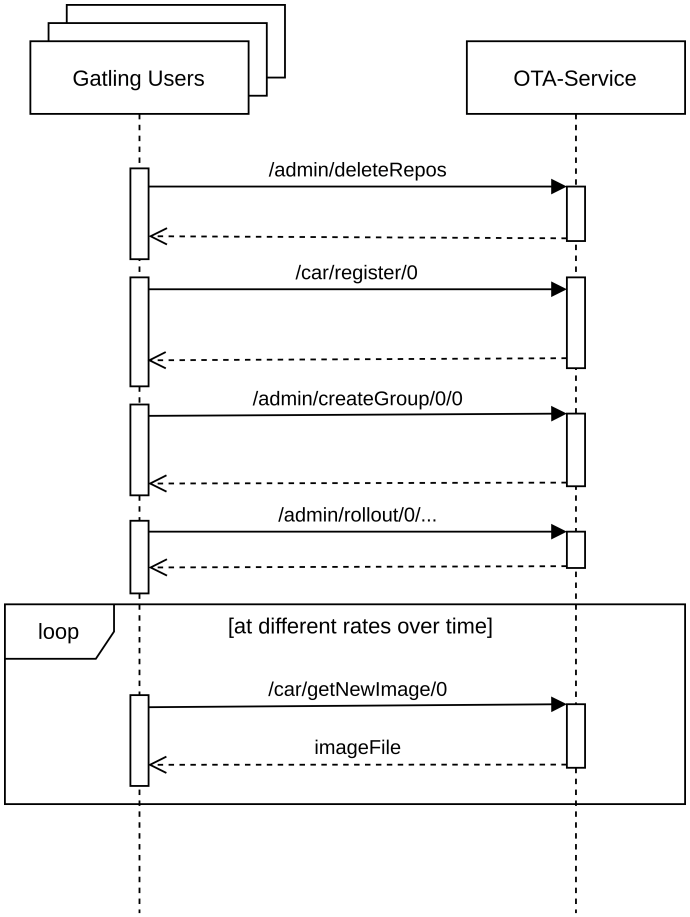

Figure 5.5: Simple elasticity job calling for a new update image

5.2 Basic Simulation Setup

The spring boot applications of the two service versions were containerized with Docker [22a]. The services were deployed on a Kubernetes cluster. Most tests were run with a static number of replicas per service, also for the used databases and the broker. By that the behavior could be shown under certain predefined resource limits which was one core per pod. At first, we ran the aforementioned basic test scenarios in sequence and with a base load of 200 cars and an image size of 4 MB. These initial values where chosen without foundation from literature. We did not find any work that covers typical car fleet sizes in the realm of OTA updates. The only reference to be named here is Aurora Labs [Lab]. They compared different OTA update approaches on their costs. For their cost calculator they have default values of 500 MB as a large image size and 0.42 MB as a small image size. The default fleet size is 10 million cars. Since these numbers are difficult to replicate in our small setup we decided to use a fleet size of 200 cars and an image size of 4 MB to begin with, which can be compared to a company's car fleet. After seeing certain trends we went further into detail with those trends. We reran the tests that showed patterns under other conditions like increased fleet size, increased image size, modified polling interval or changed delta similarity (the similarity of the previous installed against the currently installed image).

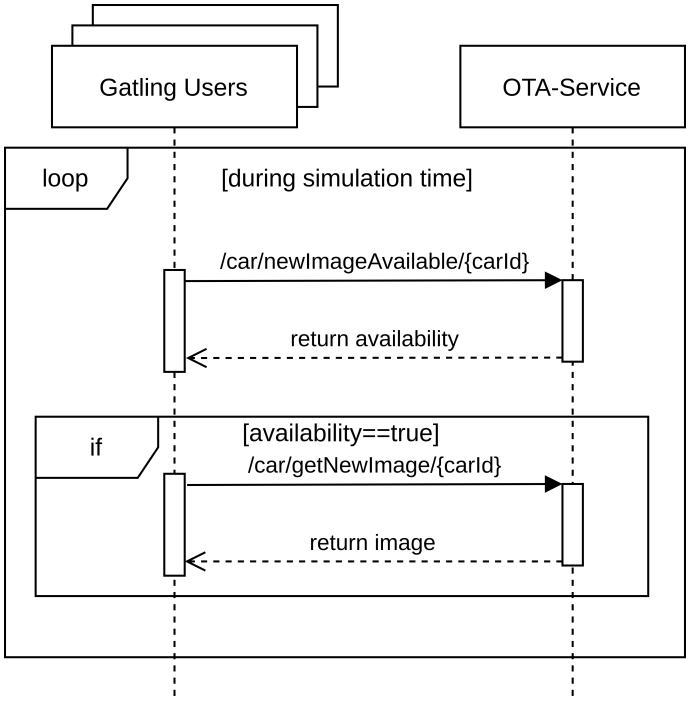

Figure 5.6: Image sending process when using polling intervals mode

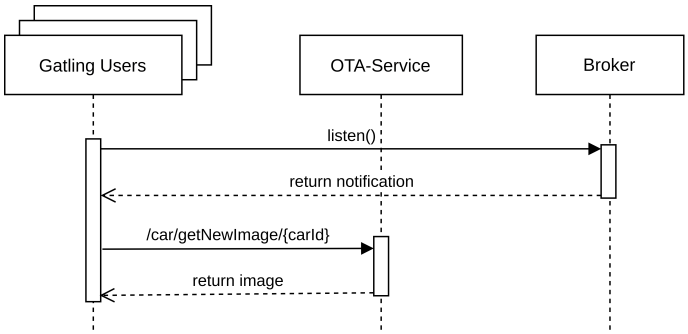

Figure 5.7: Image sending process when using optional mode

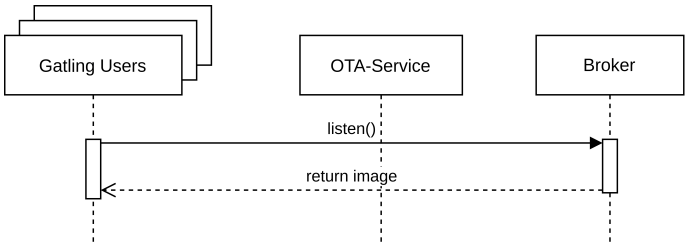

Figure 5.8: Image sending process when using push mode

The Gatling tests are run as Jobs with the given parameters. They were run on an extra node due to their large CPU consumption. The rest of the components were distributed over three other nodes. When testing one construct (e.g. without a broker) we set the number of replicas of the other (e.g. server with broker) to zero. This was done to free as many resources as possible. To observe certain metrics like CPU, memory and the network, we used Lens [Incb], which is an IDE for the work with a Kubernetes Cluster. Moreover, Lens helped in speeding up the workflow.

From every setup we used a sample size of 5. When using response time the mean is already created from all requests that are sent out in each scenario. We either put those means into box plots or calculated the overall to reduce visual when creating a graph that shows a certain a trend more easily with data points than intervals.

6 Evaluation

In the following we will recap the used study designs, which results we obtained, how we interpret them and which threats to validity there are.

6.1 Study Design

As already described in chapter 4 we emulated certain OTA dissemination strategies. These strategies were chosen from frameworks that we have found in the first part of the thesis. We used concepts like device groups or rollouts from such frameworks and simplified them to a certain extent. We did not use the frameworks directly for the measurements, since they were difficult to compare due to their large IoT management functionality which would have lead to data noise, since comparing different strategies would also mean to compare the different frameworks in general, which makes it hard to interpret test results. That is why we used our own test prototype which strove to achieve the goal of comparability when executing certain OTA update strategies. Docs of the mentioned frameworks only mention on a conceptional base how the strategies work without giving closer details of implementation. The update strategies we used are only general remarks how cars get informed about a new update and how they subsequently receive their respective image. To measure the QoS we compared metrics like response time and completion time. While response time represents the quality or performance that the car driver observes, completion time is a metric with much higher importance to a fleet administrator. Completion time is most important in scenarios where non-optional, or security critical software is supplied, to measure the time in which a fleet is exposed to a higher risk. Other metrics that we observed were the network usage on the cluster as well as CPU and memory utilization. This was necessary to observe bottlenecks in the strategies. Combining all of that, we tried to find patterns that can be observed when applying a certain load on our prototype. After finding patterns we focused more on exploring them further by changing load parameters like fleet size, image size or delta similarity. All of this is already described in more detail in section 5.2.

6.2 Results

6.2.1 QoS Measurements

In the following we report the observed test results grouped by their implications.

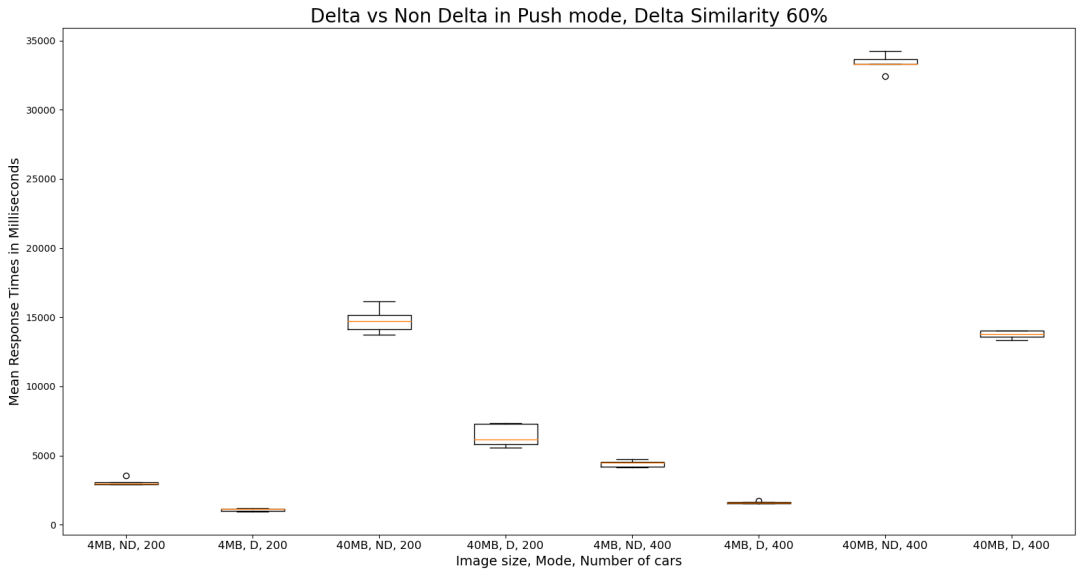

Figure 6.1: Graphic that shows the relation between Delta and Non-Delta mode. Especially with high image load we see a large reduction in absolute response time. Sample size: 5 simulations per box blot with hundreds of requests per simulation

Delta vs Non-Delta mode The first pattern we observed was the fact, that in Delta mode the response time as well as the completion time is reduced, depending on the similarity between old and new image (figure 6.1). When assuming a higher similarity the response time is reduced accordingly. In general, it was shown that the reduction in response time is linear dependent from the similarity of the two images. When having higher image payload it is reduced even further. We showed this specifically for the different strategies but also in general by comparing the normalized reduction of response time in all strategies. The reduction in completion time depends also on other factors. The polling interval size is important since it controls the overall load on the service or how fast the drivers decide to update their cars in Optional mode. So the completion time is not as dependent from the delta mode as the mean response time is. However, it is still largely reduced when using delta images.

Continuous vs Snapshot Comparing continuous against snapshot approaches we observed that with a continuous rollout a car fleet that increases its size can be updated faster, that is the new arriving cars are automatically brought to the same update state of the rest of the fleet. To achieve the same in snapshot approaches the whole car fleet has to be updated again, which leads to many more send operations compared to continuous approaches. This was shown in Optional and Push mode since in Polling Intervals there is no continuous or snapshot approach. In both cases the difference is especially high if using a larger image size (figure 6.2).

Main Strategy comparison It was observed that in general Push is the fastest approach, when using larger image sizes (figure 6.5). When having smaller images we see that Polling Intervals has the best response time (figure 6.3). This response time can be decreased even more when using a

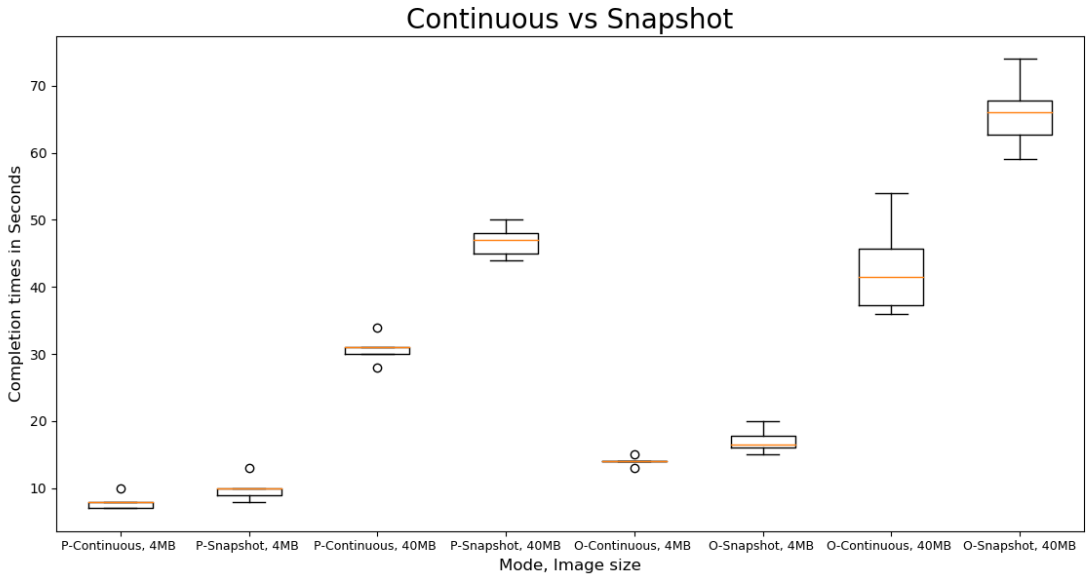

Figure 6.2: Comparing continuous against snapshot approaches. Using 200 cars that were added in 5 bunches the size of 40. Sample size 5 per box blot. P=Push, O=Optional

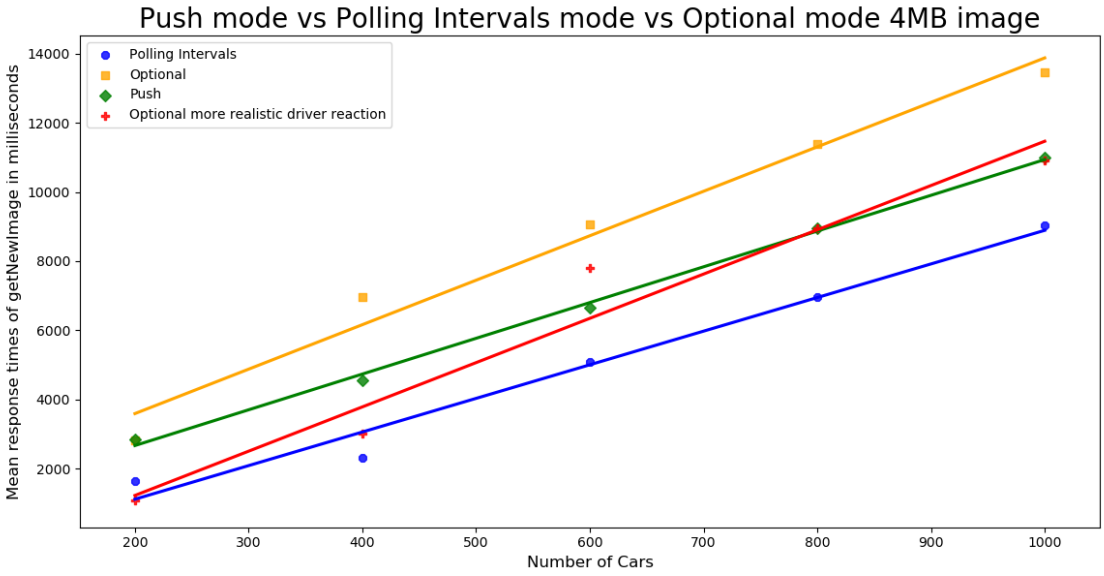

Figure 6.3: Different dissemination modes compared with different fleet sizes and 4 MB image, used metric is mean response time. Sample size 5 per data point (every of those 5 is also a mean of the response time of every car).

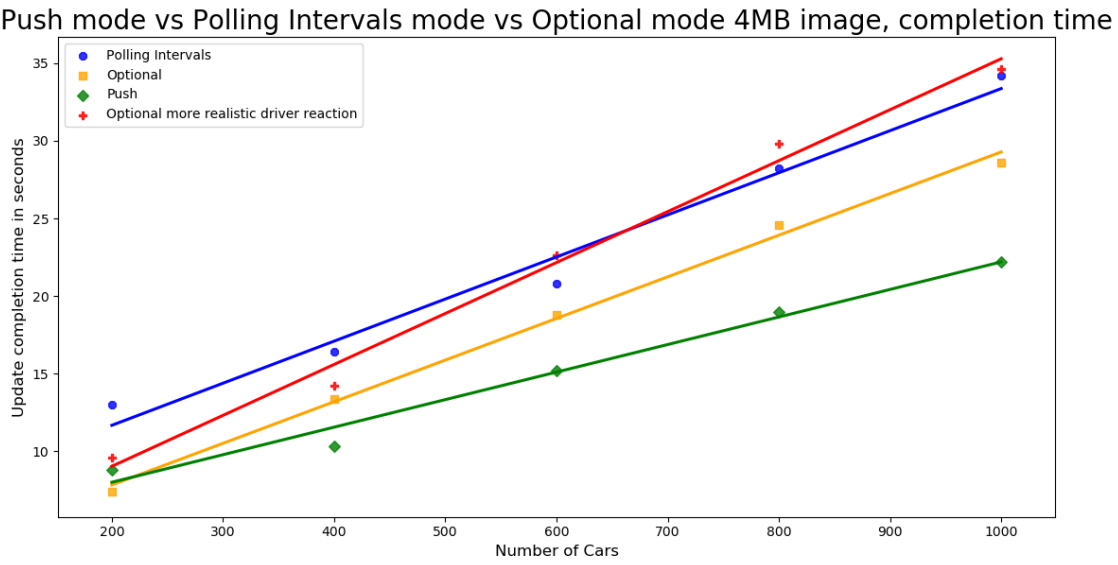

Figure 6.4: Different dissemination modes compared with different fleet sizes and 4 MB image, used metric is mean completion time. Sample size 5 per data point (every of those 5 is also a mean of the response time of every car).

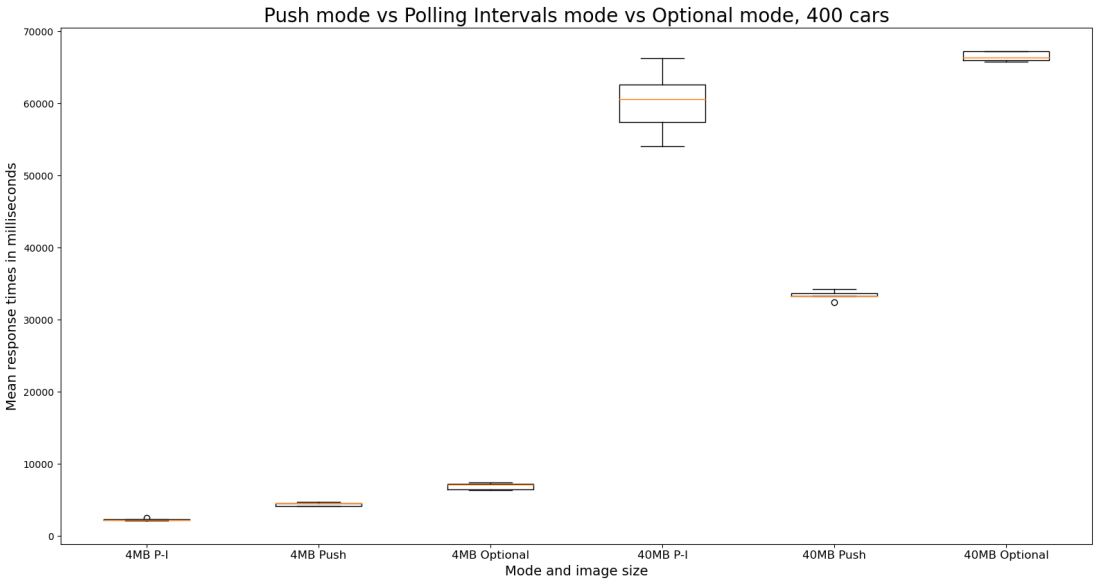

Figure 6.5: Different dissemination modes compared with fleet size 400 and different image sizes, used metric is mean completion time.

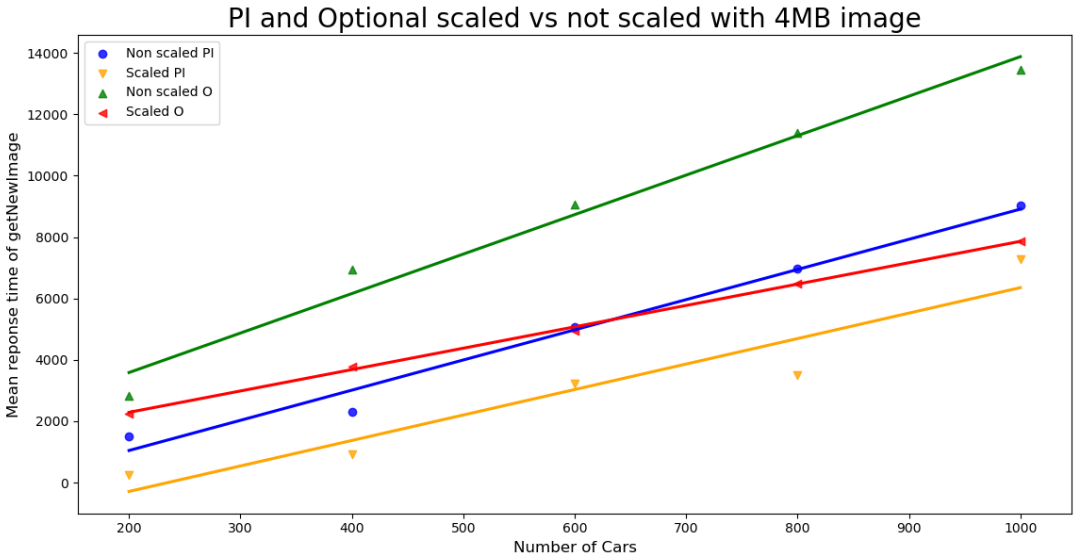

Figure 6.6: Different dissemination modes compared with different fleet sizes and 4 MB image, used metric is response time. Sample size 5 per data point (every of those 5 is also a mean of the response time of every car).

higher polling interval (with that a lower polling frequency). Optional is the slowest when assuming that every car driver immediately decides to pull the image if available. If we assume a certain random waiting time we see that the response time decreases.

In contrast to that, the completion time of the rollout increases when choosing a larger polling interval, as well as when we assume that the drivers do not immediately pull the new available image (figure 6.4). When observing completion time we see that Push is the fastest way to complete the update especially when we observe larger image sizes.

Scaling Behavior When scaling up the deployment of the main service, we observe a reduction in response, as well as in completion time. Depending, if we use the Optional mode the difference is much more noticeable than if we use Polling Intervals which can be seen in figure 6.6. Push mode is not at all effected by scaling of the main service. This is simply due to the fact that the broker carries the main load, and we did not scale the broker.

6.3 Discussion

In this section we will discuss the implications of the test results.

Observed bottlenecks The main observed bottleneck is the image database. In Optional as well as in Polling Intervals we query the image database every time a car has a new update available, which leads to huge amounts of data that needs to be sent from the database to the service and then to the cars. When having high image loads and only one replica of the MinIO deployment, it

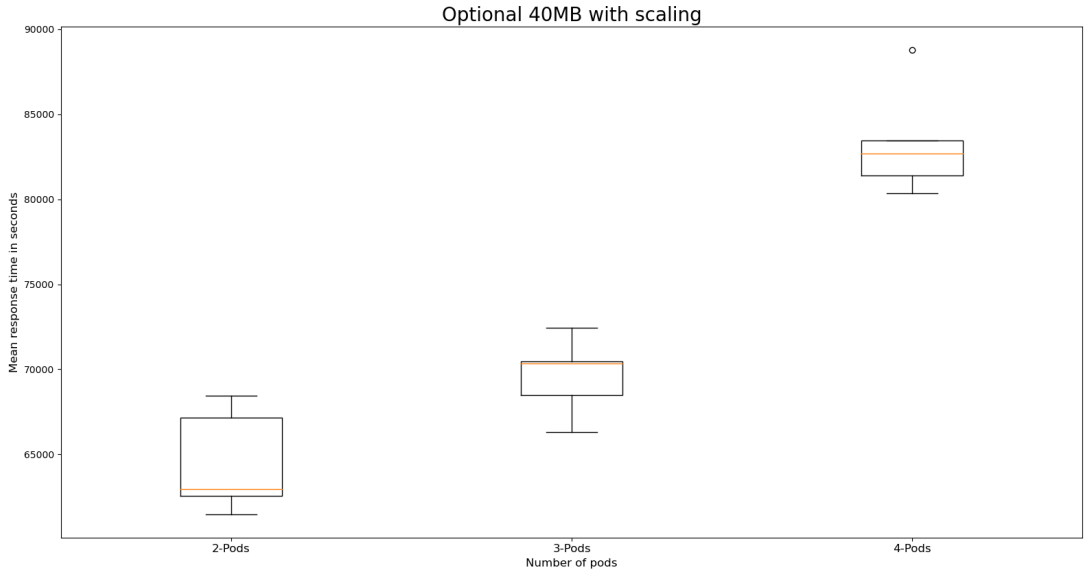

Figure 6.7: Scaling with larger image sizes in Optional mode to demonstrate the upcoming network bottleneck.

started to drop requests which lead to errors. When increasing the number of replicas it was able to cope with that load again. After increasing the number of replicas of the image database even further we expected to have a lower response time. In reality, it increased (figure 6.7). This can be explained by a network bottleneck. When increasing the image size and the car fleet size we observed, that the maximum data rate sent between the database and the service is 190 MB/s. With that the observation makes sense, since more requests are processed at the same time, since more resources are available. But since the maximum transmission rate stayed constant, every car has to wait longer for its expected response.

Delta vs Non-Delta mode The observed reduction in figure 6.1 makes sense, since the amount of data to be sent is reduced, determined by the delta similarity. However, delta updates are not always applicable in this straight forward implementation that we chose. Cars need to have the same update state to enable the creation of only one delta image for the whole car fleet. Otherwise, the delta image creation would need to be repeated for every car. If a high delta creation time is assumed it becomes clear that this would be at a certain point so ineffective, that it is more effective to start a full image (non-delta) update of the car fleet, because the time needed for delta creation would vastly exceed the time needed for transmission. One option to realistically enable such a scenario is to bring every car on the same update state when being added to the group with a continuous rollout and then perform delta updates to achieve a large reduction in completion and response time.

Continuous vs Snapshot That continuous approaches are faster in updating newly added cars makes sense, since only those cars get automatically updated. For the same to be achieved with snapshot rollouts, the whole car group would need to receive a new rollout. Even if images that are already installed on the respective cars are not sent out in this process it means still more effort for the administrator to have the need of starting a new rollout every time he adds cars to the group. So

especially in scenarios where cars are added more often to a car fleet the continuous approach is very useful. The overall usability for the administrator is increased in such cases. Also, it can be used to enforce a certain invariant that every car should fulfill when being added to the group (especially in combination with push, since then the driver cannot intervene). To give one instance for such a scenario: If cars are added to a police fleet, it is desirable to have software that constantly monitors their position to coordinate patrols more easily and to ensure more security for law enforcement officials.

Snapshot on the other hand is useful for more optional software, like a new feature for the car's infotainment system where the driver should decide if he wants to install it or not. Snapshot has the advantage of not overwhelming the car drivers with notifications about optional updates when the car is added to a fleet.

Main Strategy comparison The stability of the Push mode is explained by the fact that the image with which the fleet is supplied is only pulled once from the database and then distributed by the broker. Since the main bottleneck is the image database, we also see that this effect is much more visible with large image sizes.

With small image sizes the Polling Intervals strategy works best, since cars poll uniformly distributed over a certain predefined Polling Interval. With that the highest load (the pull of the image) is also distributed over a larger time frame. This reduces the overall load on the service and reduces the response time compared to the Optional test sample, where the cars immediately pull for a new update. To see that there is an actual difference we also created a test sample where the drivers not immediately react to the notification, but wait a random time between 0 and 5 seconds (which is also the Polling Interval in that test series). Even though we expected to then have roughly the same response time compared to the Polling Intervals mode, we observed a higher slope in the response time approximation (figures 6.4, 6.3). An explanation for that is, that the notifications, which arrive over the broker do not arrive exactly at the same time at the cars, and are also not uniformly distributed in their arrival time, which leads to higher peaks in load for the service than in Polling Intervals mode.

Regarding completion time we see that Push is outright the fastest here as can be seen in figure 6.4. This is explained by the fact that in Polling Intervals all cars are only notified about all updates after at least one Polling Interval. In Optional, if assuming that the drivers immediately decide to pull the image, the completion is lower than if they wait a certain time to perform the update. One can argue that in Optional mode the completion time is not a very important metric if it is only used for optional software components. That every car is supplied with a certain image is only crucial if the update is security critical or fixes a serious issue in the car's software. But in this case a different approach should be chosen: Either Polling Intervals, in the form that we used, such that the driver can not choose to decline the update, or the Push approach, which immediately sends out the image to all cars. In general, it can be said, that in Optional completion time and response time heavily depends on the behavior of the car drivers. How they react to an update notification, how many try to pull at the same time determines the load on the service and with that the mentioned metrics. In Edgehog [Srlb] it is mentioned that the image does not need to be sent immediately if a driver decides to pull the image after a notification. This is used as a backpressure mechanism. Out of simplicity reasons we decided not to implement this, but this also shows that the completion time is not a very important metric when it comes to optional updates.

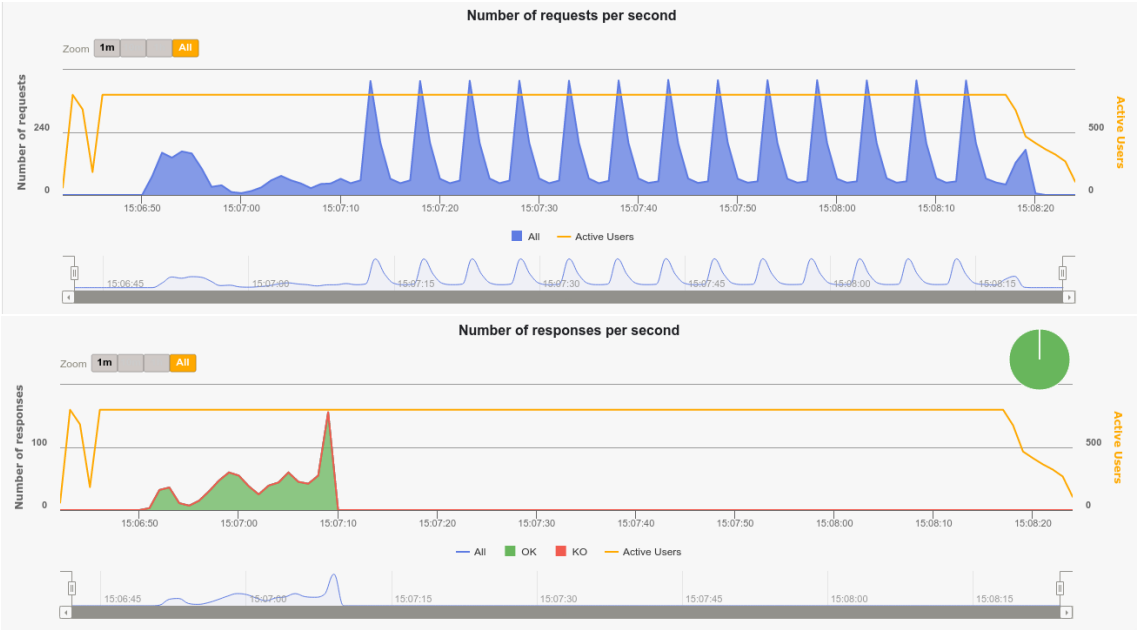

Figure 6.8: In both graphs it is clearly visible that even tough the `newImageAvailable` calls are started over a whole interval, the image pulls are mostly finished at the same time. The upper graph shows the `newImageAvailable` calls per second, the lower shows the responses of `getNewImage` per second.

In Polling Intervals mode there is a direct way to control the load that is applied to the service. If the Polling Interval is higher, the load is decreased due to a lower polling frequency and vice versa. Also, if more cars are added to the fleet, the load increases due to more cars that poll at the same time. To use the resources as effective as possible it makes sense to use a priority for different cars or update types. Polling for more security critical updates in a higher rate helps to decrease the completion time of such updates for the whole fleet. The time were some cars are still driving with outdated software is much lower and with that the risk of having serious issues is reduced. Moreover, different car types in a fleet could be configured with different Polling Intervals, to prioritize certain cars over others in the fleet. Another pattern that was observed in Polling Intervals mode was the vast difference between randomizing the start of the cars' polling. If all cars start polling at the same time, peaks are created in which the load is much higher than during times in which no car polls. Especially if a new update is available, all cars start pulling the image at the same time, which leads to a higher response time for the car drivers. But even if the polling start is randomized we still observed, that requests synchronized after one image pull (figure 6.8). Even though the cars start polling over an interval and not all at the same time, most cars finish at the same time. Which subsequently leads to synchronized `newImageAvailable` calls and thus to higher load at a time. To fight this, one approach would be to randomize also the first poll after the image pull. This would distribute `newImageAvailable` calls more uniformly again and with that also the `getNewImage` calls if a new image is available.

Scaling Behavior As already mentioned, we observe a reduction in response times for the `getNewImage` call if we scale up the deployment of the main service. The higher reduction in Optional can be explained by the higher load, which is induced, since all cars try to pull the new image immediately after getting the notification. In Polling Intervals the overall load at the same time is much lower, since cars pull uniformly distributed over one polling interval.

6.4 Threats to Validity

The limited number of samples used for update completion time can be a problem due to outliers. The data set we used for mean response time is much larger with hundreds of requests per simulation and five simulations. So the validity of our mean response time samples is higher than our completion time samples. One approach would be to increase the number of overall simulations per scenario.

When taking into account the whole intervals of response and completion times in our results for main strategy comparison and the scaling behavior, it can be seen that they overlap to a certain extent. The trends are still clearly visible, but the differences in performance between certain strategies are not as significant when taking the intervals into account.

The performance difference between Push and the other approaches is likely higher than in a real world setup. When taking caching into account, the number of requests to the image database would be reduced, overcoming the network bottleneck between main service and database. Since Push only needs to pull the image once from the database, the effect of caching can reach to a similar performance than with the Push approach.

7 Conclusion

7.1 Summary

We have grouped certain OTA update techniques into four categories and tried to find out the main trade-offs that are involved in those groups. We observed that in update management there is a trade-off in using static fleets defined by ID, or dynamic fleets defined by constraints. While the first is more effective in managing fewer fleets, where cars do not change their fleet affiliation much often, dynamic fleets enable an easier management of multiple groups, as well as the transitions between those groups, since only certain variables registered for the vehicle need to be changed, such that it changes its fleet affiliation automatically. Also, updates can be secured, by automatic checks, if the update image fits the constraints a fleet is described with. That ensures compatibility.

Regarding Update Security we have seen that there is a trade-off between higher security and memory / resource consumption. In some ECUs it may not be possible to apply certain security approaches. Car manufacturers need to keep that in mind to protect the most crucial parts of their hardware with the proper security mechanisms. Moreover, we concluded that not only the connection between server and cars needs to be protected, but also the whole system by providing compromise resilience with an approach that should be at least equivalent to Uptane.

In Update Dissemination we have found trade-offs between load balancing and update completion times. While load balancing in any form (e.g. a larger polling interval, or only a certain amount of notifications per minute) lead to a higher completion time, the mean response times reduce, and thus the overall quality observed by the car drivers is higher. Depending on the use case it can make sense to reduce the quality observed by the clients, by updating more cars at the same time. The overall time that cars are in an insecure out-of-date state is reduced. When applying updates, where car drivers observe the update process more directly (e.g. infotainment updates) a strategy that preserves the quality observed by the car drivers should be chosen. When a safety critical update is available, an approach like push should be chosen, to prevent a car driver from rejecting the update. If cars automatically pull the image after a new image is available, also the polling intervals as well as the approach where a notification arrives and cars then pull, can be used for that purpose. It is important to prevent car drivers from rejecting such an update. Also, high load needs to be handled. We have seen the impact of high load on our prototype and the importance of load balancing when having larger image and fleet sizes. We have also shown the effectivity of delta updates, when a car fleet has only one consistent update state, such that one delta image per fleet is sufficient. Also, different protocols provide certain trade-offs. Mostly the trade-off lies between compatibility of the protocol with typical used software in IoT and the already built-in support of fleet management and OTA update functionality of the protocol.

In Update Installation we have seen a similar trade-off as in Update Security. More memory leads to higher fault-tolerance and recoverability. With approaches like the Dual-Bank approach or the Double-Copy with Rescue System, we have seen approaches that rely on multiple partitions that are

used for the update installation. If the update fails, there is still at least one partition with a working version of the firmware. Car manufacturers need to decide which ECUs need to be equipped with sufficient memory for such approaches.

7.2 Benefits

This thesis benefits software architects, car fleet managers and car manufacturers. Software architects profit by the analysis of certain trade-offs in dissemination techniques, as well as in software security. We have provided advantages and disadvantages of architectural decisions, dissemination approaches and security measures. Depending on the use case, it is easier for a software architect to decide on the most suitable design.

Car fleet managers can use the trade-offs we observed in fleet management, regarding static or dynamic fleets and continuous or snapshot updates. Depending on the use case they can decide for the best approach for their fleets. Also, if multiple dissemination strategies are offered by an OTA infrastructure, they can choose for the best strategy for their use case based on our analysis.

Car manufacturers can use the information on certain installation approaches on ECUs or the concepts of full or partial verification in Uptane, to decide on the computational and memory capabilities of different ECUs they deploy in their cars. For instance using more memory in ECUs that are crucial for safety and also difficult or impossible to replace in case of an update failure, is very important, to prevent costly replacements.

7.3 Limitations

The results that we observed are likely to not uphold in a setup, where caching on the OTA update server is used. That would reduce the number of queries to the image database and subsequently reduce the amount of data to be sent between image database and main service, overcoming the network bottleneck we observed.

Also, we have only observed a scenario, where cars have virtually no delay in their communication with the server, since the cars where simulated on the same cluster, as the OTA update service.

Nevertheless, the general trade-offs we formulated are not directly effected by those limitations, only our experimentally observed results and conclusions about which of those approaches fits which use case best, is limited by the assumption of no caching in place. This especially effects the Push approach (performance difference changes with caching).

7.4 Lessons Learned

During the thesis I learned to work with Kubernetes. Especially I learned about the advantages of using an IDE instead of plain kubectl commands. Only with Lens I was able to perform the tests in an effective and convenient way while also observing certain cluster metrics like CPU consumption.

I also learned to work with SpringBoot and a lot about web services in general. SpringBoot provided a variety of different tutorials which helped me in developing the prototype.

As already explained in chapter 3, we think that it is generally better to implement an own prototype to explore trade-offs or to analyze certain techniques. This is sensible, since in most cases this is the only way to isolate certain techniques from others. In IoT management frameworks a lot of different actions are involved apart from the OTA update itself, which makes it difficult to compare techniques, especially if they are implemented in different frameworks and one would need to compare the frameworks. The results of that would be influenced by the overall architecture of the frameworks as well, such that it is hard to interpret observed results.

It was very important to look on the metrics provided by the cluster. Otherwise, an interpretation like the observed bottlenecks and the increase of response time when scaling up the services with high image load would have been impossible.

Even though Gatling does not directly provide a way to simply listen to queues, we were able to misuse the construct of `requestReply` of the AMPQ plugin [Tin23] to shape it in a way that enabled us to let Gatling user instances to listen only to queues that are connected to our web server.

7.5 Future Work

Our built library can be reused for other tests with simulated cars. One idea would be to change the setup of the cars. Instead of using a load driver, which simulates all the cars in one place, the cars could be distributed over different places to see the impact of their spacial distribution.

Another option would be to change the channel of communication with the OTA update server, since cars are simulated on the cluster and have virtually no latency to the server. For instance, it would be interesting to use a 5G connection to the server and see how response and completion times change.

The impact of image caching is another interesting topic to look at, since we implemented all the approaches without any caching, and it is reasonable to assume that Optional and Polling Intervals can perform a lot better with caching, since the database does not need to be queried as often. For that the framework as well as the Gatling load tests would need to be modified, such that not only one rollout per simulation is modeled, but also multiple rollouts. This is necessary, since only then caching is challenged in a sensible way. If only one specific image is requested, caching is trivial. Only if multiple rollouts are started and not all results can be cached, every use case would be covered. What needs to be changed in such a case at the server, is that the selection of the delta mode is not global. Since we only assumed one rollout at a time out of simplicity, this leads to a situation where all rollouts are delta or all rollouts are non-delta. For that every rollout needs to be saved in the database and not only the continuous ones.

Finally, studying the impact of security measures as they are proposed in Uptane (section 3.2.6) can be an interesting topic. Since we have not implemented any security infrastructure for our framework, it would be a helpful insight to see the effect of important security measures on the overall performance observed by car drivers and administrator.

Bibliography

[20] *Understanding Automotive OTA (Over-the-Air Update)*. en-US. June 2020. URL: https://www.pathpartnertech.com/understanding-automotive-ota-over-the-air-update/ (visited on 08/31/2022) (cit. on p. 1).

[22a] *Docker: Accelerated, Containerized Application Development*. en-US. May 2022. URL: https://www.docker.com/ (visited on 02/14/2023) (cit. on p. 41).

[22b] *Gatling - Professional Load Testing Tool*. en-US. 2022. URL: https://gatling.io/ (visited on 01/27/2023) (cit. on p. 37).

[22c] *Quality of service*. en. Page Version ID: 1113826659. Oct. 2022. URL: https://en.wikipedia.org/w/index.php?title=Quality_of_service&oldid=1113826659 (visited on 10/17/2022) (cit. on p. 3).

[22d] *UpdateHub/updatehub: A generic and safe Firmware Over-The-Air (FOTA) agent for Embedded and Industrial Linux-based devices*. 2022. URL: https://github.com/UpdateHub/updatehub (visited on 11/21/2022) (cit. on p. 25).

[ABG+17a] C. E. Andrade, S. D. Byers, V. Gopalakrishnan, E. Halepovic, M. Majmundar, D. J. Poole, L. K. Tran, C. T. Volinsky. "Managing Massive Firmware-Over-The-Air Updates for Connected Cars in Cellular Networks". en. In: *Proceedings of the 2nd ACM International Workshop on Smart, Autonomous, and Connected Vehicular Systems and Services*. Snowbird Utah USA: ACM, Oct. 2017, pp. 65–72. ISBN: 978-1-4503-5146-1. DOI: 10.1145/3131944.3131953. URL: https://dl.acm.org/doi/10.1145/3131944.3131953 (visited on 08/29/2022) (cit. on p. 1).

[ABG+17b] C. E. Andrade, S. D. Byers, V. Gopalakrishnan, E. Halepovic, D. J. Poole, L. K. Tran, C. T. Volinsky. "Connected cars in cellular network: a measurement study". en. In: *Proceedings of the 2017 Internet Measurement Conference*. London United Kingdom: ACM, Nov. 2017, pp. 235–241. ISBN: 978-1-4503-5118-8. DOI: 10.1145/3131365.3131403. URL: https://dl.acm.org/doi/10.1145/3131365.3131403 (visited on 08/29/2022) (cit. on p. 1).

[All] O. M. Alliance. *LWM2M specification*. URL: https://lwm2m.openmobilealliance.org/ (visited on 03/13/2023) (cit. on p. 13).

[Ama] I. Amazon Web Services. *OTAUpdateInfo - AWS IoT*. URL: https://docs.aws.amazon.com/iot/latest/apireference/API_OTAUpdateInfo.html (visited on 02/19/2023) (cit. on pp. 15, 27).

[ASa] N. AS. *Delta update | Mender documentation*. URL: https://docs.mender.io/overview/delta-update (visited on 02/19/2023) (cit. on p. 27).

[ASb] N. AS. *Polling intervals | Mender documentation*. URL: https://docs.mender.io/client-installation/configuration-file/polling-intervals (visited on 02/18/2023) (cit. on pp. 26, 27).

[Ast] Astarte. *Astarte – The Data Orchestration Platform*. en-US. URL: https://astarte.cloud/ (visited on 03/02/2023) (cit. on pp. 7, 8).

[Aut] gRPC Authors. *gRPC*. URL: https://grpc.io/ (visited on 03/10/2023) (cit. on p. 12).

[BRG+20] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, E. D. Poorter. "Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles". In: *IEEE Communications Magazine* 58.2 (Feb. 2020). Conference Name: IEEE Communications Magazine, pp. 35–41. ISSN: 1558-1896. DOI: 10.1109/MCOM.001.1900125 (cit. on pp. 3, 6, 20).

[Die] J. Dieter. *What is zchunk?* URL: https://www.jdieter.net/posts/2018/05/31/what-is-zchunk/ (visited on 03/14/2023) (cit. on p. 15).

[Eli] Elixir. *elixir-lang.github.com*. en. URL: https://elixir-lang.org/ (visited on 03/02/2023) (cit. on p. 7).

[Gro] I. H. W. Group. *HTTP Documentation*. en. URL: https://httpwg.org/specs/ (visited on 03/13/2023) (cit. on p. 13).

[HER] HERE. *Introduction | OTA Connect Documentation*. en. URL: https://docs.ota.here.com/ota-client/latest/index.html (visited on 03/22/2023) (cit. on p. 19).

[Inca] M. Inc. *MinIO | High Performance, Kubernetes Native Object Storage*. en. URL: https://min.io (visited on 02/01/2023) (cit. on p. 28).

[Incb] M. Inc. *Lens | The Kubernetes IDE*. URL: https://k8slens.dev/ (visited on 03/04/2023) (cit. on p. 43).

[Incc] M. Inc. *MongoDB: The Developer Data Platform*. en-us. URL: https://www.mongodb.com (visited on 02/01/2023) (cit. on p. 28).

[Incd] V. Inc. *Messaging that just works — RabbitMQ*. URL: https://www.rabbitmq.com/ (visited on 02/01/2023) (cit. on p. 28).

[KKC+16] M. Khurram, H. Kumar, A. Chandak, V. Sarwade, N. Arora, T. Quach. "Enhancing connected car adoption: Security and over the air update framework". In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. Dec. 2016, pp. 194–198. DOI: 10.1109/WF-IoT.2016.7845430 (cit. on pp. 5, 6).

[KML+] T. Karthik, D. McCoy, S. Lauzon, A. Brown, S. Awwad, R. Bielawski, C. Mott, A. Weimerskirch, J. Cappos. "Uptane: Securing Software Updates for Automobiles". en. In: (), p. 11 (cit. on p. 16).

[Lab] A. Labs. *OTA Update Cost Considerations*. en-US. URL: https://www.auroralabs.com/ota-ccg-lp-1/ (visited on 03/04/2023) (cit. on pp. 6, 41).

[Lim] P. Limited. *Differential OTA Update*. URL: https://docs.pycom.io/updatefirmware/diff_ota/ (visited on 02/19/2023) (cit. on p. 27).

[Mac] J. MacDonald. *xdelta*. URL: http://xdelta.org/ (visited on 03/14/2023) (cit. on p. 15).

[Man22] E. D. Manager. *Edgehog*. original-date: 2021-11-02T17:44:22Z. June 2022. URL: https://github.com/edgehog-device-manager/edgehog (visited on 10/19/2022) (cit. on p. 25).

[Men22] Mender. *Mender: over-the-air updater for embedded Linux devices*. original-date: 2015-12-07T09:36:43Z. Oct. 2022. URL: https://github.com/mendersoftware/mender (visited on 10/19/2022) (cit. on pp. 9, 25).

[mqt] mqtt.org. *MQTT - The Standard for IoT Messaging*. URL: https://mqtt.org/ (visited on 03/06/2023) (cit. on pp. 8, 13).

[MSAA21] M. A. Maruf, A. Singh, A. Azim, N. Auluck. "Faster Fog Computing based Over-the-air Vehicular Updates: A Transfer Learning Approach". In: *IEEE Transactions on Services Computing* (2021). Conference Name: IEEE Transactions on Services Computing, pp. 1–1. ISSN: 1939-1374. DOI: 10.1109/TSC.2021.3099897 (cit. on p. 5).

[NL08] D. K. Nilsson, U. E. Larson. "Secure Firmware Updates over the Air in Intelligent Vehicles". In: *ICC Workshops - 2008 IEEE International Conference on Communications Workshops*. ISSN: 2164-7038. May 2008, pp. 380–384. DOI: 10.1109/ICCW.2008.78 (cit. on p. 5).

[Ora] Oracle. *RandomAccessFile (Java Platform SE 7)*. URL: https://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html (visited on 02/08/2023) (cit. on p. 33).

[PB22] S. Palm, R. Bui. *Over-The-Air update techniques and how to evaluate them : A comparison of Over-The-Air updates for type ESP-32*. eng. 2022. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-115083 (visited on 09/20/2022) (cit. on p. 6).

[Poo] M. Pool. *librsync: README*. URL: https://librsync.github.io/ (visited on 03/14/2023) (cit. on p. 15).

[RP21] C. Rupp, K. Pohl. *Basiswissen Requirements Engineering*. Mar. 2021. ISBN: 978-3-96910-247-3. URL: https://content-select.com/de/portal/media/view/603e7138-368c-40c9-88ac-05bfb0dd2d03?forceauth=1 (visited on 10/17/2022) (cit. on p. 4).

[SCL+21] A. K. Srivastava, K. CS, D. Lilaramani, R. R, K. Sree. "An open-source SWUpdate and Hawkbit framework for OTA Updates of RISC-V based resource constrained devices". In: *2021 2nd International Conference on Communication, Computing and Industry 4.0 (C2I4)*. Dec. 2021, pp. 1–6. DOI: 10.1109/C2I454156.2021.9689433 (cit. on p. 14).

[SHB14] Z. Shelby, K. Hartke, C. Bormann. *The Constrained Application Protocol (CoAP)*. Request for Comments RFC 7252. Num Pages: 112. Internet Engineering Task Force, June 2014. DOI: 10.17487/RFC7252. URL: https://datatracker.ietf.org/doc/rfc7252 (visited on 03/13/2023) (cit. on p. 13).

[Srla] S. M. S.r.l. *OTA Updates — Edgehog v0.5.0*. URL: https://docs.edgehog.io/snapshot/ota_updates.html#content (visited on 03/02/2023) (cit. on pp. 7, 9).

[Srlb] S. M. S.r.l. *OTA Updates — Edgehog v0.5.0*. URL: http://docs.edgehog.io/snapshot/ota_updates.html#managed-ota-updates (visited on 02/19/2023) (cit. on pp. 27, 51).

[SSLK21] K. Sowmya, C. Srinivasan, K. V. Lakshmy, T. Kumar Bansal. “A Secure Protocol for the Delivery of Firmware Updates over the Air in IoT Devices”. en. In: *Soft Computing and Signal Processing*. Ed. by V. S. Reddy, V. K. Prasad, J. Wang, K. T. V. Reddy. Advances in Intelligent Systems and Computing. Singapore: Springer, 2021, pp. 213–224. ISBN: 978-981-336-912-2. DOI: 10.1007/978-981-33-6912-2_20 (cit. on p. 5).

[Sta] I. O. for Standardization. *ISO 25010*. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (visited on 09/18/2022) (cit. on p. 4).

[thi] thingsboard. *Over-the-air firmware and software updates*. URL: https://thingsboard.io/docs/user-guide/ota-updates/ (visited on 03/06/2023) (cit. on pp. 11, 12).

[Tin23] Tinkoff.ru. *Gatling AMQP Plugin*. original-date: 2019-06-21T09:20:59Z. Jan. 2023. URL: https://github.com/Tinkoff/gatling-amqp-plugin (visited on 04/04/2023) (cit. on pp. 40, 57).

[Toc19] S. Tockey. *How to Engineer Software: A Model-Based Approach*. en. Google-Books-ID: RGatDwAAQBAJ. John Wiley & Sons, Nov. 2019. ISBN: 978-1-119-54662-7 (cit. on p. 4).

[Upt22] M. of the Uptane Community. *Uptane Standard for Design and Implementation 2.0.0*. en. Mar. 2022. URL: https://uptane.github.io/papers/uptane-standard.2.0.0.html (visited on 03/21/2023) (cit. on p. 16).

[VA20] M. M. Villegas, H. Astudillo. “OTA updates mechanisms: a taxonomy and techniques catalog”. en. In: ISSN: 2451-7593. 2020. URL: http://sedici.unlp.edu.ar/handle/10915/115196 (visited on 09/20/2022) (cit. on pp. 3, 5, 20).

[VMwa] I. VMware. *AmqpAdmin (Spring AMQP 3.0.1 API)*. URL: https://docs.spring.io/spring-amqp/api/org/springframework/amqp/core/AmqpAdmin.html (visited on 02/06/2023) (cit. on p. 31).

[VMwb] I. VMware. *MongoTemplate (Spring Data MongoDB 4.0.1 API)*. en. URL: https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/core/MongoTemplate.html (visited on 02/06/2023) (cit. on p. 31).

[VMwc] I. VMware. *Spring Boot*. en. URL: https://spring.io (visited on 02/01/2023) (cit. on p. 28).

All links were last followed on April 13, 2023.

1. Record architecture decisions

Date: 2023-03-29

Status

Accepted

Context

We need to record the architectural decisions made on this project.

Decision

We will use Architecture Decision Records, as described by Michael Nygard.

Consequences

See Michael Nygard's article, linked above. For a lightweight ADR toolset, see Nat Pryce's adr-tools.

2. Implement approach that uses notifications and download

Date: 2023-03-29

Status

Accepted

Context

This approach is used in Edgehog, AWS IoT OTA library and Thingsboard. It is an approach that is typically used in IoT management. That is why it makes sense to implement it in our prototype.

Decision

We decided to use a broker for the notifications to enable a quick and simple way of notifying the clients about a new update. Cars download the update image from our REST-API after receiving the notification.

Consequences

We have an approach that enables notification and download. Impact will be high load when all cars start the update automatically.

3. Implement approach that uses broker to offer the Update Image

Date: 2023-03-31

Status

Accepted

Context

Offering the image directly may save some time compared to approaches, where notifications need to be sent out first.

Decision

Images are directly published to a fanout exchange, that offers the image to the cars' message queues.

Consequences

Could lead to problems with large image sizes.

4. Implement approach where cars frequently poll for new updates

Date: 2023-03-31

Status

Accepted

Context

To increase security by not offering any open ports at the car, it makes sense to use polling intervals to frequently check if new updates are available.

Decision

Cars will poll in a predefined polling interval. For that the `newImageAvailable` endpoint with the car's ID is called frequently. If that returns true, the `getNewImage` endpoint is called with the car's ID and the image is received.

Consequences

Security off the update process is increased, also load can be controlled more easily, since the number of cars that are polling in a certain time interval can be controlled by the chosen polling interval.

5. Implement dissemination approaches as continuous or snapshot

Date: 2023-03-31

Status

Accepted

Context

For management reasons it makes sense that cars that are newly added to a fleet are immediately updated with software that should be deployed, when being part of that fleet. For that we use the concept of continuous jobs used in the AWS IoT library.

Decision

For every update rollout it can be defined if it is a continuous or snapshot rollout. Using snapshot does not require any specific actions. Using continuous, the rollout needs to be registered in the database, such that every car that is added to the fleet after rollout start is automatically updated with the proper image.

Consequences

Using continuous updates can reduce the effort an administrator needs to put into the fleet management, since those updates are automatically supervised by the system. When using snapshot an administrator would need to start a new rollout for the whole fleet.

6. Use Delta Updates as rollout option

Date: 2023-03-31

Status

Accepted

Context

Reducing the amount of data to be sent is an important issue, since it reduces update completion times as well as response times when cars try to pull a new image.

Decision

We emulate a delta update by waiting a certain predefined time. A random image is created with 1-DeltaSimilarity the size of the old image. This emulates the reduction in update data that can be achieved in the best case delta algorithm. In our simulations we used a delta simulation time of 0, since the time needed for the delta creation can just be added to the update completion time of the fleet, when only one delta image needs to be created for the whole fleet.

Consequences

Delta updates reduce the overall amount of data to be sent. In our setup delta updates only work when the whole car fleet has the same update state before the rollout.

7. Use the concepts of static groups

Date: 2023-03-31

Status

Accepted

Context

Cars need to be grouped in fleets, one approach is to a unique ID for every car. A list of IDs represents a fleet or car group.

Decision

A car group is saved as a list of IDs and a group ID. Also, other information about the group is provided. In our case current and previous installed image for that whole group. (Assuming only one consistent update state)

Consequences

Car groups can be modeled now.

8. Use the concept of dynamic groups

Date: 2023-04-01

Status

Accepted

Context

In some instances it makes sense to define groups dynamically over constraints that define the cars / devices that should be part of the group. For instance, a car should have hardware xy to be in the group.

Decision

Not implemented in our prototype.

Consequences

Handling multiple fleets can become easier, especially if cars are meant to be switched between groups, this can be done by changing the pivotal variable, which leads the system to regroup the car. Also, rollouts can be secured regarding compatibility if constraints are used like a certain hardware type.

9. Provide manual updates

Date: 2023-04-01

Status

Accepted

Context

Providing manual updates can help when only one car in a fleet needs a fix.

Decision

Not implemented in our prototype.

Consequences

Important feature that enables corrections for certain cars. May involve some risks if it is not semantically checked, if the software that gets installed is really suiting the purpose of the car or the fleet.

10. Use MQTT for communication with the cars

Date: 2023-04-01

Status

Accepted

Context

MQTT is lightweight and well-supported, which makes it a good choice in many IoT tasks, also for OTA updates.

Decision

Not used in our prototype (We use AMQP)

Consequences

More lightweight than HTTP, which makes it more suitable for resource constrained ECU types. Downside: Not as widely supported as HTTP.

11. Provide one API for admins and one API for cars

Date: 2023-04-01

Status

Accepted

Context

It makes sense to logically split the API into an admin and a device / car part.

Decision

We used that in the prototype. Admin API: ota-tester.com/admin/... Car API: ota-tester.com/car/...

Consequences

The different URLs make it easier to distinguish between admin and car requests.

12. Use a checksum in update delivery

Date: 2023-04-01

Status

Accepted

Context

It must be ensured that the update file is not changed during transmission.

Decision

Not implemented in the prototype.

Consequences

Checksums ensure that the image is not changed during the transmission. The checksum needs to be checked at the ECU. Depending on the ECU this may be difficult. To increase security even check with an original checksum provided by the server, to make sure that it really came from the intended origin.

13. Use two partitions on the ECU

Date: 2023-04-01

Status

Accepted

Context

When only one partition is used, a full image update involves some risks. If power is cut during the update or anything else goes wrong, the ECU is broken. Replacement can be very expensive or even impossible.

Decision

Not implemented in our prototype. Cars are modeled much simpler in our case.

Consequences

Using two partitions, where only one is active, enables robust updates. During update the inactive partition is used to flash the new image. If everything went well the inactive partition is made

active after reboot. In any other case the system can safely roll back to the previous version by using the old partition.

14. Use a gateway between some clients and the server

Date: 2023-04-01

Status

Accepted

Context

To provide better load control options or compatibility with VPNs it can make sense to use a gateway between server and devices / cars.

Decision

Not used in our prototype.

Consequences

This is probably more suitable in a general IoT context, when managing a device fleet in a company, where every device is connected to the private company network. Cars are mostly connected to cellular networks. But for load balancing it can still make sense.

15. Use TLS to secure the connection

Date: 2023-04-01

Status

Accepted

Context

For confidentiality, authenticity and integrity of the communication channel TLS is one of the most used choices.

Decision

Not used in our prototype.

Consequences

Connection is secure. Only small overhead created for decryption and authentication. Not compromise resilient to attacks on the server!

16. Use message queues for communication between the services

Date: 2023-04-01

Status

Accepted

Context

Persistent message transmission, as well as queuing is needed for high load.

Decision

Not used in our prototype.

Consequences

Adds a small overhead in latency compared to other approaches like RPC. Great advantage lies in its persistent message transport.

17. Use RPC for communication between the services

Date: 2023-04-01

Status

Accepted

Context

When needing very fast response times RPC can make sense to achieve that, since it has less overhead than approaches like message queues.

Decision

Not used in our prototype.

Consequences

Good in low latency environments. When having high load it can get critical, since requests are not resent if they are not answered.

18. Use a monolithic architecture

Date: 2023-04-01

Status

Accepted

Context

To reduce the effort that is needed to maintain a complex system, a monolithic architecture can be helpful in smaller setups. For example having a framework that only serves a small company fleet.

Decision

Not used in our prototype.

Consequences

When having larger fleets other architecture types might be more sensible, since monolithic architectures are generally not horizontal scalable. Also, there are downsides in agility, since changes in one component, might involve changes in other components too, since they are closely coupled.

19. Use a microservices architecture

Date: 2023-04-01

Status

Accepted

Context

For large setups scalable solutions are necessary.

Decision

Used in our prototype. We have multiple services: MongoDB, MinIO, RabbitMQ and our main OTA service.

Consequences

Components in a microservices architecture can be scaled independently, changes to the overall system like adding new functionality, can be applied more easily, since components are only coupled loosely.

20. Use HTTP for communication with the cars

Date: 2023-04-01

Status

Accepted

Context

Using an uncomplicated straight-forward approach to communicate with the devices can be the best choice.

Decision

We used HTTP for the Polling Intervals approach and also for image delivery in Optional.

Consequences

HTTP uses TCP which creates a small overhead. But it is a widely used protocol that can easily be integrated in every solution. For resource constrained networks, or ECUs something more lightweight can be more effective.

21. Use CoAP for communication with the cars

Date: 2023-04-01

Status

Accepted

Context

A lightweight solution, which already provides a certain amount of IoT management functionality can be helpful.

Decision

Not used in our prototype.

Consequences

Helpful in resource constrained networks or with resource constrained ECUs. Not as compatible as more popular protocols like HTTP or MQTT.

22. Use LWM2M for communication with the cars

Date: 2023-04-01

Status

Accepted

Context

It can be helpful to have a protocol that already provides OTA update and IoT management functionality.

Decision

Not used int our prototype.

Consequences

Reduces implementational effort for IoT management or OTA updates. But often needs further implementation, since it is not as compatible with already available software as other protocols.

23. Only use one partition for installation

Date: 2023-04-01

Status

Accepted

Context

Having only very limited memory capabilities on the ECU this approach might be the only choice.

Decision

Not used in our prototype. Car are simulated much simpler.

Consequences

The risk for errors is reduced since a self-contained firmware image is used. The problem is, if something goes wrong the ECU is broken.

24. Use three partitions for installation

Date: 2023-04-01

Status

Accepted

Context

To increase robustness and fault-tolerance even further it makes sense to use even a third partition as a rescue system on a different memory.

Decision

Not used in our prototype, since we simulated the cars much simpler.

Consequences

Even if the whole memory of the ECU breaks during an update (so both partitions) there is still a secondary memory unit that provides a rescue backup of the old version. Installation can take a bit longer, since this rescue version needs to be updated after every rollout.

25. Split application and OS updates

Date: 2023-04-01

Status

Accepted

Context

Application and OS update should be split into parts to reduce downtime.

Decision

Not used in our implementation: Simplicity of car simulation.

Consequences

Downtime is reduced when only the OS gets updated at first. The OS update can be verified separately, and then the application gets updated.

26. Librsync as Delta algorithm

Date: 2023-04-01

Status

Accepted

Context

Reducing the overall data to be sent in an update is important for response, completion and installation times.

Decision

Not used in our prototype.

Consequences

Images are split in blocks and checksums are created for the blocks. If two checksums are equal, the corresponding block does not need to be sent. The amount of data to be sent gets reduced by the size of the redundant blocks. If the differences between previous and current image reach a certain threshold, delta images can even be larger than the actual image. One advantage is that this approach only needs small amounts of memory on the server compared to other approaches.

27. Xdelta as delta algorithm

Date: 2023-04-01

Status

Accepted

Context

Reducing the overall data to be sent in an update is important for response, completion and installation times.

Decision

Not used in our prototype.

Consequences

Images are split in blocks and checksums are created for the blocks. If two checksums are equal, the corresponding block does not need to be sent. The amount of data to be sent gets reduced by the size of the redundant blocks. If the differences between previous and current image reach a certain threshold, delta images can even be larger than the actual image. One advantage is that this approach only needs sall amounts of memory on the server compared to other approaches.

28. ZChunk as delta algorithm

Date: 2023-04-01

Status

Accepted

Context

Reducing the overall data to be sent in an update is important for response, completion and installation times.

Decision

Not used in our prototype.

Consequences

ZChunk needs more memory as other approaches since it saves data in chunks fragmented over the whole memory. A metadata file is used which describes which chunks are needed for the update. Cars can then determine which chunks need to be downloaded for the update.

29. Use the uptane standard to provide compromise resilience

Date: 2023-04-03

Status

Accepted

Context

When using mechanisms like TLS to protect the connection between server and cars, no security is provided if a set of keys or a server is compromised. For that Uptane helps in providing compromise resilience, even if a server or a subset of keys is compromised.

Decision

Not included in our prototype (simplicity).

Consequences

By providing metadata signing with a PKI and the use of at least two servers, where one provides on-demand signing and a second server that uses offline keys to sign metadata and images, and the use of full and partial verification on the device side, Uptane is able to provide compromise resilience. There are certain trade-offs between using full vs partial verification. Generally we think that a scheme like Uptane is not optional in automotive industry.

30. Use full verification to validate the update

Date: 2023-04-03

Status

Accepted

Context

To provide maximum security it makes sense to check every metadata type with its signature on its validity in Uptane.

Decision

Not used in our prototype.

Consequences

The update process becomes more secure, but the memory consumption as well as the computational requirements increase. Not every ECU type supports that.

31. Use partial verification to validate the update

Date: 2023-04-03

Status

Accepted

Context

When having less computational capabilities or an ECU with only small amounts of memory, one option is the use of partial verification in Uptane. For that, only target and snapshot metadata gets validated.

Decision

Not used in our prototype.

Consequences

Security is reduced in this approach, allowing more attack options. But it is applicable to a greater variety of different ECU types with high resource constraints. To provide more security a primary ECU should perform full verification and send the respective images and metadata to the secondaries, which perform at least partial verification. With that security is much higher with fewer attack options, like compromising the primary ECU.

32. Use OSTree with TreeHub to enable Delta Updates

Date: 2023-04-03

Status

Accepted

Context

Reducing the amount of data to be sent is very important to reduce response and completion times and by that the observed quality of car drivers and fleet administrators.

Decision

Not used in our prototype.

Consequences

The git-like structure of OSTree enables to only download files that have changed since the last commit. This greatly reduces the amount of data to be sent. For that a metadata file is used that contains the commit identifier. The car subsequently downloads the respective files to perform the update.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature