Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Update Management for Wireless Sensor Nodes

Denis Fedorov

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Kálmán Képes, M.Sc.
Commenced:	March 5, 2018
Completed:	September 5, 2018

Abstract

The Internet of Things (IoT) is a growing field of Information Technology. It comes into domains, such as medicine, agriculture, transportation and manufacturing. IoT promises these domains a seamless delivery of relevant data sent by a plethora of different sensors. As every application changes over time to enhance functionality or remove errors, the software used to manage the sensors must enable the update of relevant components in a seamless manner, as well. As IoT applications potentially use many sensors a wired communication is not sufficient in terms of cost and resources, therefore, the communication between IoT nodes is established using various Wireless Technologies. However, a wireless connections is often not as reliable as wired ones, conditions such as bad weather or leaving a cell in a network may cause disconnections. Additionally, sensors in the IoT paradigm are often managed by devices that are constrained in processing, storage and network capabilities. Disruptions during the update of software adapters that read the data from the sensors, may cause the devices hosting the adapters to break, or interrupt the data delivery for application components to consume.

In this Master thesis we surveyed and analyzed existing approaches and technologies enabling the seamless update process of software components and the data delivery in IoT applications. Then, we present an approach based on Dynamic Software Updating techniques and IoT Patterns, and supporting architecture for the update package delivery to IoT nodes and the updating of software components on IoT devices. As a result of this thesis, a prototypical implementation of the system to support the delivery of update package to devices and the execution of component update on devices is provided.

Contents

List of Figures

List of Tables

Listings

1 Introduction

In the modern world we are surrounded by a plethora of different electronic devices, gadgets and smartphones, which are the integral part of our life. The paradigm called Internet of Things (IoT) unites all electronic devices, so-called Things [ALM10]. On the base of IoT, plenty of applications are built in different fields of our everyday life, such as, industry, agriculture, healthcare and logistics [ALM10]. As every normal application, IoT applications change with time, to remove errors or to introduce new functionality. To bring all necessary improvements IoT applications need to deliver updates to all of its distributed devices, also called nodes. Wire technologies in that case are not sufficient in terms of cost and resources, as IoT applications potentially use many nodes, therefore communication in general and for update delivery between IoT nodes in particular is established by various wireless technologies [SMZL15]. However, a wireless connection is often not as reliable as wired ones, conditions such as bad weather or leaving a cell in a network may cause disconnections [PH07]. Additionally, sensors in the IoT paradigm are often managed by devices that are constrained in processing, storage and network capabilities [PH07]. Disruptions during the update of software components that read the data from the sensors, may cause the devices hosting the components to break, or interrupt the data delivery for back end application components to consume.

Nowadays in IoT field exist lots of different patterns solving huge amount of problems, but anyway not all of them fit to IoT device update management and the field for experiments is still wide. Also, some problems are better solved by using combinations of patterns and techniques, sometimes even not belonging to IoT field, but taken from mutual areas like desktop software engineering. The main driver of this work became possibility of combining several patterns and techniques from IoT and mutual fields. Those fields are Dynamic Software Update (DSU) and Device Shadow concept already implemented by several software giants, e.g. Microsoft[1] or Amazon[2]. Also, many companies nowadays are implementing their own approaches for a device Update Management into different off-the-shelf devices. This was another motivating factor for our work, because of possibility of using one company's use case in our Implementation and Validation Chapter.

In this thesis we present an update management approach for IoT devices. This approach combines concepts of DSU [SAM12] and IoT patterns [GBF+16; RBF+16] to implement more seamless update process in IoT applications. To achieve that, the analysis of existing works for Update Management of IoT end devices and gateways, as well as Dynamic Software Updates was done. Then, a conceptual method enabling the Update Management of IoT devices and their components was developed. At the end, prototype of the developed conceptual method in an IoT system for Device Management and device Update Management was implemented.

[1] https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins
[2] https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html

In the following, the overall structure of the thesis work is presented, with short descriptions of each chapter, for better understanding:

- **Chapter 2 - Fundamentals:** In this chapter an overview of the needed fundamentals to understand our approach is given.
- **Chapter 3 - Related Work:** In this chapter an overview of necessary Related Work in area of Dynamic Software Update and IoT Patterns to help development of our approach is given.
- **Chapter 4 - Concept:** In this chapter our approach for Update Management in IoT field and detailed overview of its steps are presented.
- **Chapter 5 - Implementation and Validation:** In this chapter we describe the prototypical implementation and the validating use case to show the feasibility of our approach.
- **Chapter 6 - Conclusion and future work:** This last chapter presents the results of this work and discusses future work.

2 Fundamentals

In this chapter we will describe fundamentals of IoT, such as, patterns and concepts necessary for understanding of the main concept of the work. In Section 2.1 we describe IoT basics, reference architecture of IoT, Device Twin concept and several IoT related protocols used in our implementation. Section 2.2 describes modern virtualization technology called Container. Section 2.3 presents difference between constrained and unconstrained devices. Then in Section 2.4 we briefly describe state of Device Management in IoT. In the last Section 2.5 Dynamic Software Update basics and techniques are overviewed.

2.1 Internet of Things

In the near future more and more devices will be inter-weaved with everyday objects via the Internet. Even nowadays RFID (Radio Frequency IDentification) tags are used in companies to track actions of employees or on factories to see work pieces through the whole production cycle. Or for example buses send their location information via network so the navigation applications installed on smartphones can calculate the best route based on real time context information. All that interconnection of devices and things with these devices integrated, via the Internet is called an IoT [ALM10]. Devices with sensors connected with each other via wireless technology can form a groups collecting data and so providing a service of so-called Sensor Networks [RM04] to external applications via the Internet. But in general distributed networks consist of different heterogeneous devices. There are small devices which have little power consumption and performance. The more powerful devices can have hardware enough for conventional Operating Systems (OS) like Linux [BEK14].

On the other hand the IoT has lots of challenges such as low quality of interconnecting communication media, heterogeneous interfaces between different devices, diverse data transmission formats and communication protocols. Moreover, there are big challenges of device orchestration, provisioning and proper security due to their high numbers. Some problems and vulnerabilities must be solved by sending updates to devices.

There exist other components and entities besides devices in an IoT environment. For ease of perception and communication between devices and back end applications intermediate components are used. They abstract access to the devices and produce data. These components also called Middleware [ALM10] and aim to ease a process of applications development for the IoT, so developers do not need to know details of each device and communication technology to use. So for example developer may not know that device using protocol A and another one – protocol B, communicate with each other via the Middleware bridging these devices. Another component of the IoT system is the application, which normally is based on the Middleware and provides to the end

users the functionality of the system [ALM10]. It can be, for example, a User Interface (UI) with a visualization of the gathered data from devices, which can be viewed on your PC or smartphone via web browser and give some control for the devices.

2.1.1 Reference Architecture

In this thesis we are presenting an approach for IoT devices' components update. This approach must base on some architecture, but there is a problem of existence of many diverse architecture models for different usage scenarios. Therefore, it is necessary to have an abstract architecture describing the different components and relations used in the IoT.

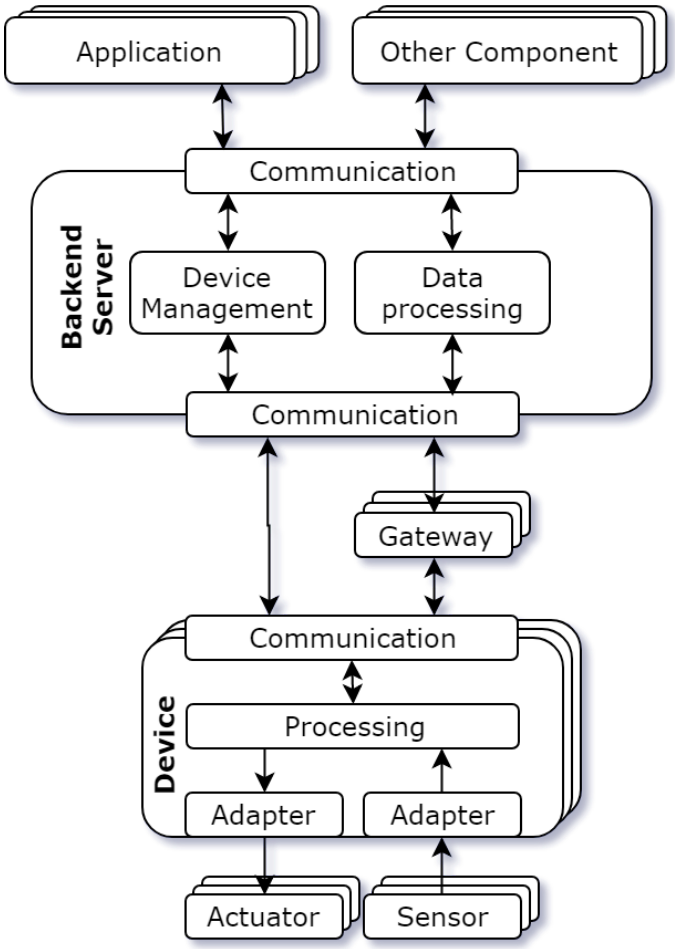

Figure 2.1: IoT Reference Architecture (based on [RBF+16] and [GBF+16])

Such Reference Architecture variant proposed by Guth et al. [GBF+16] was taken as a core and slightly modified without significant changes using another solution suggested by Reinfurt et al. [RBF+16]. Result can be seen on the Fig. 2.1. In the following, components of that Architecture are described in detail.

Sensor is a physical component used for measuring environments properties like temperature, humidity, magnetic or electric fields, etc., converting into digital form and delivering to *Device*. Sensors cannot run any software, but can be configured by it. Most of the time they are integrated into *Devices*, but can be separately plugged.

Next low level component is *Actuator*, which acts or influences on the physical environment, by translating electrical signals in some form of physical action, like servo drive opening the door or led lights blinking. As a *Sensor*, *Actuator* can be part of *Device* or as a separate component receiving signals from *Device*.

Device is hardware component for connecting and integrating *Sensors* and *Actuators*. *Adapters* are used for control over *Sensors* and *Actuators*, also giving received information for *Processing* component, which in turn groups information and sends it via communication component to *Back end Server*. Also, *Devices* can be self-contained and act as small automation systems gathering data from *Sensors* and instantly reacting with *Actuators* according to gained from *Processing* component information.

In cases when *Device*'s communication means are not enough, e.g. particular protocol is not supported, or another additional level of processing is needed, a *Gateway* is used. For example, in our implementation *Gateway* is used as a bridge to translate from one protocol to one used by *Back end Server*.

The *Back end Server* in our representation is the same abstract Middleware component as in [GBF+16], but with small differences. First to mention, it is not a single component, but distributed one, can be spread into the Cloud and be multilevel. Here, we abstracted two important subcomponents *Data Processing* and *Device Management*. First sub component is mostly the same as in *Device*, but in bigger volumes. Second one is needed for control, addressing, updating and information routing of *Devices*, *Gateways*, *Sensors* and *Actuators*. The main goal of that *Middleware* component to make all above-mentioned functionality accessible to *Applications* or *Other Components* via APIs, e.g. via HTTP-based REST APIs.

An *Application* component normally uses a *Back end Server* to control environment via *Actuators* and take measurements from *Sensors*, as was described in *Device*'s self-contained automation example. Also, the *Application* gives to the User direct control with the necessary abstraction level.

Other Component can be another *Application* or *Middleware System* from totally different field.

It is possible that in other architectures of IoT Systems some components can be missing due to narrow specialization of such Systems. Some can have no *Actuators* or miss *Gateways*. For example, *Device* may embed a *Sensor* for temperature data collection and also has communication module able to connect directly to *Back end Server* without help of *Gateway*.

2.1.2 Device Twin concept

In current subsection another important concept of IoT will be discussed and explained. It has plenty of names, most famous of them is Device Shadow [RBF+16] (used by Amazon Web Services[1] [Ama18]), also known as Thing Shadow [RBF+16], Virtual Device[RBF+16] or Device Twin (see Fig. 2.2) (used by Microsoft Azure[2] [Mica]). In the leftover of this work, the term Device Twin is used to refer to the Device Shadow concept.

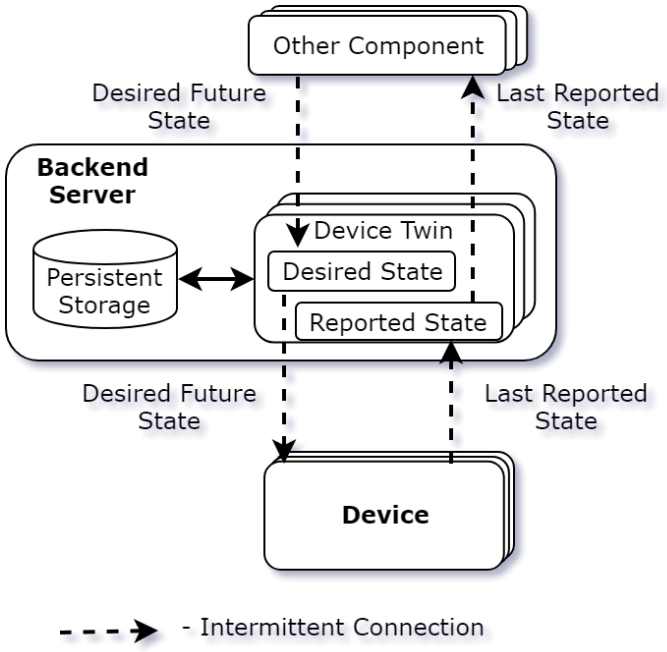

Figure 2.2: Device Twin (based on [RBF+16])

Sometimes devices can be inaccessible by other components, and these components might still need to interact with devices. To solve that, the Device Twin which can store each device's persistent virtual representation can be used. Device Twin can be simple JSON document stored on the back end server or sometimes on a gateway and giving some level of decoupling on different layers of IoT architecture. According to [RBF+16] device state can be read and commands to it can be given even when it is offline or inaccessible. If needed commands can be queued, otherwise fresh ones are taken as relevant. Next time device connects to back end component according to some schedule or was offline cause of network problems, it reports with its last known state and receives commands from desired properties list [Mica]. For differentiation of offline and online state of the device or even for some complex state definition like updating or malfunction, state flag can be defined and used. Also, such flag can be set automatically to offline, if for example, device was not answering in defined time span. That concept gives our update approach a tool to create some sort of indirection between component to update on device and dependant components in back end application.

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/en-us/

2.1.3 Protocols

In the following subsection several transport protocols regarding IoT domain will be described and their usage in the subsequent text is briefly explained.

MQTT

All devices in IoT interact with each other by means of different interfaces and data transfer protocols. One of the most popular protocols for Industrial IoT is MQTT [OAS15]. MQTT or Message Queue Telemetry Transport – is a lightweight, compact and open protocol for data transfer between remote locations, where small code footprint is necessary and there are restrictions in channel bandwidth. MQTT is an asynchronous protocol and can work in intermittent connection media, supports several levels of Quality of Service (QoS) and has easy integration of new devices. It works on application OSI (Open Systems Interconnection model) level and uses 1883 port as default one (8883 for SSL connection). Message exchange in MQTT is executed between publisher and subscriber, which are decoupled by MQTT broker in between them. Publisher sends data to MQTT broker setting some topic. Subscribers receive different data from variety of publishers, depending on the subscriptions to the relevant topics. Topics are UTF-8 Strings represented in hierarchical tree-structure using slash "/" symbol, which eases organization and access to data. Users can define topics in the way they want, and the following string "/home/living-space/living-room1/temperature" is an IoT example of a topic to which the temperature sensor in the living room publishes its data to the MQTT broker. Using wildcards subscribers can get the data from several topics at a time. There are two types of wildcards: single level and multi-level ones:

- First applied with "+", e.g. "/home/living-space/+/temperature" will show us temperature data in all living rooms inside the living-space.
- Second wildcard type can be applied with "#", e.g. "/home/living-space/#" will give us all data from all sensors in all rooms in the living-space.

As is was told earlier MQTT supports several levels of QoS, they are 3:

1. **QoS 0 At most once.** At this lowest level the sender just transmits message to receiver without guarantees that the message arrives. If message is lost neither the sender nor receiver will notice.
2. **QoS 1 At least once.** This middle level guarantees that the message will be delivered to receiver. The receiver must send acknowledgement of receipt of message, otherwise after some certain time sender will repeat message transmission of message. It will be repeated until sender receives confirmation for receiving the message. Resending of messages may result in duplicates, which must be caught by higher level software of recipient.
3. **QoS 2 Exactly once.** In addition to logic of previous level, this highest one offers mean against duplicates. There are 4 messages per single message transfer sent. Sender transmits the message with unique Packet ID (used for Packet identification) and keeps, and keeps it unconfirmed until will receive answer from receiver with the same Packet ID. Then sender sends end transfer message to recipient, which upon receiving it deletes original message and sends confirmation of transaction end to sender.

Each level of QoS demands more efforts from sender and receiver, so their capabilities define necessity for appropriate QoS level.
Security in MQTT is enabled with client authentication using username and password, clients access control via Client ID and connection via SSL or TLS protocols.

CoAP

Another widely used protocol in IoT for machine to machine (M2M) communication is Constrained Application Protocol (CoAP) [SHB14]. CoAP is an application level protocol using the same widely used REST (Representational State Transfer) model as HTTP (Hyper Text Transfer Protocol) protocol with four methods:

- GET - executes search of resources and provides information corresponding to resource in URI (Uniform Resource Identifier) reference, there resource is a source of data, e.g. sensor.
- PUT - requests creation or update of the resource mentioned in URI according to payload of method request.
- POST - used to request processing of the payload part of method request.
- DELETE - asks for deletion of resource mentioned in URI.

This is the reason why CoAP and HTTP can intercommunicate easily using special proxies. As underlying transfer protocol UDP (User Datagram Protocol) is used, enabling usage of SMS (Short Message Service) and other packet based communication protocols. It means CoAP is connectionless comparing to HTTP or MQTT which use TCP (Transmission Control Protocol) for transport. CoAP packets are much smaller than MQTT or HTTP ones, and they are simple to generate and to parse, demanding less RAM (Random Access Memory) in constrained devices.
As it was written previously access to resources is gained via URI – a sequence of symbols identifying physical or abstract resource. URI is also used for identifying the location of resources. It is kept in the header of CoAP packet. There are four types of messages which actually enable QoS for CoAP:

- Confirmable (CON) message includes request or response, requiring confirmation and safe transfer. Each CON message causes one ACK message or R message.
- Non-confirmable (NON) - same as CON message, but needs no confirmation and safe transfer, because is transferred regularly, e.g. sensor data.
- Acknowledgement (ACK) message confirms delivery of CON message, but does not mean success or fail of any method in CON message.
- Reset (R) message shows that CON or NON message was delivered, but corrupted and cannot be processed or recipient is overloaded. Also used for checking of node availability as ping method.

CoAP also has support for negotiation of content, like HTTP. By use of Accept options, preferred representation of resource is requested by client and then Content-Type option in server's reply tells client what representation of resource they will get. That gives client and server possibility to add new representation without being dependent on each other.

For the data transmission in CoAP, a block-wise transfer is implemented. The block-wise transfer is essential in case of an unstable communication channel, so the transfer process if interrupted due to some reason, e.g. leaving network cell or bad weather conditions, then can be successfully continued and eventually finished.

Security connection in CoAP is enabled via DTLS (Datagarm Trasnport Layer Security) [RM12] protocol. It works above UDP and protects data on transport level.

Lightweight M2M

According to [All18b] Lightweight M2M (LwM2M) is the application layer communication protocol for device management and service enabling on resource constrained devices. It uses lightweight and compact protocol mechanisms and provides an efficient data model. LwM2M uses CoAP as resource-constrained communication.

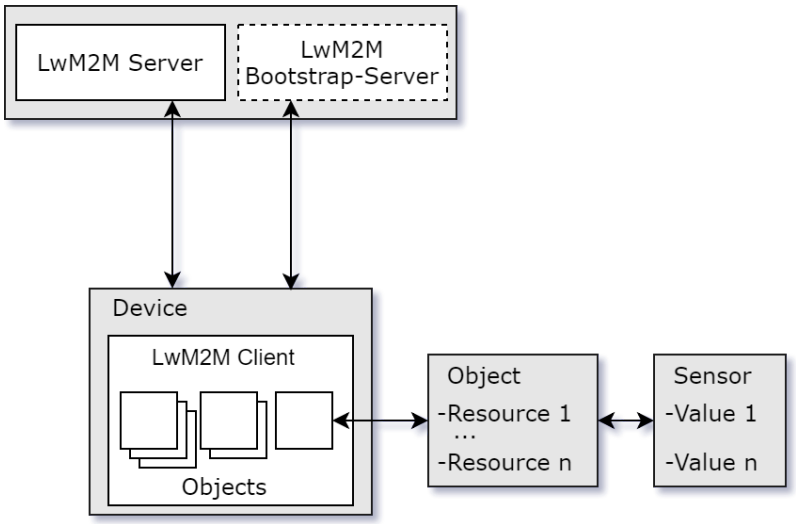

Figure 2.3: LwM2M Architecture (based on [All18b])

LwM2M enabler consists of three entities (see Fig. 2.3): a server, an optional bootstrap server and a client. LwM2M client normally deployed on a constrained Device, e.g. Bosch XDK [DG17] or unconstrained device, e.g. Raspberry Pi [Fou18] or MICA device [Gmb].

According to LwM2M resource-object model, clients include Objects, which can be any components, for example temperature sensor or even container on MICA device. Each such component can have multiple instances, managed separately. Instance has resources, e.g. maximum and minimum temperature values for the temperature node or start, stop, reset for the container. According to LwM2M specification [All18b], resources can be read, written or executed. For example, simple request to LwM2M client's standard Firmware Update object, PkgVersion resource denoting current downloaded update package version, looks like this "Read /5/0/7". Here "5" is object, "0" is instance number and "7" is a resource identificator. The LwM2M server manages registration of devices and processes or retransmits devices' data. More details about protocol will be given in Implementation and Validation Chapter 5.

2.2 Container Technology

Nowadays Virtualization is a frequently-used technology for the simulation of hardware-based resources or components in software. For example, that allows parallel and isolated execution of multiple Operating Systems (OS) or applications on one computer [Vau06]. The software do not need to access hardware directly, and instead uses an abstraction layer. The usage of Virtualization technologies allows the better utilization of hardware resources [Vau06]. Virtualization is important for many areas, like cloud-computing technologies such as Infrastructure as a Service (IaaS). There are a lot of examples of Virtualization solutions, like VirtualBox[3], VMware[4], Xen[5], KVM[6] and Hyper-V[7] available. All of these Hypervisor-based solutions emulate part or whole hardware system on which virtualized OSs are executed [Vau06]. The Hypervisor manages access to existing resources and isolates all virtualized OSs from each other [Vau06]. Each instance of virtualized OS is a full version of OS with its own kernel [MKK15].

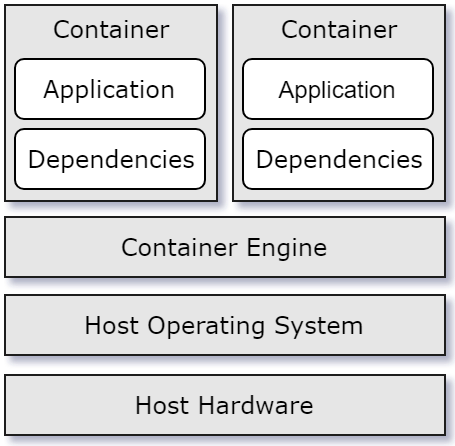

Figure 2.4: Architecture of Container-based Virtualization (based on [Bui15])

Except Hypervisor-based solution, there is also lightweight Container-based virtualization. There are a lot of Container-based solution examples, which are OpenVZ[8], Docker[9], rkt[10], Jetpack [GG18] and LXC[11]. In Container-based virtualization technology all Containers share the kernel of the OS they run on [MKK15]. The individual Containers are isolated processes in the host OS [Bui15].

In the Fig. 2.4 the Architecture of Container-based Virtualization technology is shown. The Host OS is working between Container Engine and Host Hardware, so the Container Engine needs no drivers, because the Hardware is accessed through the Host OS, which already has drivers. The Container

[3] https://www.virtualbox.org/
[4] https://www.vmware.com/
[5] https://www.xenproject.org/
[6] https://www.linux-kvm.org
[7] https://www.microsoft.com/de-de/cloud-platform/server-virtualization
[8] https://openvz.org/
[9] https://www.docker.com/
[10] https://coreos.com/rkt/
[11] https://linuxcontainers.org/

Engine manages the containers, starts and stops them. Inside the containers there are components and all necessary dependencies. For example, in Docker the component is an application, what means application-centric approach, while in LXC the approach is closer to hypervisor-based virtual machines, so the component is a part of Host OS [Bra18; SAG+]. Both Docker and LXC use kernel techniques cgroups and namespaces [MKK15; SAG+]. The namespaces ensure that the resources of the OS are isolated and available only for individual process groups. The cgroups classify processes into a tree data structure, where each process-node can be assigned system resource limits, such as memory and CPU (Central Processing Unit) [HJD+17]. All the processes located below the node in the tree share the resource limits applied to the node. Using cgroups and namespaces, containers are separated from each other and can be given limited resources each.

In current work the container is used as one of the components to which we apply the update approach. So for that case we use the unconstrained type of device, supporting Linux [BEK14].

2.3 IoT Devices

There exist many devices that are suitable for IoT. Most of them are very diverse in characteristics, such as CPU power, volatile memory (e.g. RAM) or persistent storage (e.g. ROM, Flash memory) sizes and sensors or actuators connected. Generally these devices are separated in two groups according to their performance characteristics: constrained and unconstrained.

In [BEK14] three classes of constrained devices are described. According to Flash memory and RAM they are grouped into classes C0, C1 and C2 (see Table 2.1).

Device Class	RAM	Flash/ROM
C0	« 10 KiB	« 100 KiB
C1	10 KiB	100 KiB
C2	50 KiB	250 KiB

Table 2.1: Classes of Constrained Devices (KiB = 1024 bytes) (based on [BEK14])

Class C0 describes severely restricted devices, which cannot directly and securely communicate with Internet, and they rarely receive configuration or software updates. In Class C1 devices with which are less limited, but still cannot implement full protocol stack for over Internet communication. C2 describes devices with few limitations and which can use full communication protocol stacks as normal computers, with higher amounts of memory and CPU resources. But it is also possible that all that classes of devices can be shifted in forth with values of characteristics, because those in the Table 2.1 may be a bit old, due to the time passed since the [BEK14] was published. Constrained devices have many times higher characteristics than unconstrained ones.

In [KH14] there are described two types of IoT devices: Gateway and Sensor node. In Gateway all peripherals, RAM, Flash memory and the processor are implemented separately from each other and communicate via buses. So different combinations of hardware components are possible, varying in performance. In contrast to the Gateway, Sensor node has the processor, RAM and flash memory packaged into single device, with possibility of only radio, actuator or sensor selection. That means the unconstrained Device can be either the Sensor node or has Gateway architecture. For our work

architecture does not have any influence, since our approach is device architecture independent, so we will use unconstrained Gateway device for implementation part. The Implementation for constrained devices will be left for future works.

2.4 IoT Device Management

In IoT networks there are plenty of devices which have to be managed. Management of IoT devices is very challenging task, because of variety of heterogeneous communication protocols, different types of data generated, mobility of devices, diverse network topologies used, necessity for registration of new Devices and enabling of access control for end users. According to [All16] Device Management can be defined as setting of initial configuration on devices, subsequent updating of persistent information on devices, getting of management information from devices, execution of primitives on devices, processing of events and alarms generated by devices. For such case there are different protocols for Device Management (*DM* shortening will be used in *italics*, because in Chapter 4 another DM (Device Manager) shortening is used) used. Some major of them are:

- TR-069 (Technical Report 069) [BDW14] - is an application level protocol mostly for management and automatic configuration of Internet access devices, e.g. modems, routers, gate ways or VoIP-phones, by Auto configuration Servers (ACS).
- OMA DM (Open Mobile Alliance Device Management) [All16] - ia another application level *DM* protocol designed for management of mobile phones, PDAs and tablet computers.
- LwM2M - defined in subsection 2.1.3 of this work, previously.

2.5 Dynamic Software Update

All the software components and applications are not perfect and can have errors and bugs, which must be fixed as soon as possible. Some components are more demanding in terms of safety and time to the process of delivery and applying of update or fix. Non-stop systems such as air-traffic control, financial transaction processors, huge enterprise applications , and networks must provide service without interruptions while bug-fixing or updating. For such cases variety of Dynamic Software Update (DSU) techniques were developed. DSU is a process of software component or application update, without their execution interruption as defined in [SHB+07]. According to [Mak09], a typical DSU process includes several steps:

1. Stopping of the component old version in a given state.
2. Applying a *state mapping* function to the given state to obtain new state.
3. Continuing execution of the new component version with new state from previous step initialised.

In [MPJ18] there are three aspects of DSU described: *code transformation*, *state transformation* and the *update point*. *Update point* is an execution moment when update process occurs. *Code transformation* is an actual code substitution and *state transformation* refers to *state mapping* in second step of DSU process in [Mak09].

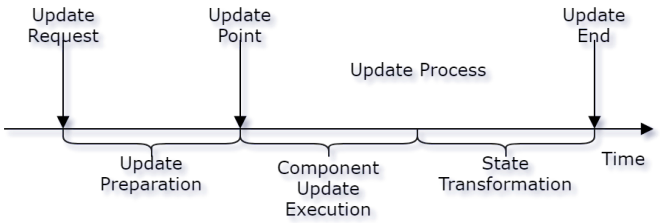

Figure 2.5: Time-line of Dynamic Software Updating Process (based on [MPJ18; SAM12])

From [SAM12], some missing description of time preceding *update point* was taken, and for simplicity renamed from *Update Bringing Forward Mechanism* to *Update Preparation*. Using information and time-line descriptions from [MPJ18; SAM12], time-line of DSU was drawn. In Fig. 2.5 DSU time-line is pictured, where all steps are shown. In Scope of our work we refer *Update Preparation* to several processes, such as acquiring of Update Package, and if necessary State saving, which will be described in Chapter 4. *Component Update Execution* corresponds to term *Code transformation* mentioned earlier in this section. *Update Request* corresponds to update initiation point, which means, for example user pressing update button in remote terminal GUI (Graphical User Interface).

According to exhaustive categorization given in [SAM12], there are huge amount of techniques dealing with different issues of DSU exist. Describing them all in details here would be very redundant task and is out of the scope of the thesis, so only the short description most important ones for thesis will be given:

- *Type-safety* - means that each component in the program has type which its dependent components expect from it. For example, String data type must not be interpreted as integer. That problem may arise if interfaces of components are not updated atomically. We assume that all dependant components are upgraded simultaneously.
- *Time of update* defines time when upgrade package will be applied to components, and is important issue to solve. There several possibilities to deal with that problem: waiting for component to enter special ”calm” state called quiescence [KM90] or tranquillity [VEBD07]; set *Update points* for component manually [HSHF12]; bring component forward to update causing service disruption [SAM12]. We will reason for that in Concept Chapter 4.
- *State transformation* is a process of a state transfer from an old version of component to its new version. Without the state transformation process, any DSU system becomes just a stop/start update service. In details that process will be described later in Concept Chapter 4.

In the list above, the tranquility term was mentioned. The component is in tranquility state if it is:

- not involved into transaction it has started;
- not going to start any new transaction;

- not processing any request at the moment;
- not engaged in a transaction the component and its dependants has already participated in and might participate later.

Important to define, what does the state in our approach description means. A computer program stores data in variables, which represent storage locations in the computer memory. The contents of these memory locations, at any given point in the program's execution, is called the program's state [Lap00; Mis01; Pra04]. An application's state is roughly the entire contents of its memory, e.g. Erlang's[12] server loops, which explicitly pass all the state of the application in a variable from one invocation of the function to the next. In other programming languages, the "state" of the program is all its global variables, static variables, objects on the heap and on the stack, registers, open file descriptors and offsets, open network sockets and associated kernel buffers, and etc. We can actually save that state and resume execution of the process elsewhere.

[12] https://www.erlang.org/

3 Related Work

In this chapter, works related to the area of the thesis will be presented. First works describing the IoT and its concepts are presented. Then we describe some works about Container technology. Consequently IoT devices are described and classified. Last but not least IoT related protocols specifications used in this work are discussed. In the end DSU contributors are listed.

Internet of Things

A lot of definitions of the IoT were presented in different survey papers, we used those given by Atzori et al. [ALM10], where except definition usage areas and actual challenges discussed, and by Agrawal and Vieira [AV13], where enabling technologies and protocols explained. Sensor networks definition given in [RM04], as well as their characteristics, classification according to these characteristics and wide range of application fields.

For Reference Architecture two papers were taken as the best relating to topic. In Guth et al. [GBF+16], they present abstract generic IoT architecture and proving it, by making a thorough comparison of several IoT solutions' architectures and pointing out similarities and differences. As a result their abstract IoT architecture embodies all three compared ones. Also descriptions of architecture's components with examples are given.
In [RBF+16] Reinfurt et al. they, like Guth et al., describe core components, but with some more inner details concerning inner processes, and show possible dependencies between components.

In Device Twin subsection, as a base, pattern description by Reinfurt et al. [RBF+16] was taken. They state the problem of demand for asynchronous communication with intermittently available Device and present Device Shadow concept and architecture of the system using it as solution with possible advantages and drawbacks. Also implementations of the Device Shadow concept are presented in commercial products of Amazon [Ama18] and Microsoft [Mica]. Both companies use JSON files of limited size mapped to each registered IoT device, as Device Shadows or Device Twin in Microsoft notation.

Container Technology

In its article Vaughan-Nichols [Vau06] touches topic of virtualization and its importance for IT Industry. He points out new possible lightweight approaches in virtualization such as Containers and Paravirtualization, discusses about tools for their management, and lists their pros and cons. Morabito et al. [MKK15] in their comparison paper present detailed comparison of Containers - lightweight virtualization technology versus traditional hypervisor-based solutions. They show that Container technology achieves better performance than the traditional virtual machines, and has relatively small performance difference with other technologies. From the work of Bui [Bui15]

the architecture of Container-based Virtualization technology was taken. In his work Bui analyses a security of Docker Container technology. In [Bra18] and [SAG+] there are plenty of technical details on LXC and Docker technologies are given by Git users. In [HJD+17] documentation on cgroups is given, describing how resources in LXC can be limited.

IoT Devices

Terminology for Constrained-Node Networks by Bormann et al. [BEK14] presents the definition of constrained devices and their classification according to characteristics, setting these devices apart from from unconstrained devices, e.g. laptops, servers, smartphones. From the work of Kruger and Hancke [KH14] we get another level of differentiation of devices on Sensor Network Node and Gateway device, according to their architectural designs.

Protocols

This OASIS MQTT Standard [OAS15] gives description of MQTT protocol capabilities, packet structure, architecture, application cases and security measures.
In Internet Engineering Task Force (IETF) standard [SHB14] Shelby et al. describe Constrained Application Protocol, its requirements, application field and capabilities.
LwM2M Specification [All18b] by Open Mobile Alliance, presents architecture and resource model for M2M Device Management enabling.

Dynamic Software Update

Seifzadeh et al. in [SAM12] provide review of many research works on DSU and develop a framework for DSU techniques evaluation, what helped to facilitate our research in DSU. In [MPJ18], Mugarza et al. make analysis of existing DSU techniques applicable for safe and secure industrial control systems. In its dissertation [Mak09] Makris presents whole-program DSU mechanism, which executes updates atomically and with bounded delay, necessary for multi-threaded applications. Another new DSU technique, for Java programming language, is described in [ORH02]. It is based on the use of proxy classes and needs no support from runtime system, and allows updating Java programs in runtime, by substituting, adding and deleting of classes. The proxy classes give us a level of indirection necessary to mask update inconsistencies. We adapt that idea but on a higher level of IoT application, by mixing that concept with the Device Twin IoT pattern.

In [SHB+07] Stoyle et al. present Proteus - a calculus to model upgrades in imperative programs, by balancing the concerns of flexibility and safety, adding assurances of predictability. In [HSHF12] Hayden et al. discuss problem of hard reachability of a full quiescence state in multi-threaded programs, which can be solved by manual annotation of a small number of update points.

In their work, Kramer and Magee [KM90] give description of quiescence – concept of a component idle state, when this component can be safely updated during DSU. That quiescence concept is used by Vandewoude et al. [VEBD07] as base for introduction of weaker, but less disruptive to DSU process, concept of tranquility. But in IoT System with multiple of components it is nearly

impossible to reach such a state due to high number of dependencies between components, which ruin conditions of tranquility. Therefore we forcefully put components to be updated into a passive state, without necessity of waiting its tranquility.

Weißbach et al. [WTW+16] present a DSU mechanism for IoT systems which coordinates the upgrade process of multiple distributed nodes in a running service. Their solution was developed for a decentralized network of devices, where update process is conducted in parallel on all devices. Their main problem was the simultaneous activation of all updated components on the devices, so the inconsistencies due to non-simultaneous updates are avoided. They solved that problem by implementing decentralized middleware layer, which coordinates software updates on distributed devices transitionally, so the consistent transition of the application through the update process is ensured. To assist that, the LyRT [ngu16] runtime, implementing a mechanism of dynamic instance binding is assumed to be installed on all these devices.

4 Concept

In this chapter we will describe the Concept of our method to enable component updating in IoT System. In the first Section 4.1 the Concept's idea is explained. Second Section 4.2 describes the architecture supporting our method. The leftover Sections 4.3, 4.4, 4.5 and 4.6 describe each of the four steps of the Concept correspondingly.

4.1 Overview

In this Section we will discuss the DSU techniques and the Device Twin pattern in a wireless IoT nodes' update process. Also, short preview of the update method Concept is given.

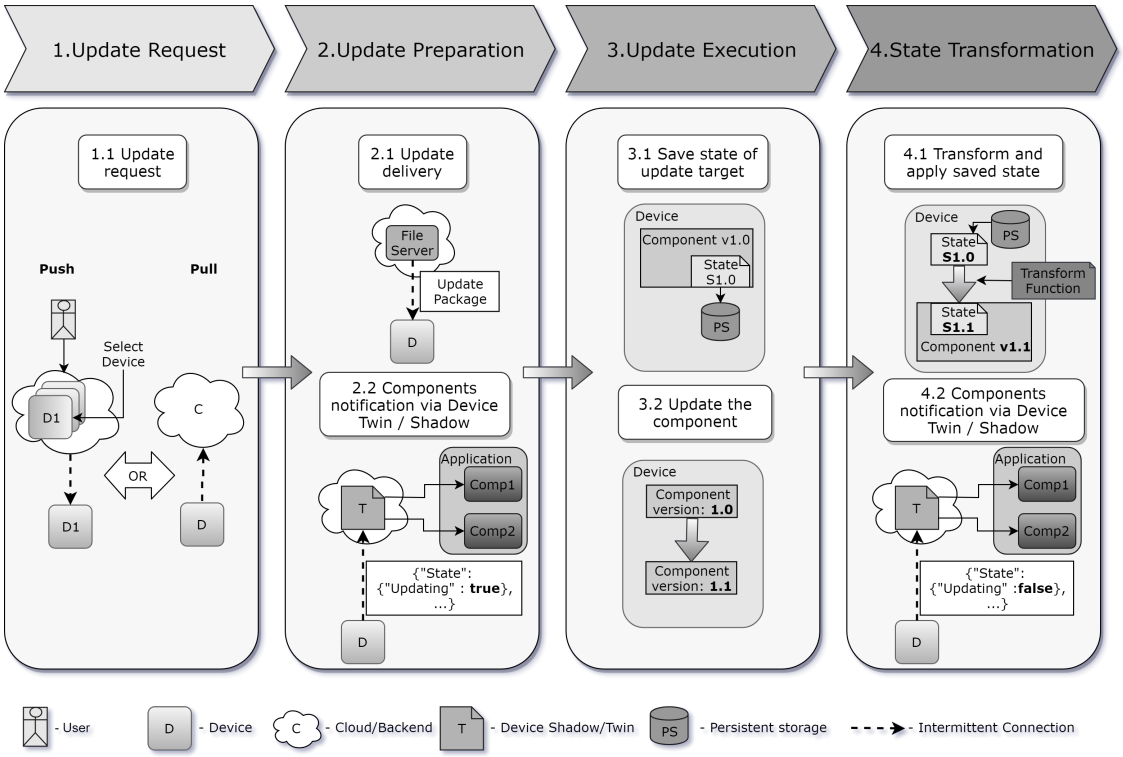

Figure 4.1: Update Method Concept general view

The concept main idea is based on the integration of the several Dynamic Software Update (DSU) (see Section 2.5) mechanisms, with the IoT patterns. If to look at an IoT system as on a big application or a program, one can see that end devices, gateways, intermediate middleware and

other applications – are the components. When any component need to be updated its dependants may still need the access to the updating component's resources. Therefore, the applying of the DSU method to the IoT system may not be feasible, when certain design decisions are selected, e.g. an update of several IoT components, which can span over a global set of components in different locations, may block the execution of entire applications. In the same case, in a DSU system, some level of indirection to all accesses in the system is added. Such a DSU technique can be implemented in the IoT system by the usage of the *Device Twin* concept, which is intended for the decoupling of applications from the concrete devices (see Subsection 2.1.2).

Another feature of this Update method is a component state management during the update process. Our method regards components of two types: stateless and stateful, each is handled differently at the update time:

- In case of the stateful component, the Update method handles the state transfer (migration), before and after the rewriting of the stateful component. The stateful component is the component, in which all its functionality works in the context of its state. To handle the state there are method sets, each consisting of sub methods "save state", "transform state" (using the appropriate transform function delivered with the Update Package, see Section 4.3) and "apply state". For each stateful component, e.g. an application, a container of a container-based virtualization system or a firmware, there is its own set of the sub methods implementations. In the following list these sub methods are explained:
  - *Save state* sub method saves the current state of the component to be updated in a persistent storage. Is used in the Update Execution step.
  - *Transform state* sub method takes a copy of the state from the persistent storage. Then, this sub method applies the transformation artifact from the update package to the state. It is performed in the State Transformation step.
  - *Apply state* sub method initializes the updated component with the state changed with the transform state sub method. It is used in the State Transformation step.

  From a device to device an implementation of the sub methods can be different, but the sub methods' logic will stay the same. Implementation specific details are given in the Chapter 5.
- For a stateless component, the update process consists only of a stopping of the component, rewriting it with a new version and starting it up.

Important point is – when to apply the Update procedure to the component. In the DSU several mechanisms exist (see Section 2.5 and 3): a system developer can manually define the Update points, when the Update process must occur; the Update process can wait for the special quiescent state, which may never happen in an IoT system, because several IoT applications may use the same components; or the component can be forcefully brought forward to the Update process start.

In the Fig. 4.1 the four parts of the Update method Concept are shown. Below these steps are enlisted with short teasers.

1. Update Request – is the first step of the Update method, when the initiation of update is done. Also, we select the way how it is done: Manually by user (Push) or Periodically by device (Pull).

2. In Update Preparation step the Update Package is delivered to the end device, there the necessary component must be updated. Also, details of delivery described. At the end of the step, dependent components are notified.
3. Update Execution step describes, what happens with the state of component, if it is stateful. In stateless component case we do nothing at that point. Then, component is stopped and renewed.
4. In the last State Transformation step, we transform the state of the renewed component, if necessary, and apply it. In stateless case, step is omitted. Consequently, component is started and dependent application components are notified.

4.2 System Architecture

In this section, an abstract system architecture based on IoT Reference Architecture from Subsection 2.1.1 is overviewed and explained with Update method steps.

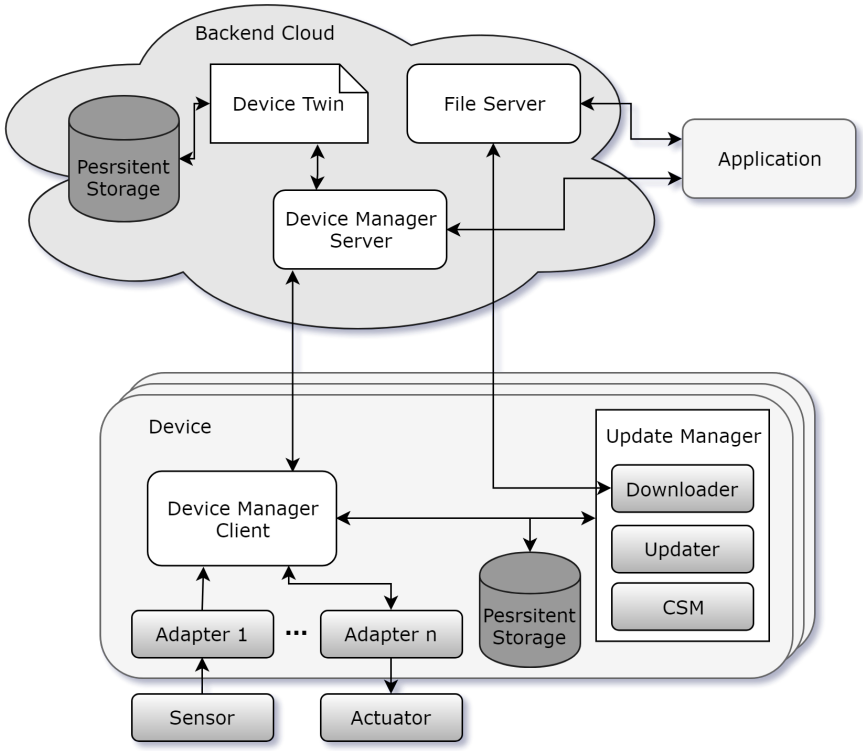

Figure 4.2: System Architecture view

System abstract Architecture can be seen in the Fig. 4.2. Here, the *File Server* is a simple repository, where necessary for the update process files are stored. *File Server* has an interface for an authorized package upload on it and an interface for answering to download requests from devices and from

the application. For example, it can be a simple file server deployed on any Cloud service, which manages HTTP or CoAP GET requests and answers with a file transfer back to the requester component.

The *Device Twin* stores a virtual representation of an IoT end device connected to our system, e.g. a JSON-document. Last reported states from a device are kept in there. Also, commands to the device can be put in the *Device Twin*. More detailed description of the *Device Twin* is given in the Subsection 2.1.2.

Update Manager is a component consisting of *Downloader*, *Updater* and *Component State Manager*. *Downloader* must implement functions of the receiving and storing files into the *Persistent Storage*. Updater is a component which replaces component to update with its newer version in the Update Execution step of the Update method. The *Updater* is responsible for the component rewriting in the Update Execution step. The *Component State Manager* (CSM) is used for state management actions in the Update Execution and the State Transformation steps of the Update method. In the Update Execution, the *Component State Manager* takes the state from a stateful component and saves it in the *Persistent Storage*, in case if the component does not have its own mechanism for the state export and the subsequent saving of the state in the *Persistent Storage*. In the State Transformation step, the *Component State Manager* takes an appropriate state of a stateful component out of the *Persistent Storage*, transforms it using the state transformation artifact, if latter is supplied, and applies state to the corresponding component.

Device Manager Client and *Device Manager Server* are components incorporating communication and device components' management functions. These entities are not showed in the Update method's Concept Figure and its steps' Figures. They are described in the Chapter 5 and referred as Device Manager LwM2M Client and Device Manager LwM2M Server correspondingly.

4.3 Update Request

In this Section two possible ways of the initiation of the update process Push and Pull (see the Fig. 4.3) are described. In the first part of the Fig. 4.3 (see step Prerequisite in Fig. 4.3) reader can see the *File Server* and *Update Package*. The *Update Package* here is a composition of files necessary for the update process. In the Fig. 4.3 one can see that the *Update Package* consists of the State Transformation Artifact and the Component Update Artifact. The first Artifact is needed for the State Transformation step, if stateful device is updated, and is explained later in the Section 4.6. The second Artifact – Component Update Artifact includes the new version of the component. So here we assume that the user uploads the *Update Package* onto the *File Server*, and we do not consider the case of an empty *File Server*.

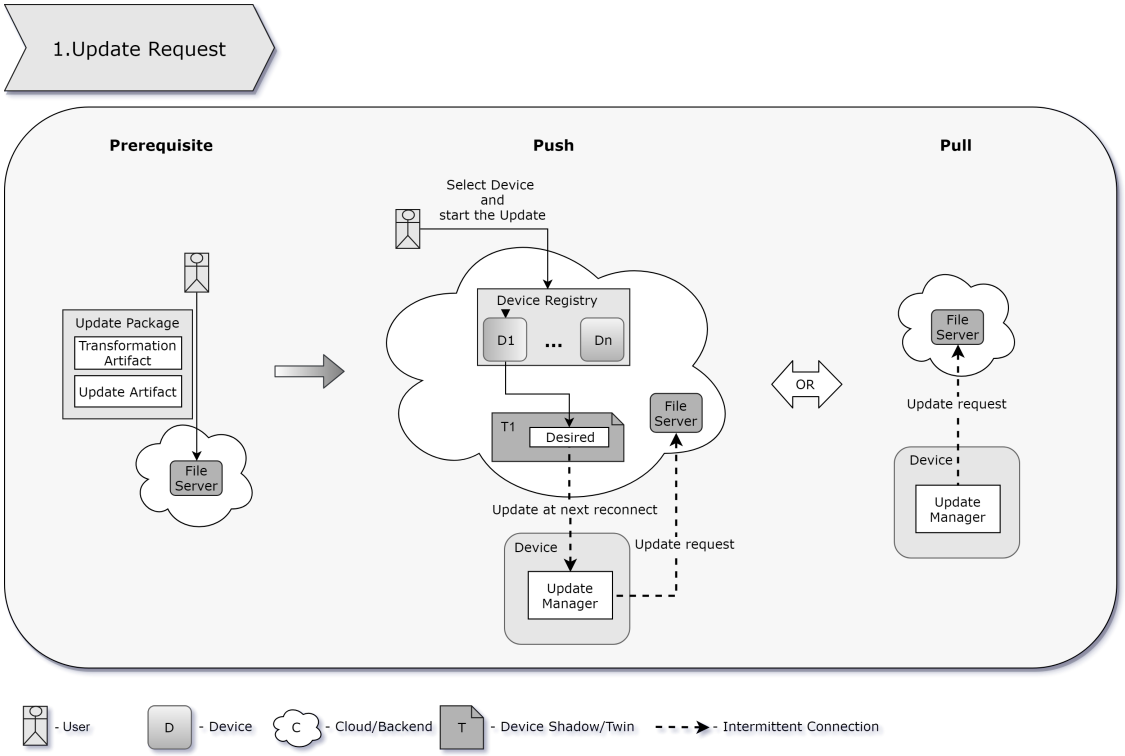

Figure 4.3: Update Method Concept: Update Request view

Right after the Prerequisite part is accomplished, the Update Request step continues with one of the following two ways:

- First way, called Push can be seen in Push part in Fig. 4.3. There the user selects the device in the *Device Registry*, e.g. device D1 in the Fig. 4.3, and initiates the update process by sending appropriate commands. Selection may be done via various ways: the user knows the command to start the update and the name of the device, then sends that data to the *Device Registry*; or the user uses the UI (User Interface) for the communication with the *Device Registry*, selects the device or multiple devices for the update and launches the update command. After that, the *Device Twin* corresponding to the device identity from the *Device Registry* is selected. The update command is put into the *Device Twin* for the execution by the device connected with that *Device Twin*. The update command may be an object consisting of the device component's name and the instruction to execute. As long as communication between the device and the *Device Twin* is not stable, the device gets the commands from the *Device Twin* upon the next successful communication session. When that happens, the *Update Manager* sends the request (e.g. HTTP or CoAP GET request) for the appropriate *Update Package* to the *File Server*.
- Pull (see part Pull in Fig. 4.3) is another way to initiate the update process, but from the device's perspective. The device must have a schedule, where the update check times are prescribed. That schedule must be set by the user, e.g. via GUI in back end application. The only difference is that instead of the instant update command, the scheduled update check command is sent. After the *Update Manager* of the device received that instruction, it begins

sending check request directly to the *File Server*. The check request includes a name of the device's component to be updated and its current version number. The *File Server* receives the check request, then compares the current version number received with those of the relevant component on the *File Server*, and if latter one is bigger than the former one, download starts.

At the end of Pull or Push way, the Update Request step converges into the point of starting download the *Update Package*. The next actions are described in the following Update Preparation step.

4.4 Update Preparation

In this section the Update Preparation step is described, giving the details of steps preceding the Update Execution step. You can see it in Fig. 4.4.

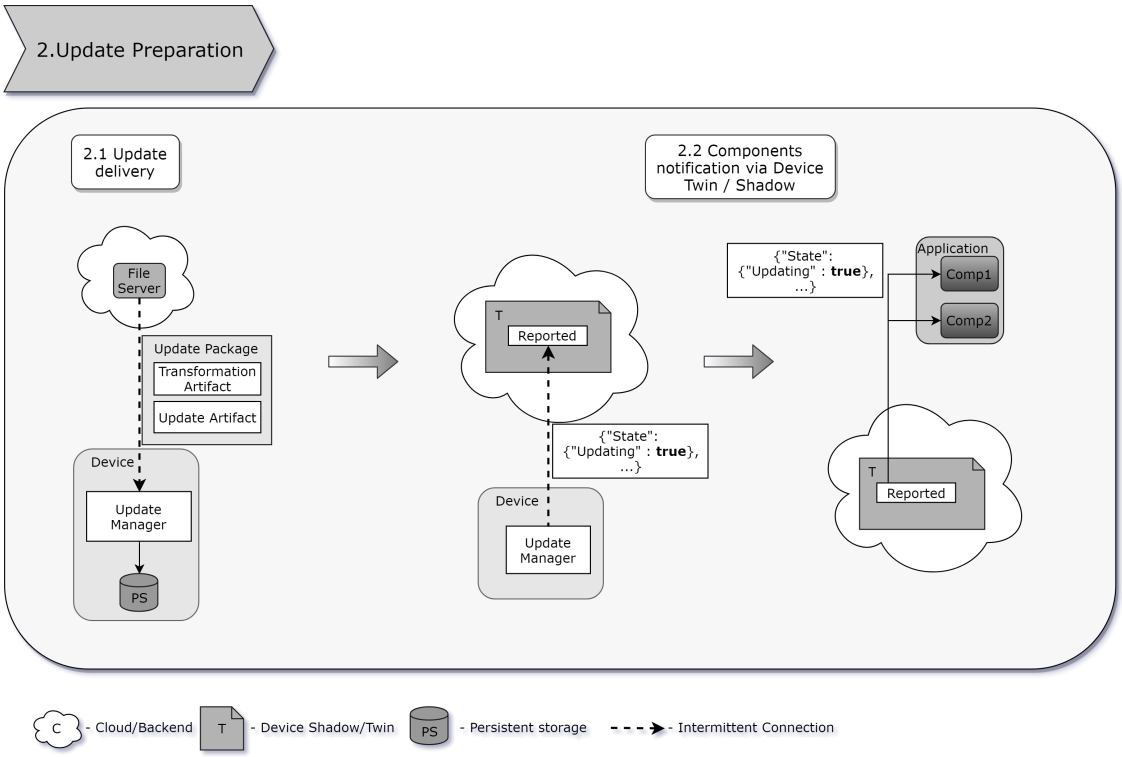

Figure 4.4: Update Method Concept: Update Preparation view

When the *File Server* receives a request for a component *Update Package* download from the device, it initiates a transfer of the *Update Package* to the device *Update Manager* (see most left part in Fig. 4.4). Then, the *Update Package* is saved to the *Persistent Storage* until it is needed in the next, Update Execution step.

In the next action (see middle part in Fig. 4.4) an inaccessibility of the device component is masked with the help of the *Device Twin*. That moment is essential, if back end application's component is dependent on the data from the device component, which going to update. So the back end application component can distinguish a simple communication error from an update process. To

achieve that the device sends status message containing an information, that the device entered the update mode, to the *Device Twin* and will be unavailable for some amount of time. That ”unavailability” time, if needed can be derived from summing of an approximate time, necessary for a concrete update process, plus time, when the device is inaccessible due to environment conditions, and can be put into the status message. All that information including the unavailability time and status of device can be used by the system to plan its actions regarding the device appropriately or even to try finding other normally functioning sevice. Also, the system will be informed, that the data in the *Device Twin* is not synchronized and should be used with caution. Upon the next synchronization of the *Device Twin* and the application (see most right part in Fig. 4.4), the *Device Twin* sends to the application all gathered in the previous sub step information about the update process.

4.5 Update Execution

When the previous Preparation step is finished, the main part of the Update method starts – an Update process execution (see Fig. 4.5).

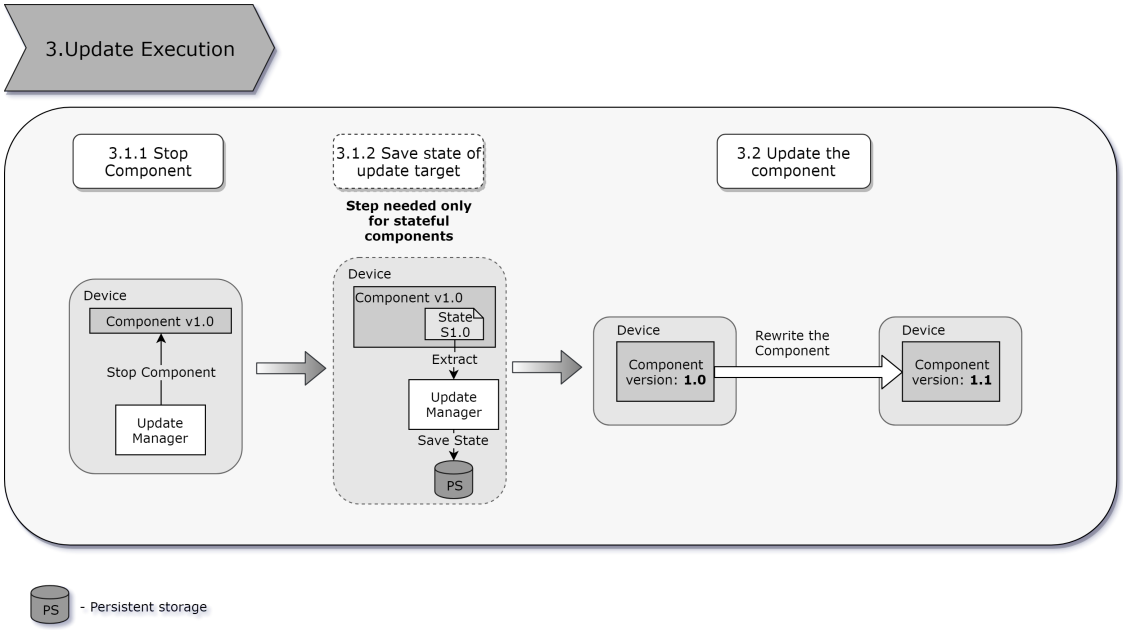

Figure 4.5: Update Method Concept: Update Execution view

Before rewriting the component it must be stopped, but if it is involved in transaction process with other components stopping means possibly deadlock situations or errors for those dependant components (see the Stop Component part in Fig. 4.5). That means, the component to be updated must be stopped at the proper time, or to be in the so-called tranquillity state. According to the Dynamic Software Update description (see Section 2.5), the component is in the tranquillity state if it is:

- not involved into transaction it has started;

- not going to start any new transaction;
- not processing any request at the moment;
- not engaged in a transaction the component and its dependants has already participated in and might participate later.

For example, the sensor adapter component to update, sends received from sensor data each 5 seconds to other components. Assume not to consider the sensor as dependent component, as long as it just sends data independently if it is accepted or not. It is configured that the adapter starts transaction every 5 second to receive data from sensor and then retransfer it in appropriate format, and that moment breaks the tranquility state second condition, which says not to start any new transaction. In that case, *Update Manager* stops the subscription of any dependant component to that sensor data right after last retransfer from adapter and then will stop the adapter forcefully and will notify any dependent component on device of its stopped state. If component successfully stopped, next part of the step is launched in case of stateful component, otherwise we omit this step. In stateful component's case, the Component State Manager of the *Update Manager* extracts the state of the component or the component gives its state, and puts the state in the *Persistent Storage*. Right after previous preparation procedures are finished, *Update Manager*'s *Updater* rewrites the component with the new one.

4.6 State Transformation

In this final step of the Update method (see Fig. 4.6), the state management operations are finished, then the application is notified about the end of the update process.

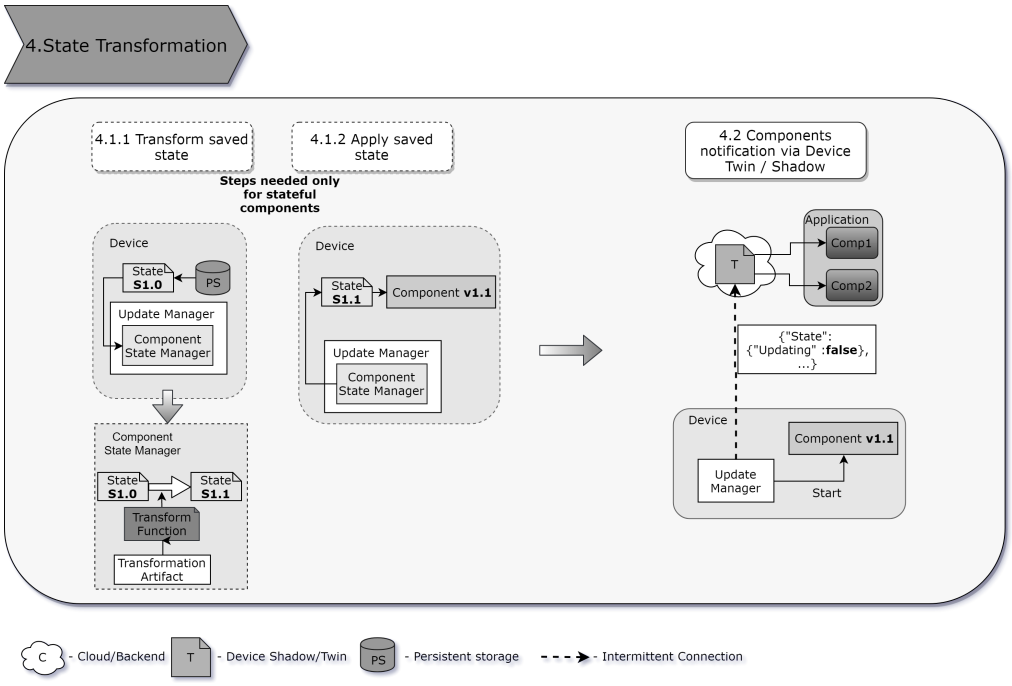

Figure 4.6: Update Method Concept: State Transformation view

As previously described, beginning of this step is different for stateful and stateless components. Clearly, stateful components require the ”Transform saved state” and ”Apply saved state” actions (see actions 4.1.1 and 4.1.2 in Fig. 4.6), while stateless components skip these steps right to the last action (see action 4.2 in Fig. 4.6). First, the Component State Manager component of the *Update Manager* copies the state of component from the *Persistent Storage* into its buffer. A copy of the state may be left in the *Persistent Storage*, so that the operation ca be repeated in the case of a fault or the device shutdown. Next, the Transform Function is applied to the state. Then, the Component starts, and the *Update Manager* sends a notification to the corresponding *Device Twin*. So the application receives the information that the component is available again.

5 Implementation and Validation

In this Chapter, we describe the implementation of the Update Method Concept presented in previous Chapter 4 and demonstrate its feasibility.

5.1 Implementation

In this Section each part of the Prototype system (see Fig. 5.1) is described in detail and its mapping to abstract Architecture parts is explained. Each Subsection corresponds to each part of Prototype.

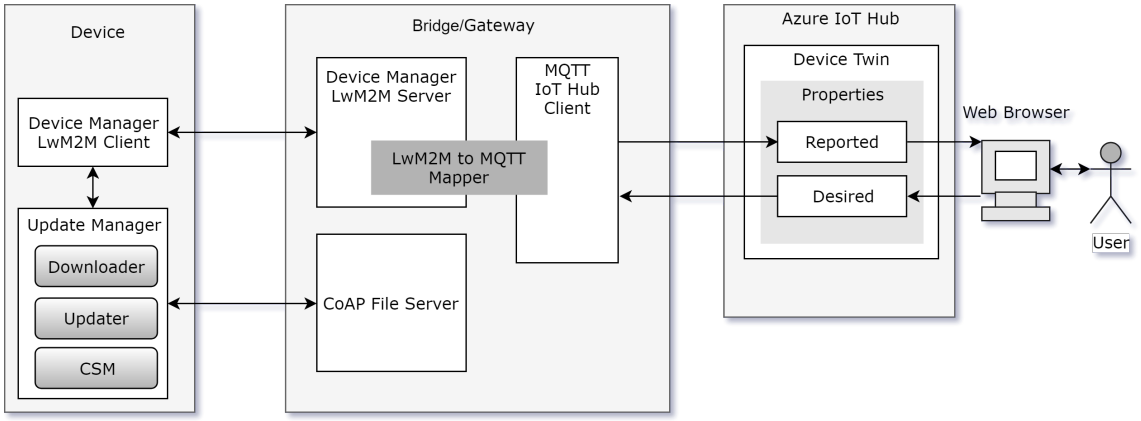

Figure 5.1: Prototype Architecture (CSM – Component State Manager) (based on [LH18; Per17])

5.1.1 Device

This Subsection describing Device part of prototype is divided into several parts. First one, called MICA, presents Device's hardware and software capabilities. Second part, called Device Manager, explains a logical structure of the Device Manager and its functions. Last part, describes functions and components of the Update Manager.

MICA

As a device, the Harting's Modular Industry Computing Architecture (MICA) was taken (see Figure 5.2). MICA is an industrial computer with an 1 GHz ARM processor, 1 Gb RAM and 4 Gb eMMC flash memory [HARb]. MICA also has an M12 Ethernet 10/100 Mbps port, a micro SD card

connector and 8 digital GPIO pins with 12V or 24V[HARb]. In USB version there also 2 USB Type A ports [HARb]. As a power supply a Power over Ethernet, a 12V power supply or a 24V power supply can be used. [HARb].

Figure 5.2: Modular Industry Computing Architecture: from left to right two USB ports, M12 port for GPIO and M12 port for Power over Ethernet.

Also, MICA fulfills the IP67 standard of protection and EN 301489, EN 60950, EN 50364, EN 50155 standards [HARb]. The MICA is split into a supply board with a power connection and a network connection, a processor board and a function board, communicating via with each other via USB [HARc]. For some MICA models, the function board is already pre-assigned, so the USB version uses the function board for the USB ports. The function board can also be determined by the customer. Also, in some versions of the MICA a different processor can be selected, and Bluetooth, LTE, WLAN, Modbus or RFID integrated [HAR18]. MICA firmware is based on Linux and all applications run in a virtualized Linux containers (see section 2.2), and offers a JSON-RPC (Remote Procedure Control) 2.0 based Single-Sign-On (SSO) interface [HARb; WR]. SSO allows services to identify users who run on containers and allows containers to be managed. There are plenty of functions to control container environment available, e.g. start, stop, install or delete containers. Also, Harting company provides different containers based on Debian Linux and BusyBox, for development of applications [HARa]. There are also specialised Java, Python or Node.js containers available [HARa], which are used for current implementation.

Device Manager

In the Device, LwM2M Client acts as a *Device Manager*, which manages the resources of device and controls the Update Manager. As mentioned in the end of Section 5.1.1, MICA can have specialized containers, e.g. Python, Node.js or Java, for different applications. In our prototype Java container for our application including the *Device Manager* and the *Update Manager* is used. The *Device Manager* is basically a Java implementation of the LwM2M client, based on a set of Java libraries of the Eclipse Leshan project [Git18]. The *Device Manager* implements LwM2M resource-object

model for an access and a control of resources and components on the End Device. There are three mandatory Objects have to be implemented in LwM2M Client: *Security*, *Server* and *Device*. All the implementation details of each are given in Leshan Project's documentation. For example purposes *Device* object will be shortly described. There is a model for a *Device* object, where number of resources inside may be defined [All18b]. There are several mandatory resources such as *Reboot*, *Error Code* and *Supported Binding and Modes*, and optional resources, e.g. *Battery Level*, *Timezone*, *Device Type*, etc. Mandatory resources must be always implemented, while optional ones are free to enable [All18b].

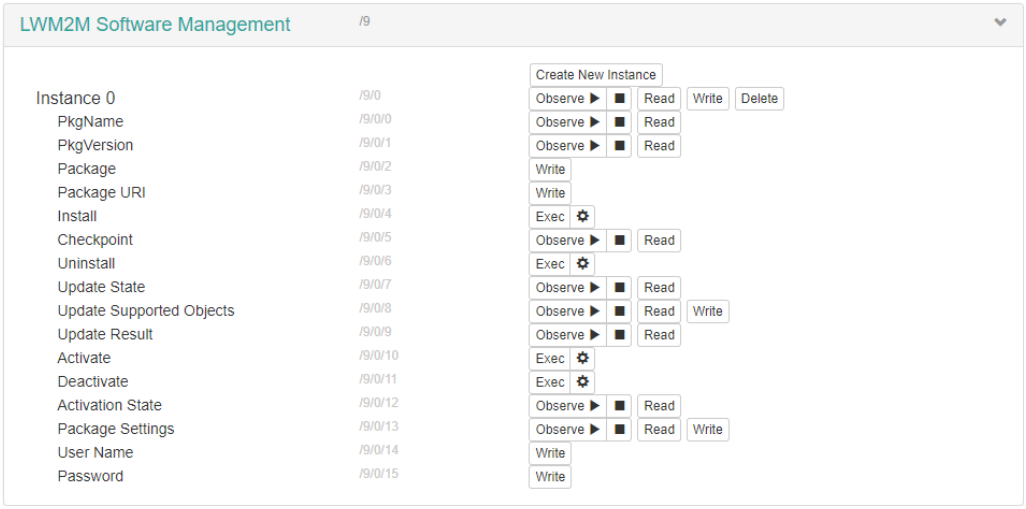

Figure 5.3: LwM2M Software Management object: web interface

As long as different software components are to be addressed and updated, a model for that case is needed. The standard OMA LwM2M object *LWM2M Software Management* model for Software Management [All18a] is used (see Fig. 5.3). It can be adapted to update containers, applications inside containers and Firmware. For latter, there is a separate object Firmware Update, which is recommended [All18b]. The resources of Software Update object are mapped for the container update. For example, mandatory executable resource *Activate* may be mapped to *Start container*-JSON RPC call to Base OS of MICA. Same mapping can be done for internal container applications. The type of component to update, e.g. container or application is derived from resource *PkgName* of object *LWM2M Software Management*. Therefore, appropriate names for *Update Packages* must be used. The following form of the *Update Package* is used: "*type.component_name.package_name.ext*", where *type* is the type of component, container or application, *component_name* is the name of component to update, e.g. Java8Container, *package_name* is the name used for a naming that *Update Package*, before applying our naming scheme, and *ext* is an extension of the *Update Package*. Dots "." in the *Update Package*'s name are needed as stopping elements for parsing by the *Update Manager*. So, the example name for Java container with name Java8 will have a following look – "container.Java8.java_8_container_v1_3_0_b.tar", where "java_8_container_v1_3_0_b.tar" is the *Update Package* name of a Java container update archive issued by the Harting Company. The incoming message with a request to write the *PkgName* resource "/9/0/0" (see Fig. 5.3) of *LWM2M Software Management* object will be "PUT /9/0/0 container.Java8.java_8_container_v1_3_0_b.tar", where "/9/0/0" is a resource definition in LwM2M notation as described in Section 2.1.3.

Update Manager

Whereas the *Device Manager* is a supporting part of the prototype system, the *Update Manager* is a core of it. Basically, the *Update Manager* – is a library, which provides its methods upon the calls from the *Device Manager*. The following components implement the *Update Manager*'s functionality:

- **Downloader** – is a component, which implements several methods necessary for the *Update Package* download from the *File Server*. Upon call, Downloader must receive an address of the *File Server* and a name of the *Update Package* in the form of a URI (Uniform Resource Identificator), e.g. "coap://localhost:5683/data.rar". After, the Downloader makes a CoAP GET-request to the *File Server* for a block-wise downloading of the appropriate *Update Package*. All possible communication interruptions during the download process are handled by the CoAP QoS . It must be mentioned that in a single CoAP communication channel, a message exchange is sequential. So, until the block-wise transfer is in process, next request to send must wait of the transfer end. If the block-wise transfer is interrupted, CoAP waits some interval and retries. It continues a certain number of times, with the interval to the next retransmission increasing each time. The number of retries and the interval can be set in a configuration file, called "californium.properties". It is a standard name for a configuration file used in Californium[1] – the Java CoAP implementation. After the last unsuccessful retry, the block-wise is cancelled. In case of a successful *Update Package* transmission, the *Downloader* must receive "2.04" message from the *File Server*, which indicates that the *File Server* has successfully fulfilled the request and that there is no content to send in the response payload body.
- **Updater** – is an interface, which provides methods for the rewriting of a component. The methods may be implemented in any language for any component. Necessary implementation selection is done via the Updater Factory. In our prototype we implemented three methods for the component rewriting. The method for a container update executes the necessary JSON RPC calls to the MICA Base OS via the Web Socket (WS) interface [HAR17]. Then the actual update process of a container including the necessary state management of the container and its rewriting is performed by the MICA Base OS. In case of the Base OS (Firmware) update, the steps as for the container update are performed, but different JSON RPC calls are issued by the appropriate method implementation. But, if the implementation for the updating of a software component located on any container of the MICA is called, then the sequence of procedures for a component rewriting is used.
- **Component State Manager** – is a part of the *Update Manager* called only in case of the stateful component update. In our system, only methods for the software component update are implemented, because, as was written in the *Updater* definition, the container and the firmware components' state management is done by MICA Base OS. When the call to update the stateful component arrives from the *Update Manager*, the *Component State Manager* initiates the "save state" method. It copies the state of the component into the *Persistent Storage* of the container, in which the *Update Manager* is located. Upon the "transform state" method call, the *Component State Manager* takes a copy of the state from the *Persistent*

[1] https://github.com/eclipse/californium

Storage and applies to the state a code from the *State Transformation Artifact* (see beginning of the Section 4.3). If there is no *State Transformation Artifact* supplied with *Update Package*, then nothing is applied and the ”apply state” method is called. The ”apply state” method takes the state from the ”transform state” method and initialises the renewed component with that state.

5.1.2 Bridge Gateway

Since the Microsoft Azure IoT Hub does not support the LwM2M protocol, so the need for a ”bridge” from the LwM2M protocol to the MQTT, AMQP or HTTPS protocols, has appeared. As long as, there was the sample LwM2M-to-MQTT bridge written, the decision to use it to spare a time was taken [Per17]. The bridge contains the LwM2M server and the MQTT IoT Hub client, which are implemented by [Per17] with a use of the Node.js[2]. Both, the client and the server are mapped to each other. The LwM2M Server communicates with the LwM2M Client located on the MICA device, while the MQTT IoT Hub Client communicates with the Azure IoT Hub using the IoT Hub connection string (see Fig. 5.1) [Per17]. The connection string – is a unique identifier, which maps a unique device with its identity in the IoT Hub’s identity registry. The identity registry stores information about the devices and modules permitted to connect to the IoT hub. Upon the initial LwM2M Client registration by the LwM2M Server, the bridge sends to the Client the commands from the desired *Device Twin* properties (see Listing 5.1).

```
...
"properties": {
  "desired": {
    "method": "write",
    "resourceUri": "/9/0/0",
    "newValue": "container.Java8.java_8_container_v1_3_0_b.tar",
    ...
  },
  ...
}
...
```

Listing 5.1: Device Twin’s Desired properties example

A *Device Twin* JSON-object is received from the IoT Hub right after the first communication session of the bridge with the IoT Hub, for the initialisation of the bridge. In the next sessions the bridge uses the *Device Twin*’s properties to report the state of the device to and receive commands from the back end application. For the reporting of information from the device to the application the ”reported” properties are used. For the transfer of commands in the opposite direction, the desired properties are used.

[2]https://nodejs.org/en/

5.1.3 Azure IoT Hub

As a back end solution for a message transfer and as a *Device Twin* storage, Azure IoT Hub was used [Micb]. It maintains a *Device Twins* for each device connected to the IoT Hub. Each *Twin* is connected to the device identity in the identity registry mentioned before.

As an interface to our system prototype, we used an Azure IoT Hub web page. It allows an addition and deletion of IoT devices. Each device upon creation is given a name and may have an automatically generated primary and secondary keys (see Fig. 5.4). Keys are symmetric shared access keys stored in base64 format. Also, connection strings based on keys are generated. Connection strings are used in API calls which allows device to communicate with IoT Hub.

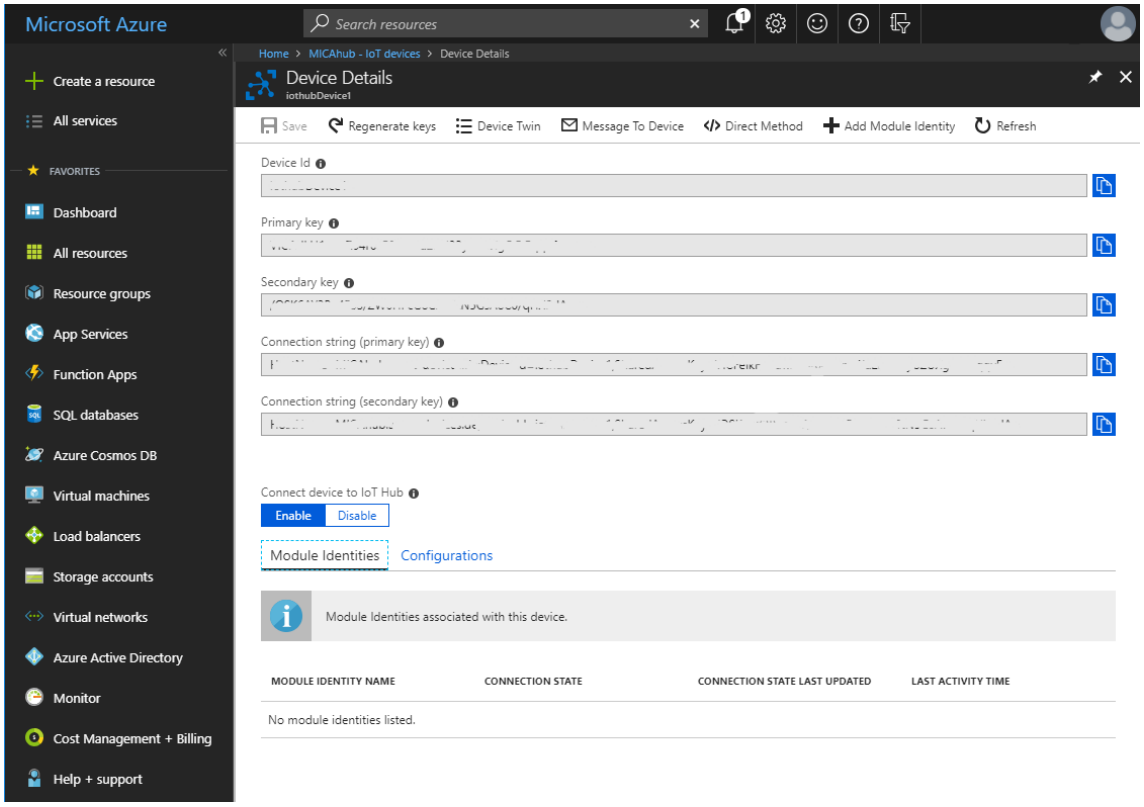

Figure 5.4: Azure IoT Hub Web Interface: Device Details

For the viewing and editing the *Device Twin* we use Device Twin partition of the Azure IoT Hub. In the Fig. 5.5 there is the input field, where desired properties can be set and reported ones viewed.

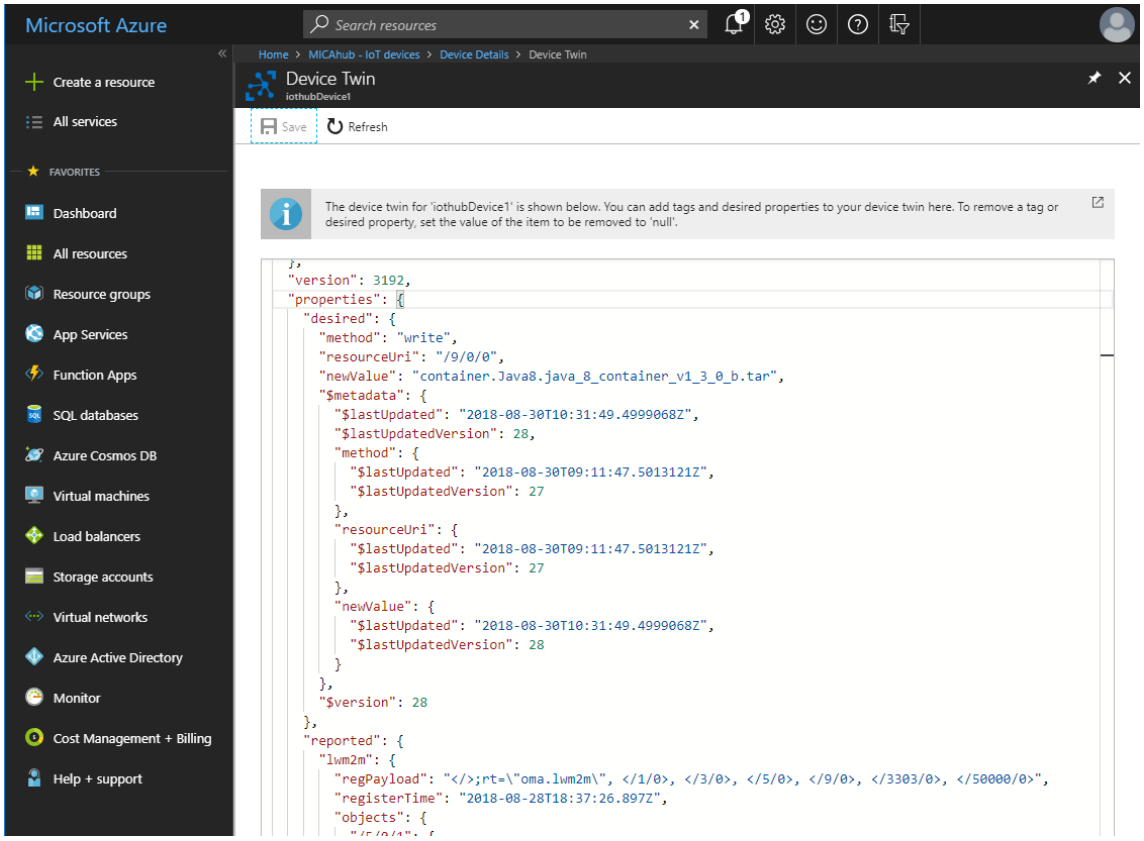

Figure 5.5: Azure IoT Hub Web Interface: Device Twin

In our use case it is the main interface to interact with the connected devices.

5.2 Validation

The use case for the validation of our prototype system is based on a component update scenario of a single MICA device. Using the web interface of the Azure IoT Hub web page, we request an update of the selected component on the MICA device. The update request is done in a form of the key-value pairs entered into the desired properties of the *Device Twin* JSON-object (see Listing 5.2).

```
"desired": {
  "update": {
    "type": "Push",
    "component": "container",
    "componentName": "Java8",
  }
...
}
```

Listing 5.2: Push update of container

The IoT Hub sends the update request to the bridge. On the bridge the *Device Twin* JSON object is parsed. The parsed properties of the twin are transformed into a LwM2M format and sent to the MICA device. The *Device Manager* on the MICA device uses received the data from the bridge, to initiate the update process of targeted component. The LwM2M Software Management object is used to keep the name of the update package and to parse the name to define what type of the component to update, the component's name, and the name of the update package to download. Then that object calls the *Update Manager* using the parsed data as arguments. The *Update Manager* initiates the Update Preparation step of the update process. In the current case, we used the Push update type (see Listing 5.2), so the *Downloader* downloads the update package mentioned in the update request from the *File Server*. Then, the *Update Manager* uses the LwM2M Software Management object's status resource to send the update status via the bridge to the *Device Twin* in the IoT Hub. At that moment the status will be seen in the *Twin*'s reported properties on the IoT Hub. After the notifying, the *Updater* of *Update Manager* makes JSON RPC calls to MICA Base OS to stop container, update it and start again. All the state management functions in between of the update process of a container are delegated to the MICA Base OS. Right after the *Updater* received a status message from the MICA Base OS about the update end, it notifies the *Device Twin* in the IoT Hub about the end of the update process.

6 Conclusion and future work

In this thesis, the main goal was to develop an approach for wireless IoT devices updating and implement it using any of existing IoT technologies. For that case works in Dynamic Software Updating area were studied and several techniques applicable in the IoT field are taken as a base for our approach. Also, several papers about existing IoT patterns were researched, from which we took a reference architecture and Device Twin concept to design our IoT system architecture for updates delivery and management. As a result the approach for the IoT system update management combining techniques and patterns from the researched papers was developed and the system realizing that approach was implemented. For implementation were used: Microsoft Azure IoT services, Harting MICA device and implementations of LwM2M, MQTT and CoAP protocols. The system has shown successful update of container components during its functioning, hence proving the applicability of the developed approach for unconstrained IoT devices.

Future work

In future work, the approach applicability for constrained devices can be checked by implementing a system in IoT network with constrained devices. Also, the update coordination system can be developed to allow automatic centralized update of spans of multiple devices in IoT system simultaneously, without causing system instability and inconsistency

Bibliography

[All16] O. M. Alliance. *OMA Device Management Protocol*. Tech. rep. Open Mobile Alliance, Feb. 9, 2016 (cit. on p. 24).

[All18a] O. M. Alliance. *Lightweight M2M – Software management Object (LwM2M Object – SwMgmt)*. Version 1.0. Mar. 1, 2018. URL: http://www.openmobilealliance.org/release/LWM2M_SWMGMT/V1_0-20180301-A/OMA-TS-LWM2M_SwMgmt-V1_0-20180301-A.pdf (cit. on p. 43).

[All18b] O. M. Alliance. "Lightweight Machine to Machine Technical Specification: Core". In: (Aug. 6, 2018). URL: http://www.openmobilealliance.org/release/LightweightM2M/V1_1-20180710-A/PDF-version/OMA-TS-LightweightM2M_Core-V1_1-20180710-A.pdf (cit. on pp. 21, 28, 43).

[ALM10] L. Atzori, A. Lera, G. Morabito. "The Internet of Things: A survey". In: *Computer Networks* 54.15 (Feb. 2010), pp. 2787–2805. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2010.05.010.. URL: http://www.sciencedirect.com/science/article/pii/S1389128610001568 (cit. on pp. 13, 15, 16, 27).

[Ama18] Amazon. *AWS IoT Developer Guide*. 2018. URL: https://docs.aws.amazon.com/iot/latest/developerguide/iot-dg.pdf (cit. on pp. 18, 27).

[AV13] S. Agrawal, D. Vieira. "A survey on Internet of Things - DOI 10.5752/P.2316-9451.2013v1n2p78". In: *Abakós* 1.2 (May 2013). DOI: 10.5752/p.2316-9451.2013v1n2p78 (cit. on p. 27).

[BDW14] J. Blackford, M. Digdon, J. Walls. *TR-069 CPE WAN Management Protocol*. Tech. rep. The Broadband Forum, Jan. 8, 2014. URL: http://www.broadband-forum.org/technical/download/TR-069_Amendment-5.pdf (cit. on p. 24).

[BEK14] C. Bormann, M. Ersue, A. Keranen. *Terminology for Constrained-Node Networks*. Tech. rep. May 2014. DOI: 10.17487/rfc7228. URL: https://tools.ietf.org/html/rfc7228 (cit. on pp. 15, 23, 28).

[Bra18] C. Brauner. *LXC*. 2018. URL: https://github.com/lxc/lxc (cit. on pp. 23, 28).

[Bui15] T. Bui. "Analysis of Docker Security". In: *ArXiv e-prints* (Jan. 2015). arXiv: 1501.02967 [cs.CR] (cit. on pp. 22, 27).

[DG17] B. C. Devices, S. GmbH. *Cross Domain Development Kit | XDK*. Nov. 24, 2017. URL: https://xdk.bosch-connectivity.com/documents/37728/87798/XDK_Node_110_combined_Datasheet.pdf/9e8a29c9-08bc-4232-a7c7-1d42cd55239e (cit. on p. 21).

[Fou18] R. P. Foundation. *COMPUTE MODULE DATASHEET*. 2018. URL: https://www.raspberrypi.org/documentation/hardware/computemodule/datasheets/rpi_DATA_CM_2p0.pdf (cit. on p. 21).

[GBF+16] J. Guth, U. Breitenbücher, M. Falkenthal, F. Leymann, L. Reinfurt. "Comparison of IoT platform architectures: A field study based on a reference architecture". In: *2016 Cloudification of the Internet of Things (CIoT)*. IEEE, Nov. 2016. DOI: 10.1109/ciot.2016.7872918 (cit. on pp. 13, 16, 17, 27).

[GG18] mpasternacki (Github-User), lenada (Github-User). *Jetpack*. 2018. URL: https://github.com/3ofcoins/jetpack (cit. on p. 22).

[Git18] sbernard31 (Github-User) et al. *Eclipse Leshan is an OMA Lightweight M2M (LWM2M) implementation in Java*. 2018. URL: https://github.com/eclipse/leshan (cit. on p. 42).

[Gmb] H. D. GmbH. *Was digitalisiert die Fabrik? SPS, Raspberry Pi und MICA® im Vergleich*. URL: https://www.harting-mica.com/sites/default/files/2017-08/White_Paper_SPS_Raspberry_Pi_MICA_im_Vergleich.pdf (cit. on p. 21).

[HARa] HARTING IT Software Development GmbH & Co. KG. *FIND YOUR MICA CONTAINERS!* URL: http://mica-container.com/ (cit. on p. 42).

[HARb] HARTING IT Software Development GmbH & Co. KG. *HARTING IIC MICA Data sheet* (cit. on pp. 41, 42).

[HARc] HARTING IT Software Development GmbH & Co. KG. *HARTING IIC MICA The Integrated Industry Platform* (cit. on p. 42).

[HAR17] HARTING IT Software Development GmbH & Co. KG. *HARTING HAIIC MICA Programming Guide*. 3rd ed. Version v 1.5.0. 2017. URL: https://www.harting-mica.com/sites/default/files/2017-10/Programming_Guide_V1.5.pdf (cit. on p. 44).

[HAR18] HARTING IT Software Development GmbH & Co. KG. *IP67 industrial computer hardware that fits anywhere*. 2018. URL: https://www.harting-mica.com/en/mica-variants (cit. on p. 42).

[HJD+17] T. Heo, J.Weiner, V. Davydov, L. Torvalds, P. Parav, W. T. King, T. Klauser, S. Hallyn, K. Khlebnikov. *Control Group v2*. Version v2. Aug. 2, 2017. URL: https://github.com/torvalds/linux/blob/8fac2f96ab86b0e14ec4e42851e21e9b518bdc55/Documentation/cgroup-v2.txt (cit. on pp. 23, 28).

[HSHF12] C. M. Hayden, K. Saur, M. Hicks, J. S. Foster. "A study of dynamic software update quiescence for multithreaded programs". In: *2012 4th International Workshop on Hot Topics in Software Upgrades (HotSWUp)*. IEEE, June 2012. DOI: 10.1109/hotswup.2012.6226617 (cit. on pp. 25, 28).

[KH14] C. P. Kruger, G. P. Hancke. "Benchmarking Internet of things devices". In: *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, July 2014. DOI: 10.1109/indin.2014.6945583 (cit. on pp. 23, 28).

[KM90] J. Kramer, J. Magee. "The evolving philosophers problem: dynamic change management". In: *IEEE Transactions on Software Engineering* 16.11 (1990), pp. 1293–1306. DOI: 10.1109/32.60317 (cit. on pp. 25, 28).

[Lap00] P. A. Laplante. *Dictionary of Computer Science, Engineering and Technology*. CRC Press, 2000. ISBN: 0849326915. URL: https://www.amazon.com/Dictionary-Computer-Science-Engineering-Technology/dp/0849326915?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0849326915 (cit. on p. 26).

[LH18] T. Lindh, M. Ha. "Enabling dynamic and lightweigth management of distributed Bluetooth low energy devices". In: *2018 International Conference on Computing, Networking and Communications :* QC 20180327. 2018, pp. 615–619. ISBN: 978-1-5386-3651-0 (cit. on p. 41).

[Mak09] K. Makris. "WHOLE-PROGRAM DYNAMIC SOFTWARE UPDATING". PhD thesis. ARIZONA STATE UNIVERSITY, Dec. 2009, p. 118 (cit. on pp. 24, 25, 28).

[Mica] Microsoft. *Understand and use device twins in IoT Hub*. URL: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins (cit. on pp. 18, 27).

[Micb] Microsoft. *What is Azure IoT Hub?* URL: https://docs.microsoft.com/ru-ru/azure/iot-hub/about-iot-hub (cit. on p. 46).

[Mis01] J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. SPRINGER VERLAG GMBH, June 26, 2001. 420 pp. ISBN: 0387952063. URL: https://www.ebook.de/de/product/4255124/jayadev_misra_a_discipline_of_multiprogramming_programming_theory_for_distributed_applications.html (cit. on p. 26).

[MKK15] R. Morabito, J. Kjallman, M. Komu. "Hypervisors vs. Lightweight Virtualization: A Performance Comparison". In: *2015 IEEE International Conference on Cloud Engineering*. IEEE, Mar. 2015. DOI: 10.1109/ic2e.2015.74 (cit. on pp. 22, 23, 27).

[MPJ18] I. Mugarza, J. Parra, E. Jacob. "Analysis of existing dynamic software updating techniques for safe and secure industrial control systems". In: *International Journal of Safety and Security Engineering* 8.1 (1 Jan. 2018), pp. 121–131. DOI: 10.2495/safe-v8-n1-121-131 (cit. on pp. 25, 28).

[ngu16] nguonly (Github-User). *LyRT with transaction*. Version latest commit 985a606. June 13, 2016. URL: https://github.com/nguonly/lyrt-with-transaction (cit. on p. 29).

[OAS15] OASIS. *MQTT Version 3.1.1 Plus Errata 01*. Ed. by A. Banks, R. Gupta. OASIS Standard Incorporating Approved Errata 01. Dec. 10, 2015. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html (cit. on pp. 19, 28).

[ORH02] A. Orso, A. Rao, M. J. Harrold. "A technique for dynamic updating of Java software". In: *International Conference on Software Maintenance, 2002. Proceedings.* IEEE Comput. Soc, 2002. DOI: 10.1109/icsm.2002.1167829 (cit. on p. 28).

[Per17] J. Perez. *LwM2M 1.0 to Azure IoT Hub Bridge Sample*. Version commit ea3b6f6. Feb. 9, 2017. URL: https://github.com/Azure-Samples/iot-hub-lwm2m-bridge (cit. on pp. 41, 45).

[PH07] L. Paradis, Q. Han. “A Survey of Fault Management in Wireless Sensor Networks”. In: *Journal of Network and Systems Management* 15.2 (Mar. 2007), pp. 171–190. DOI: 10.1007/s10922-007-9062-0 (cit. on p. 13).

[Pra04] S. Prata. *C Primer Plus (5th Edition)*. Sams Publishing, 2004. ISBN: 0-672-32696-5. URL: https://www.amazon.com/Primer-Plus-5th-Stephen-Prata/dp/0672326965?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0672326965 (cit. on p. 26).

[RBF+16] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. “Internet of things patterns”. In: *Proceedings of the 21st European Conference on Pattern Languages of Programs - EuroPlop ’16*. ACM Press, 2016. DOI: 10.1145/3011784.3011789 (cit. on pp. 13, 16, 18, 27).

[RM04] K. Romer, F. Mattern. “The design space of wireless sensor networks”. In: *IEEE Wireless Communications* 11.6 (Dec. 2004), pp. 54–61. DOI: 10.1109/mwc.2004.1368897 (cit. on pp. 15, 27).

[RM12] E. Rescorla, N. Modadugu. “Datagram Transport Layer Security Version 1.2”. In: (2012). ISSN: 2070-1721 (cit. on p. 21).

[SAG+] M. Stanley-Jones, R. Anderson, N. (Github-User), V. Bialas, gdevillele (Github-User), J. Fernandes, travis-rodman (Github-User), J. Mulhausen, M. Friis, A. Duermael. *Docker frequently asked questions (FAQ)*. Version 88e6e28. Docker Inc. URL: https://github.com/docker/docker.github.io/blob/88e6e2852bddf2ce5d33b46620df0d1cc69f5c3d/engine/faq.md (cit. on pp. 23, 28).

[SAM12] H. Seifzadeh, H. Abolhassani, M. S. Moshkenani. “A survey of dynamic software updating”. In: *Journal of Software: Evolution and Process* 25.5 (Apr. 2012), pp. 535–568. DOI: 10.1002/smr.1556 (cit. on pp. 13, 25, 28).

[SHB+07] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, I. Neamtiu. “Mutatis Mutandis”. In: *ACM Transactions on Programming Languages and Systems* 29.4 (Aug. 2007), p. 22. DOI: 10.1145/1255450.1255455 (cit. on pp. 24, 28).

[SHB14] Z. Shelby, K. Hartke, C. Bormann. *The Constrained Application Protocol (CoAP)*. Tech. rep. June 2014. DOI: 10.17487/rfc7252 (cit. on pp. 20, 28).

[SMZL15] Z. Sheng, C. Mahapatra, C. Zhu, V. C. M. Leung. “Recent Advances in Industrial Wireless Sensor Networks Toward Efficient Management in IoT”. In: *IEEE Access* 3 (2015), pp. 622–637 (cit. on p. 13).

[Vau06] S. Vaughan-Nichols. “New Approach to Virtualization Is a Lightweight”. In: *Computer* 39.11 (Nov. 2006), pp. 12–14. DOI: 10.1109/mc.2006.393 (cit. on pp. 22, 27).

[VEBD07] Y. Vandewoude, P. Ebraert, Y. Berbers, T. D’Hondt. “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates”. In: *IEEE Transactions on Software Engineering* 33.12 (Dec. 2007), pp. 856–868. DOI: 10.1109/tse.2007.70733 (cit. on pp. 25, 28).

[WR] K. Walther, J. Regtmeier. *Virtualization for Manufacturing and IoT*. HARTING IT Software Development GmbH & Co. KG (cit. on p. 42).

[WTW+16] M. Weisbach, N. Taing, M. Wutzler, T. Springer, A. Schill, S. Clarke. “Decentralized coordination of dynamic software updates in the Internet of Things”. In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. IEEE, Dec. 2016. DOI: 10.1109/wf-iot.2016.7845450 (cit. on p. 29).

All links were last followed on August 31, 2018.

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