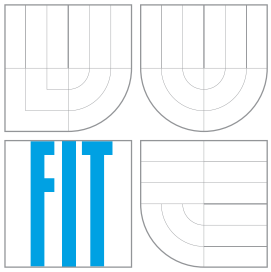
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

REMOTE MANAGEMENT OF EMBEDDED SYSTEMS

VZDÁLENÁ SPRÁVA VESTAVĚNÝCH SYSTÉMŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
PETER MALINA

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. JAN VIKTORIN,

BRNO 2016

Zadání bakalářské práce/18302/2015/xmalin26

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Malina Peter**
Obor: Informační technologie
Téma: **Vzdálená správa vestavěných systémů**
Remote Management of Embedded Systems
Kategorie: Vestavěné systémy

Pokyny:

1. Seznamte se s dostupnými řešeními pro vzdálenou správu vestavěných systémů se zaměřením na aktualizace, monitorování, změnu nastavení, atd.
2. Seznamte se se systémem řízení inteligentní domácnosti s ohledem na použitá zařízení.
3. Navrhněte aplikaci pro vzdálenou správu těchto zařízení.
4. Aplikaci implementujte
5. Zhodnoťte funkčnost a použitelnost navrženého řešení a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Viktorin Jan, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstract

Possibilities of today's embedded devices are growing rapidly. Their performance allows them to run more complex applications in Internet of Things (IoT) environments. Complex applications tend to be error-prone and require continual updates. A system that is capable of updating a multitude of remote embedded devices was designed and implemented. This system was created based on the study of existing solutions and requirements of the project BeeeOn, which concerns itself with smart homes.

Abstrakt

Možnosti dnešních vestavěných zařízení rapidně rostou. Jejich výkon dovoluje běh složitějších aplikací v prostředích Internetu věcí (IoT). Složité aplikace bývají náchylné na chyby a vyžadují průběžnou aktualizaci. Systém, který umožňuje aktualizace většího množství vzdálených vestavěných zařízení, byl navrhnut a implementován. Systém byl implementován na základě studie existujících řešení a podmínek projektu BeeeOn, který se zabývá chytrou domácností.

Keywords

Internet of Things, Embedded system, Software update, Smart home

Klíčová slova

Internet věcí, Vestavěné systémy, Aktualizace software. Smart home

Reference

MALINA, Peter. *Remote Management of Embedded Systems.* Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Viktorin Jan.

Remote Management of Embedded Systems

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mr. Ing. Jan Viktorin The supplementary information was provided by Ing. Tomáš Novotný All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Peter Malina
May 17, 2016

Acknowledgements

I would like to thank to Ing. Jan Viktorin for his willingness, support and advices during the writing of this thesis. I would also like to thank to Ing. Tomáš Novotný for his guidance and help during the implementation phase.

Contents

Chapter 1

Introduction

A market of IoT (Internet of Things) devices is growing rapidly every year. We are now talking about billions of devices connected to the Internet [18]. Most of these devices can also be called embedded devices. An embedded device usually lives with the same software its whole life. However, the development process speeds up rapidly and possibilities of modern embedded devices are growing. This situation opens up a way to update a software of an embedded device instead of its full replacement.

There are at least two major challenges for the development teams regarding device updates. The first challenge is to ensure that a device is updated correctly. This means that a device and its updated software continues to work properly. The second challenge is possible installation and runtime errors. Most software issues are often visible and fixed during the testing phase of an application development. However, many issues become apparent after an application is already deployed. Security issues may be discovered after months of a successful production use and globally misused in a matter of hours.

This thesis describes basic principles of embedded systems, principles of software update and an embedded device initialization after it is produced. It introduces the A10-OLinuXino-LIME embedded device used during the system implementation. Next, it presents existing solutions and their possibilities.

The work covers a basic information about the BeeeOn system of a smart home. The BeeeOn system is the target environment for software implemented in this work. The thesis then describes the implementation of a system that can be used to rapidly and reliably distribute new software to embedded devices. It describes a process of a continuous device monitoring that is helpful ragarding the issue discovering. The last part describes an applicability of the implemented system and possibilities for its extension.

Chapter 2

Principles

This chapter gives to the reader a basic understanding of embedded systems, their components and how the processes of updating and monitoring are handled. It describes embedded systems in general, i.e. their memory, CPU and peripherals. It also explains why is a process of a plain device setup important.

2.1 Embedded System

An embedded system is a device that is a combination of a software and hardware, specialized to do a particular function. They are commonly used in industry, automobiles, airplanes and more [14]. Embedded systems are commonly used as parts of other systems, while their internal behavior is usually hidden from the user.

2.2 CPU, memory and peripherals in embedded systems

Each embedded system consists at least of:

- a CPU
- a memory
- a set of peripherals

These components are giving an embedded system a capability to compute, store data and communicate with an environment.

Embedded systems are frequently designed to have a very low latency. This enables them to quickly react to external events. An external event is usually captured by a sensor as an analog signal. The embedded device therefore converts the incoming analog signal to a digital representation. Reactions to external events are usually done by actuators. An actuator generally expects an analog input. Thus, the embedded device converts an output digital signal to analog once it is processed as shown in figure 2.1 [11, 9.3.2 Signal Conditioning].

Embedded systems are mostly built with a performance in mind. Field-programmable gate array(FPGA) or Application Specific Integrated Circuit(ASIC) is often inseparable part of them. However, utilization of these parts makes remote updates complicated. ASICs provide minimal capability for a configuration and FPGAs need quite a big configuration image (depends on a chip size).

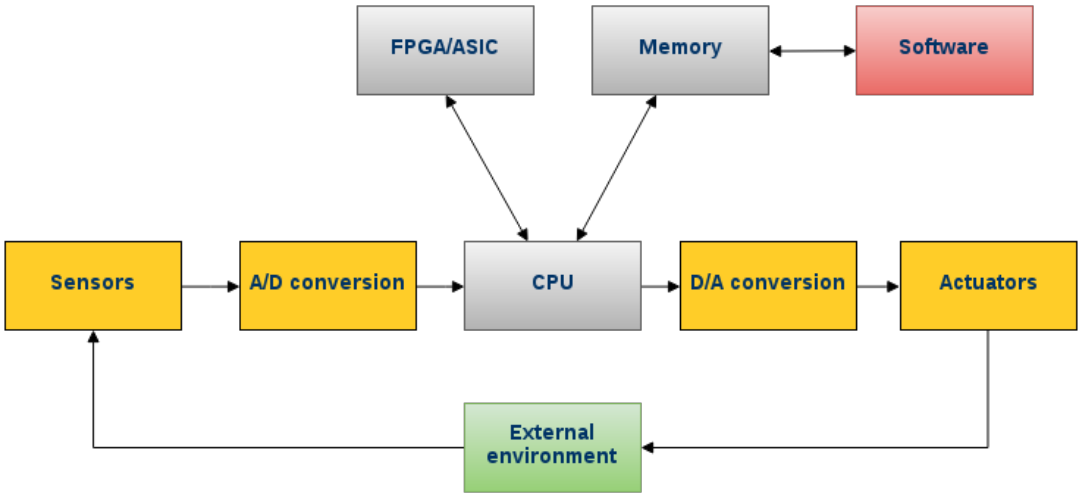


Figure 2.1: Architecture of an embedded system

2.3 Principles of update

A software update belongs to a release stage of the software development life cycle 2.2. It is solving the major problem of replacing an old software version with the new one. A software update is commonly used to patch errors and add new features.

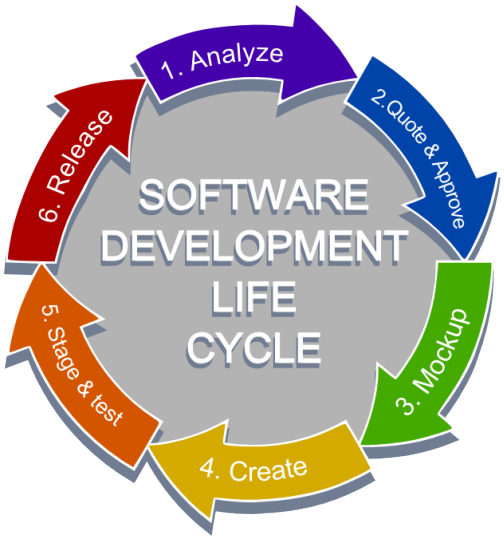


Figure 2.2: A software update consists of more stages. Each stage is essential to for a quality of a product.

Source: http://kobridgeconsulting.com/default-item/software-development-life-cycle-training/

There are usually more versions of the same software, moreover, there may be more versions and types of hardware. This opens up a new problem of compatibility, where not every device is able to run all the written software. Thus, the update software process needs to be able to distinguish the systems and their possibilities.

A process of software update should always satisfy a previously specified criteria. These criteria must be based on the critical parts of the software, type of the software distribution, a device that should run the software and other specific conditions. One of the biggest challenges is an ability to update a software immediately after a stable version was released. This situation gets even more critical if an old software contains security issues.

2.4 Factory state

Devices that come from the manufacture are in the so called factory state. These devices are provided with a data storage that holds a system image they should execute. However, this is not the required state of a device. A device must be uniquely distinguishable to operate correctly with a remote services. An initialization of a device is therefore needed.

A factory script may be used to retrieve this data and register device to a remote database of devices. Once the device has an authentication data, it is able to work correctly with remote services.

The factory script is often responsible for an initial network security setup of a device. It may load certificate by which a device can be uniquely distinguished and connected to a private network. However if this fails without a notice and the certificate or authentication information is stolen, it may lead to an unauthorized access to the data that are provided to the device.

A device also needs to be correctly configured before it is ready for the production use. Configuration files are therefor also loaded into the device with the initial setup. Wrong configuration of the device may lead to a inability of a device to run correctly, e.g. in case of a host and port configurations for a remote services.

Chapter 3

Goals

This section describes the goals of the work and provides a basic information about the device that is used during the development and experiments.

3.1 Introduction to BeeeOn

BeeeOn is a project focused on smart homes with a goal to provide smart home components into houses without need of any reconstruction. This includes sensoric devices that can read environmental data like temperature, air pressure, humidity, gas concentration or motion [2]. A system that is described in this work should run on devices mainly responsible for a data gathering from sensors and controlling of other connected devices.

There are many devices that are used in the whole project, from which this work focuses mainly on a device that is responsible for the management of a so-called Adapter device, which is responsible for the communication between the smart house and servers which store and process a collected data.

The main goal of the described system is its ability to automatically update and monitor a high number of adapters, as the project intends to spread to a wide range of homes, where improper maintaining may lead to an extensive time loss.

3.2 Requirements for the system

A software update process should always satisfy defined criteria depending on the target production environment. The target environment consists of a device itself and other aspects that affect that device. These criteria include hardware limitations, security, availability and user experience. There are many critical criteria, that should always be satisfied in a production environment.

One of the main criteria for the system to be usable on an embedded device is its size and performance. Thus the system must be lightweight to be able to run in the performance and memory limited environment.

Security of the system is also one of the main challenges and criteria for the system. As the system operates with a private data of a device such as MAC address or device ID, it is necessary to be able to distinguish if the data is sent to the correct destination. The data that are sent through the network should be at least in a human unreadable form, so they are unlikely to be easily decoded.

A system should be modular to be able to effectively replace modules based on the target environment device. Different devices may provide the data differently based on their hardware and configuration. Use cases of devices may greatly vary based on the requirements for their usage, thus the system should be composed from more smaller parts which can be enabled or disabled depending on the needs.

A system that is responsible for an updates and error reporting must be sufficiently reliable due to its potential to break the system or leave it unconsciously in a non-consistent state, alternatively reporting errors that are not present on a device.

Next requirement of the system is the ability to run on an application layer 3.1 of an Linux operating system. The system should be able to easily manage applications that are running on a device, read and update their configurations and retrieve their current state in a case it should be reported or checked.

Last requirement comes from the fact, that devices mostly run more applications at the same time. A system should be integrable into an existing system and environment. Modularity of the system should assure, that missing modules can be can be simply integrated.

The range of criteria is pretty wide and the basic specification only assumes common environment conditions. This does not include any extreme conditions such as memory and network corruptions or high amount of radiation.

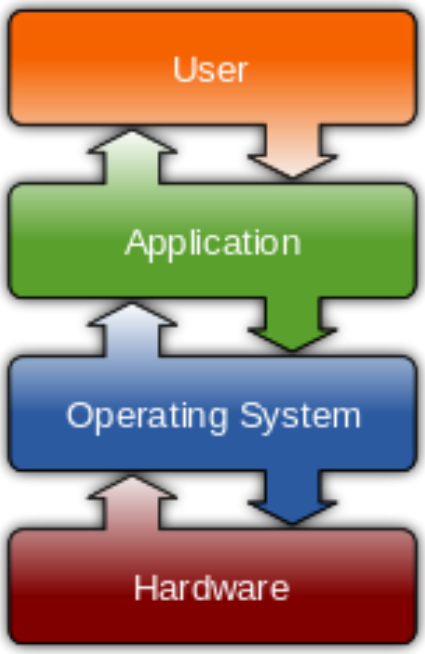


Figure 3.1: Layers of a general system. A device running the system for update and monitoring should run on the Application layer of the general system.

Source: https://en.wikipedia.org/wiki/Operating_system

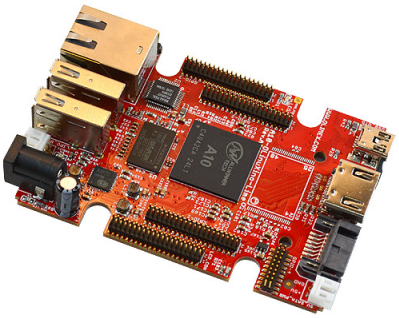
3.3 A10-OLinuXino-LIME

A10-OLinuXino-LIME (also shown on figures 3.2a and 3.2b) is an embedded system that was selected as a core component of the smart house ecosystem. This device comes with the A10 Cortex-A8 processor, 512 MB of DDR3 RAM memory, USB, HDMI and SATA connectors and supports 100M(MegaBit) ethernet connector. There are many more available features like power and battery charge leds, available buttons or EEPROM. It has 5$\tilde{V}$ input power supply [13]. A MicroSD card connector enables to store a larger amount of data on the device. This also provides a capability to instantly replace the system that should be executed.

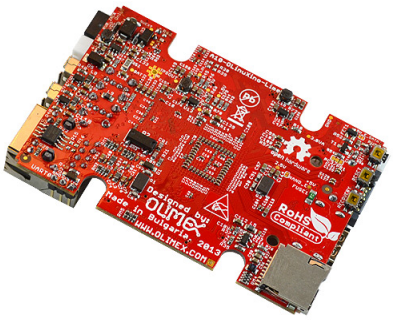
A size of the device is 84x60 mm which makes it portable and easily positionable within a smart house. However, the device requires an available power supply and Internet connection. This limits the placement of a device to the positions where sockets of a house are

installed.

An ideal position of the device is intended to be next to the Internet router, where it can be simply connected to the Internet.



(a) A10-OLinuXino-LIME top view [13]



(b) A10-OLinuXino-LIME bottom view [13]

Chapter 4

Existing solutions

This chapter provides a basic information about an existing solutions that provide the capabilities for update, remote execution, automatic repair or monitoring.

4.1 Manual management

Manual management of remote devices is not the most common way to handle an update process or monitoring. While sometimes manual repair or troubleshooting is necessary, it often consumes a lot of time and needs a proper amount of information about the system running on the device. Manual management of high amount of remote devices might be error-prone and insecure, as manipulating different devices requires proper inspection and knowledge of the system.

A device that should be manually managed may be either a local one, thus we have direct access to it, or remote one. A device should provide an interface for the communication in case it is remote. SSH (Secure Shell) is one of the most common ways to access a remote devices [1]. Manual management of a local device is more straight forward. It does not involve many network security risks and includes possibility to directly affect the device, e.g. by changing an external memory storage, such as SD card. This approach is commonly used on the development or debug devices.

4.2 Start-up scripts

Devices running an operating system (OS) often use some kind of start-up scripts to start additional applications. They can also be used, to execute one-shot scripts, which may report status of the device and check for possible updates.

Linux systems are executing start-up scripts contained in the */etc/init.d/* folder. These scripts are commonly written in the Bash[2] shell. A shell is both a programming language and an interpreter. Shell is a macro processor that executes commands. A Macro is a command expanded to a larger expression [7].

Usage of start-up scripts may solve the problem of software updates if we have access to the device and are able to restart it after each software release. However, scripts are usually weakly typed with a dynamic type-checking. This makes them harder to read, test and debug.

[1] Telnet service was used previously, but it is not considered for this text, as it is insecure and obsolete.
[2] https://www.gnu.org/software/bash/

4.3 Bootloaders

A bootloader is started by the Basic Input Output System (BIOS)[3] and is responsible to load the operating system kernel and prepare the execution environment before an OS execution starts. Bootloaders often include a scripting language that makes it possible to use a certain booting recipe such as:

- booting in a defined order from a storage connected to a device
- booting from network

However, the level of accessing a booting image is pretty low. The network access is usually limited to UDP/IP (e.g. Trivial File Transfer Protocol (TFTP)) and only certain parts of a file system are accessible.

4.3.1 GRUB 2

GRUB2 is a bootloader that is used by the major Linux distributions such as Ubuntu, Fedora or CentOS. GRUB2 is a successor to the original GRUB bootloader. It supports a wide range of processor architectures. It also supports a large amount of the file systems such as: ext, File Allocation Table (FAT) or New Technology File System (NTFS). It extends the original GRUB by the features like rescue mode, dynamic module loading, scripting support and live ISO image OS loading [3].

4.3.2 Syslinux

Syslinux is a collection of a lightweight bootloaders. It runs on the FAT file system. It had an aim to simplify the first-time installation of the Linux systems and special purposes as creating rescue or boot disks [1]. Syslinux is also scriptable via Lua. Lua is a lightweight easily embeddable scripting language[15]. Syslinux also supports the TFTP transfer protocol.

4.3.3 Das U-Boot

Das U-Boot is a scriptable bootloader with support for wide range of operating systems as Linux, OpenBSD or Solaris. It also supports a range of file systems as FAT, ext2, ext3 and ext4. It also supports Network File System (NFS) and TFTP at the network level.

4.3.4 LILO

LILO is a bootloader that can boot the operating system from any file system. It can only access the hard disk via BIOS drivers. It is not scriptable, but supports bzip2 and gzip decompression. LILO was recently discontinued in December 2015, due to the lack of developers.

4.3.5 RedBoot

Redboot is a bootloader written specifically for use in embedded systems. It is lightweight, configurable and portable. It supports both, development and production environments.

[3] ARM systems use System on a Chip (SoC) instead of BIOS.

It also supports booting via TFTP. It can be used to debug applications with GDB via a serial or Ethernet cable. It provides an interactive command-line interface to simplify configuration editing or image downloads. It supports a booting scripts that can be loaded on the start, allowing a device to load an image via TFTP or from the Flash memory [16].

4.4 Package Management systems

A package management system is nowadays bundled with every major Linux distribution. It is used to install, update or remove a software on an underlying OS. There are severl such systems, the most important are:

- rpm
- dnf
- dpkg
- opkg
- Guix

Package Management systems are very powerful in resolving dependencies or possible conflicts in the package versions. While they often do not provide atomic updates, there are some which do, for example Guix. Package management systems can be used together with start-up scripts, Cron or more to check for updates once in a while. A fallback mechanism has to be added to provide a fail-safe self-updating system.r

4.4.1 rpm

The rpm[4] was created to be used in the Red Hat Linux but can be nowadays found in many other Linux distributions. It allows cryptographic verification of packages with Message-Digest algorithm (MD50 and the GNU Privacy Guard (GPG). It also includes original source archives, which makes it easier to verify a package by comparing hashes of an original and downloaded package. RPM is able to create patch files to allow configuration patching. The rpm resolves defined dependencies of a package during the package build time. This leads to a faster package resolving when a package is being installed.

4.4.2 dpkg

Debian based operating systems are bundled with the dpkg package manager. The major difference from the other package managers is it can not automatically download packages or their dependencies. That is why it often comes with the Advanced Package Tool (APT)[5]. The APT is a collection of tools created to handle *.deb* (Debian) packages. It nowadays supports also rpm packages. One of APT tools is apt-get. The apt-get handles the installation and modification of Debian packages. APT is using repositories to look up packages. A repository is a collection of packages. Repositories are commonly used to store packages for different operating systems. APT supports so called pinning. Pinning allows a user to specify which version and repository should be used to install the package. This would not allow a package to be updated if such update may lead to conflicts.

[4] http://www.rpm.org/
[5] https://wiki.debian.org/Apt

4.4.3 opkg

The opkg package manager[6] is intended to be used in embedded Linux devices. As all other package managers, it is also able to install, update, upgrade or remove packages. It handles OpenWrt[7] packages. The opkg was forked from the Itsy Package Management System (ipkg)[8]. These packages use the *.ipk* extension.

4.4.4 Guix

Guix is a functional package management tool that supports atomic updates. An atomic update is either entire complete or fails and all changes are rolled back. Guix also supports rollbacks and has an ability to remove unused packages that are no longer referenced. It isolates a build processes into containers. This gives the build only access to directories and files that are in the current build. After the package is built, it is placed into a store. A store is used to save correctly built packages. Every package is installed to their own directory in the store.

Guix supports profiles, which enable users to use only the packages they want. A profile is stored in the *$HOME/.guix-profile* file. A profile points to all packages that are used by a user.

4.5 SWUpdate

SWUpdate is a Linux update agent. It is able to update an embedded system's software. This system also provides pre- and post-install Lua scripts. Developers that are already familiar with the Lua programming language can easily write scripts, that may establish or repair the environment and check, if the installation was completed correctly. This also opens a way to provide the built-in functionality in the SWUpdate to Lua scripts. This solution handles more hardware memory configurations. This makes it easier to deploy to a wide range of devices.

An update package is described by the SWUpdate configuration file written in Extensible Markup Language (XML)[9]. This configuration file should provide all necessary information to perform the update process. Configuration files provide information for each device that should be supported, making the package available for more devices.

4.6 Turris

Turris is an auto-updating router from the CZ.NIC company. It is able to protect a home network by analyzing its data streams. It has a built-in reporting mechanisms. The Turris developers can get information about the dangers in the network, fix them and automatically roll the updates to all of the Turris devices [4].

Turris devices have a capability to analyze data that flow to and from its network, which gives them an ability to assume cyber-attacks.

[6] https://wiki.openwrt.org/doc/techref/opkg
[7] https://openwrt.org/
[8] https://en.wikipedia.org/wiki/Ipkg
[9] https://www.w3.org/XML/

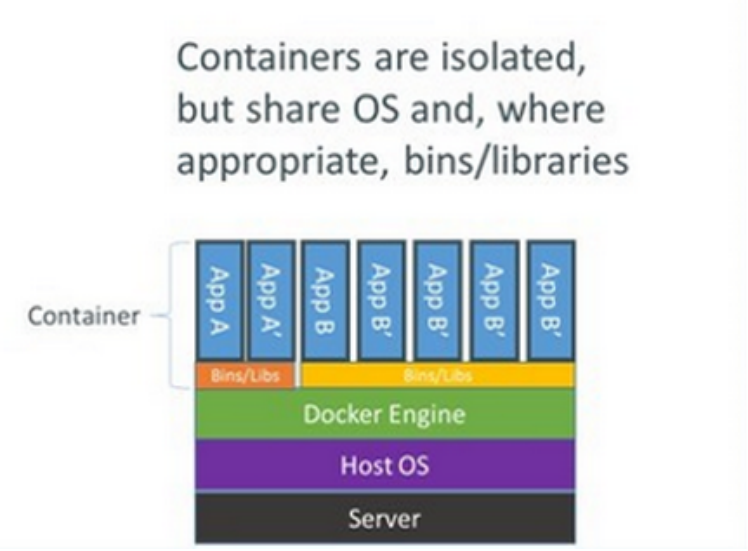


Figure 4.1: An architecture of a system running Docker

Source: https://www.upguard.com/articles/docker-vs.-vmware-how-do-they-stack-up

However, the original Turris devices are no longer distributed and new Turris Omnia[10] devices are nearly ten times more expensive then previously discussed A10-OLinuXino-LIME.

4.7 Docker

Containerization is today experiencing a huge boom. It provides an isolated environment for the applications to run with the only requirement of the Linux kernel. This enables developers to create a development environment that is much closer to the production environment. It also enables applications to run in a consistent container, which is not affected by the changes of the outside system. A container is also not affecting the outside system.

Docker is a lightweight, open and secure[11] platform for application delivering. It is using the container technology to isolate applications from each other as shown on the figure 4.1. Docker is growing in a popularity and can often replace a virtual machine. While Docker is a great choice for applications that run on a server, the memory, CPU and disk requirements are hard to meet in embedded systems.

Docker containers provide isolated environment for applications to run. Two docker containers with the same dependencies on different versions would not collide. Containers also provide a root file system. A separate file system makes an illusion of an autonomous system. Applications that depend on each other on a network level can be linked together via Focker networking system. The Docker networking system is isolated from an outer network (i.e. Internet). Any port exposure must be therefore explicitly made.

While docker can not be used as an ordinary package management tool, it resembles it

[10] https://omnia.turris.cz/en/
[11] https://www.docker.com/docker-security

by its behavior. Docker has a pull and push functions, which allow user to pull the container image from the repository (install) or push a new version of the image into the repository (publish). All docker images must be built from a source via *docker build* command.

4.8 Continuous Integration systems

Continuous Integration System (CI) is a system, that is used to automate process of an application building and testing, often providing a way to deploy built software straight to the device or push it to the repository. Repositories can then be used to distribute the application.

CI can run the build and tests in the isolated container, that is able to emulate the behavior of a device. This allows developers to test applications even before they are run on the specific embedded device.

4.9 SystemD

SystemD is a Linux init system which has been adopted to the many Linux distributions by far. It consists of a number of tools, that provide application managing, logs or network interface configurations. SystemD is using the *Unit files* to store configurations of specific daemons. This system allows on-device unit management and monitoring.

4.10 Conclusion on existing solutions

There are many solutions for a software distribution and tracking of logs. Moreover, they are widely used and many of them are developed by a company or a team of professionals. Package management systems usually provide a fallback mechanism in case of installation failure and are great at resolving dependencies. There are solutions like Docker that took it even further, isolating applications from each other, thus evading dependency collisions. At last, continuous integration systems are able to deploy a new software just after it was successfully built.

However, most of the solutions are designed to operate only locally. A direct access to a device is therefore necessary. An auto deployment via a CI system is hardly utilizable due to a potential of high amount of devices. A high amount of devices would unnecessarily lock the system until all devices were updated. The proposed solution is to utilize the power of already existing tools like package managers or SystemD by an automated remote interface.

Chapter 5

System of a smart home

BeeeOn system of a smart home is an Internet of Things (IoT) system by design. It consists of a couple of components, which are communicating together, to simplify processes in a house. There may be many use cases from the utilization to the automation of the house. While sensors are able to read values from an environment such as temperature, pressure or light, other devices can receive the values and trigger a specific events based on it.

5.1 Components

This section will cover major components of the intelligent housing system. There are 3 core components of the system:

- cloud
- gateway
- sensors

The most distant component from the user is the cloud, which is used to store and distribute data across the network of the devices. Gateway, on the other hand, is used to collect the data from the sensors and distribute them to the cloud and in the private network of the sensors. After all, sensors are used to get the actual values from the environment.

5.1.1 Cloud

The cloud is used to store and distribute data around the network of the devices that are connected to it as shown on the figure 5.1. Cloud is a server, that accepts connections from the gateways to receive the data and send them commands that need to be executed on the gateway. It also accepts connections from the other devices, like mobile phones, smart watches running application or PCs running web applications. While the data transfer may feel like a main purpose of the cloud, it must also store the data. Cloud persists the data that are being processed to ensure a wide range of devices is able to access them. They are also commonly used to track what is happening in the system.

System therefore consists of persistent database, respectively PostgreSQL[1]. Database allows Cloud to persist data for the future use like post processing, analysis or statistics.

[1] http://www.postgresql.org/

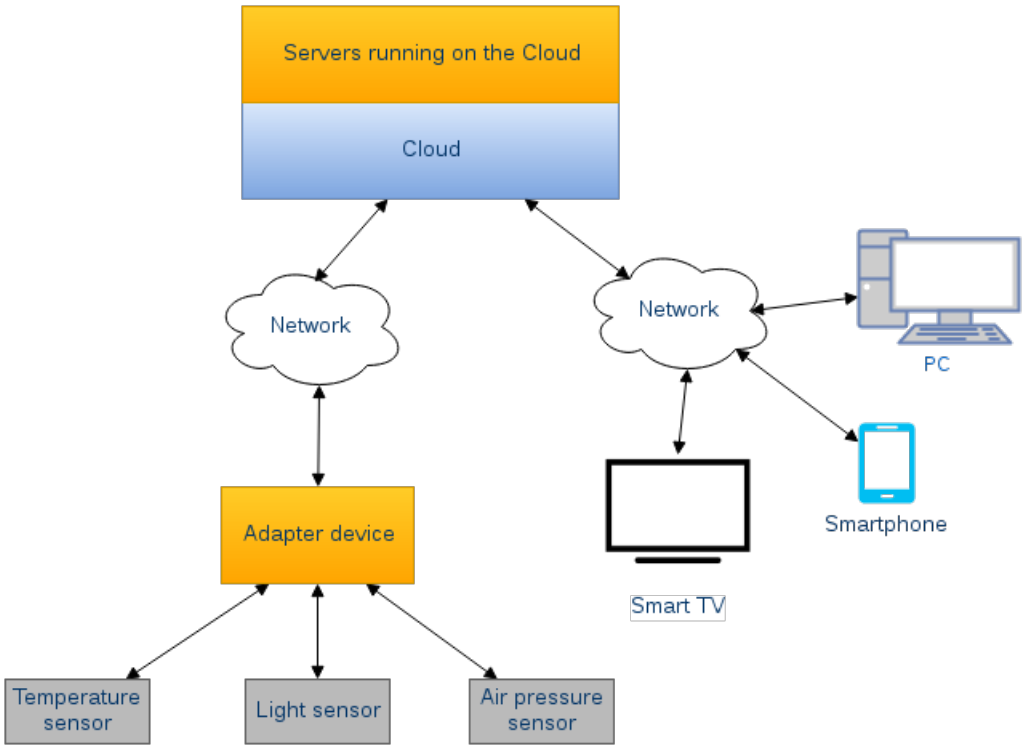


Figure 5.1: An architecture overview of the smart house project

Cloud consists of applications, that are handling specific parts of the system communication. The main applications are *UIServer* and *AdaServer*. While the UIServer is there to handle the communication with an android devices, the AdaServer communicates directly with gateways. Both applications are communicating together via their own protocol. There are also other applications running. *Web* is used to handle the traffic coming from the web application users. Web server is directly connected only to the database, which allows it to fetch newest data. At last, *Manager* server application is running on the Cloud to allow Gateways to connect to it and perform updates or maintenance.

5.1.2 Gateway

Gateway is a main collector, distributor and manager of the specific intelligent house system. Gateway collects the data from the sensors and sends commands to other connected devices. The main role is to distribute the data collected from the sensors to the Cloud, where they are processed. Gateway runs the client version of AdaServer called AdaApp [2], which connects directly to the Cloud's AdaServer. The channel that is created between these two applications is able to transfer all collected data and control commands.

While all the data is distributed to the Cloud, it is also cached on the Gateway. This allows other applications on the Gateway to access them. They are distributed in real-time using MQ Telemetry Transport (MQTT). The data are also stored in a cache for some time after sending to the server in case Gateway unexpectedly disconnected from the Cloud.

Applications that are executed on the gateway device must be managed. The device

[2] https://beeeon.org/index.php?title=Gateway#Firmware

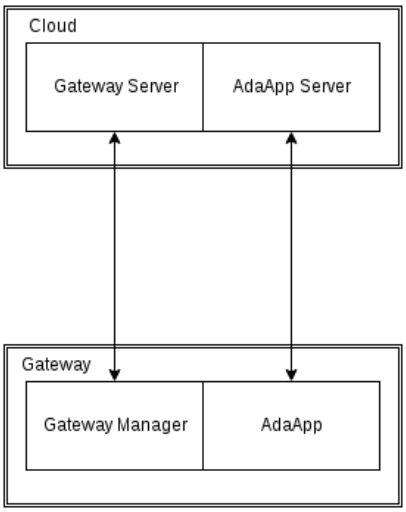


Figure 5.2: Placement of two main applications on the Cloud and gateway device

should be monitored in case of errors or unexpected behavior. This is where the remote manager belongs. The Gateway Manager must run with root permissions to have the correct access rights. The manager is also connecting to the Cloud and its server counterpart. This allows developers and support to update or maintain the device manually or automatically. Placement of AdaApp and Gateway Manager can be seeon on the figure 5.2.

The transferred sensoric and command data must be correctly secured. Gateway thus uses a VPN to accomplish this. Proper usage of SSL (Secure Sockets Layer) certificates should remove most of the dangers in the open network.

5.1.3 Sensors

Sensors are used to collect data from an environment they are in. They are directly connecting to the Gateway to send the data. Unlike other devices in the smart home system stack, sensors are under a power only once in a while. Their purpose is to get an environmental data and send them as fast as possible to the gateway. They are mostly not connected to a power supply. The power consumption is one of big challenges. This challenge is solved by the batteries that can be replaced.

There is a couple of specific sensors, that are used to measure temperature, air pressure or humidity. Every sensor has its specific hardware which reflects on the battery consumption.

5.1.4 Secondary Gateway management channels

As mentioned before, Gateway has a primary manager application, which allows the Cloud or to maintain gateway devices. Since the manager is in the development, there are operations which require manual execution. In this case, the gateway supports a direct SSH connection. This is limited due to the fact, that Gateway must have a public IP address or a connection to the private or local network. A direct access to the device is common in the development environment. However, most of devices does not have a public IP when deployed.

Chapter 6

Design and implementation of the Gateway Manager system

The Gateway Manager is a software integrated into the BeeeOn smart house system. It provides a communication channel between gateways and the cloud. Both parts, the server and the client are written in the C++[1] programming language with supporting libraries Poco[2], Google Protocol Buffers[3] and Soci[4].

The Gateway management system also consists of a factory script[5] that allows to initialize the device for the deployment. This factory script is written in the Python[6] language. The factory script is directly communicating with a factory server written in Golang[7]. Libraries used within the factory server will be covered independently.

6.1 The version control

A version control is an important part of the project development flow. It is needed because of it's ability to track all work in iterations, explore the history of changes, review the work that was done. However, the main reason to use a version control lies in a possibility to have multiple code branches, which enables to modify specific scenarios without constantly breaking an existing code.

Git[8] has been chosen because of it's features and simple usage. Git is a distributed version control system able to handle all sizes of projects with a great speed [5].

The Gateway Manager project was divided into 5 Git repositories: gateway-manager-core, gateway-manager-server, gateway-manager-client and adapter-factory. The last Git repository was created for the toolbelt that is being used on a devices that diverged from the adapter-factory repository.

The Gateway Manager project adopted the git-flow[9] Git extension set for the Git operations to be less error-prone and more effective. The git-flow extension set allows to

[1] http://www.cplusplus.com/info/description/
[2] http://pocoproject.org/
[3] https://developers.google.com/protocol-buffers/
[4] http://soci.sourceforge.net/
[5] https://github.com/BeeeOn/adapter-tools/blob/master/factory-script/
[6] https://www.python.org/
[7] https://golang.org/
[8] https://git-scm.com/
[9] http://danielkummer.github.io/git-flow-cheatsheet/

effectively use the Vincent Driesen's branching model.

6.1.1 The branching model

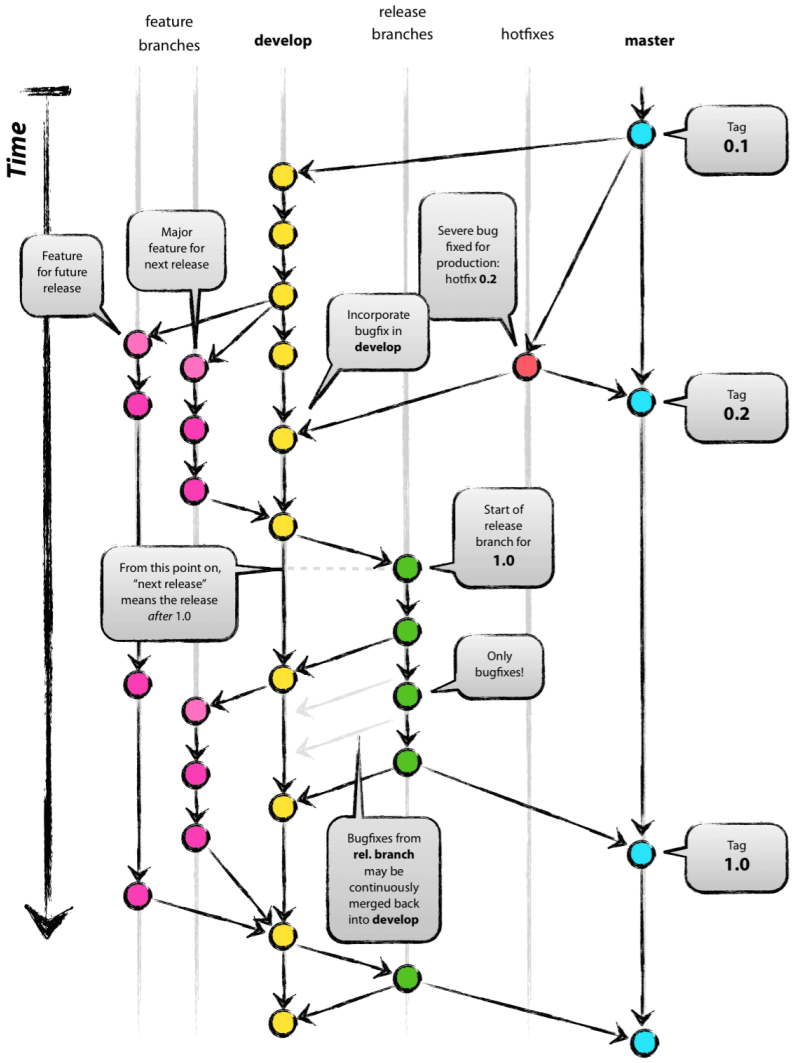


Figure 6.1: Vincent Driesen's branching model that is used during the project implementation

Source: http://nvie.com/posts/a-successful-git-branching-model/

The model shown on the figure 6.1 consists of more branches with different purposes. Starting from the right, the master branch is the main branch for releases. Master branch should be updated only in a case new stable version of the system is released.

The hotfixes branch serves in a special situations, where the system is already released but a critical errors are recognized. This branch is where the fixing happens. Once the fix is tested, it is pushed into the master and develop branches.

Most of the development usually occurs in the develop branch. However, if an implementation is bigger and may cause instability of the develop branch, a feature branch is

created. Once a feature is tested against the develop branch, it is merged into it.

After an iteration of development is done, the release branch is used. This is usually the last step before a release. Only fixing is allowed in this branch and all fixes are continuously merged into the develop branch. If all tests pass, the version is released and tagged in the master branch. From this moment, everything starts again in the develop branch.

6.2 Languages and libraries

Standard C++11 is a powerful language to write a software. It helps to maintain an object oriented architecture and provides a basic set of features like collections, strings and exceptions. However, SSL encryption, file compression, data serialization and connection to a database can be solved by libraries built exactly for that purpose.

Three main libraries have been chosen to enhance the base feature set of the C++11:

- Poco as a general purpose library with a large set of networking, data manipulation and utility components
- Protocol Buffers for the purpose of data serialization and deserialization
- Soci as a PostgreSQL connector

6.2.1 Poco project

The Poco project is a set of libraries which provides a general use capabilities by its wide range of modules (also shown on figure 6.2), such as logging, networking, compression, cryptography or multithreading [6]. It's documentation mainly consists of presentations and generated documentation from inline code comments.

The logging capability of the Poco library is widely used in the implementation of the gateway software. This is due to the easy configuration management, eight logging levels from the trace to the fatal priority. These configurations can be reconfigured during the runtime.

The networking library of the Poco project provides a high level components such as:

- reactors that eliminate a need of socket polling, replacing the polling mechanism with observers[10]
- Hypertext Transfer Protocol (HTTP), FTP or email handling modules
- Transmission Control Protocol (TCP) server and client modules

Socket implementation of the Poco Net library enhances the socket implementation of an underlying OS, by simplifying the use of non-blocking actions. The Poco networking library also supports I/O operations on a socket for collections like vector or queue.

The Zip library of the Poco project provides an ability to compress and decompress files or byte streams. The zip library is needed in the gateway manager software, due to packing and unpacking of the data blobs when transferring a large files like updates or assets.

[10] https://sourcemaking.com/design_patterns/observer

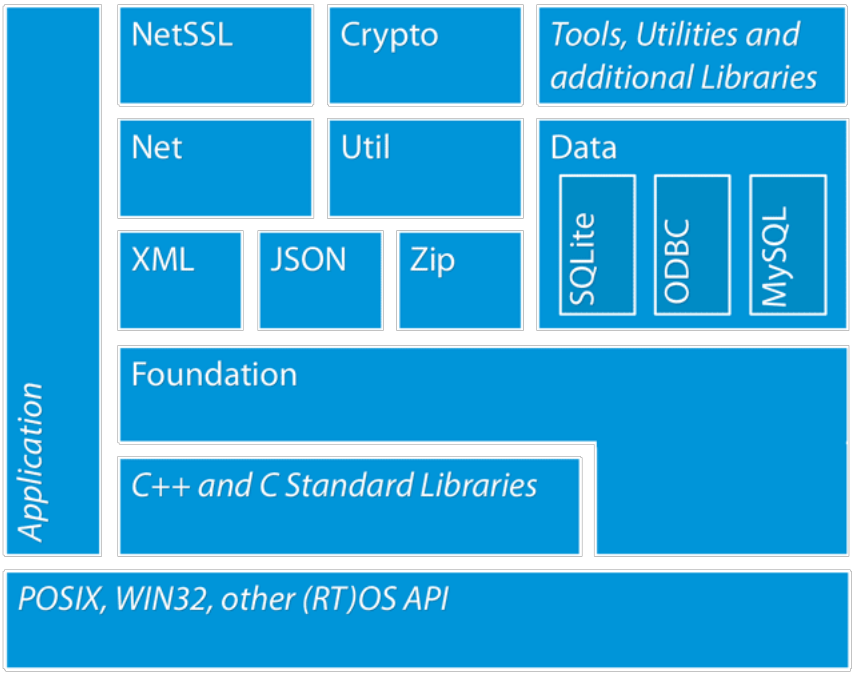


Figure 6.2: Poco project libraries infrastructure

Source: http://pocoproject.org/features.html

6.2.2 Protocol Buffers

Protocol Buffers is a library that allows a fast, automated and simple serialization and deserialization of data structures [8]. It is using a byte encoding unreadable by a human eye, which makes it more efficient than other serialization mechanisms like JSON or XML in a terms of a transfer speed and total byte size.

Each message can be different, thus needs a different way to be serialized or deserialized. Protocol Buffers generates message definition files for this purpose. These files can be generated in languages like C++, C#, Go or Java, which helps developers to utilize a power of the given language, such as static type checking. This also speeds up serialization process, as the serialization algorithm is determined during the time of code generation, rather than application runtime.

Protocol Buffers comes with it's own neutral language, which provides a way to define messages. This provides a capability to write an application infrastructure in more programming languages while maintaining the same message encoding.

Listing 6.1: Protocol Buffers definition file that defines a Ping message

```
syntax = "proto3";

package protocol;

message Ping {
        uint32 id = 1;
        string message = 2;
```

}

Listing 6.1 shows a Protocol Buffers definition file, that defines a Ping message. Protocol Buffers library supports more versions of its syntax. That is why we need to define which version should be used on the first line. An optional package definition allows us to encapsulate defined messages. This behavior is specific for the language we want generate, e.g. C++ will encapsulate all messages in a namespace with the given name [10]

Any message definition within the Protocol Buffers definition file starts with the keyword *message*, continuing with a name of the message. Listing 6.1 shows a message definition with two fields. Each field in a definition is uniquely numbered and has a name and a value type [9].

6.2.3 Soci

Soci is a C++ Database Access Library. It simplifies a writing of Structured Query Language (SQL) queries in C++. It currently has a support for DB2, Firebird, MySQL, ODBC, Oracle, PostgreSQL and SQLite databases [12].

Soci provides a simple interface for the communication with the database and error handling. It uses the basic SQL syntax with the custom syntax extensions that allow a developer to use application variables in queries and get values from results back to an application as shown in listing 6.2.

Listing 6.2: An exmaple of Soci query from the official Soci website [12]

```
int id = ...;
string name;
int salary;

sql << "select name, salary from persons where id = " << id,
        into(name), into(salary);
```

Soci is licensed under the Boost license which allows it to be used in a non-commercial or commercial products, while maintaining the copyrights.

6.3 Gateway Factory server

The Gateway Factory server involves implementation of an interface that can be used to retrieve and save data about the adapters. This server is intended to be used in a private network due to sharing of private information like SSL certificates.

Golang programming language was selected for the purpose of implementation of this server. Golang is a simple language with a strong standard and a high amount of supporting libraries. Golang brings a capability to built concurrent applications easily, compared to other languages like C, C++ or Python. Two main libraries are used to implement the factory server. Gorm[11] is an ORM library, that is able to execute SQL queries based on calls to the entities defined in the application. This greatly helps as it abstracts and automates work with databases. Gorm is also used during development to migrate the schema if it was recenty changed. Echo[12] is the second library, which is built on top of the standard golang http package[13]. It has a support for building (Representational state

[11] https://github.com/jinzhu/gorm
[12] https://github.com/labstack/echo
[13] https://golang.org/pkg/net/http/

transfer) REST APIs. Echo supports so-called middlewares. A middleware is a function called on every request in the specified group. A middleware is commonly used for the request logging or authorization.

Gateway Factory server handles a creation of new devices by the HTTP request, which should be called by a factory script. It updates a database with the given information from a device and creates a new certificate. Every new device needs to have a unique ID with the length of 16 digits. The first digit is always left 1 to indicate a gateway device. Next 14 digits are randomly generated and the last digit is calculated by the Damm table lookup algorithm[14]. If the generated number does not already exist in a database, it is registered, certificate for the device is issued and the id with the issued certificate is sent back to the device in an HTTP response.

The Damm algorithm showed on a figure 6.3 was chosen because of its simple implementation and a low performance requirements.

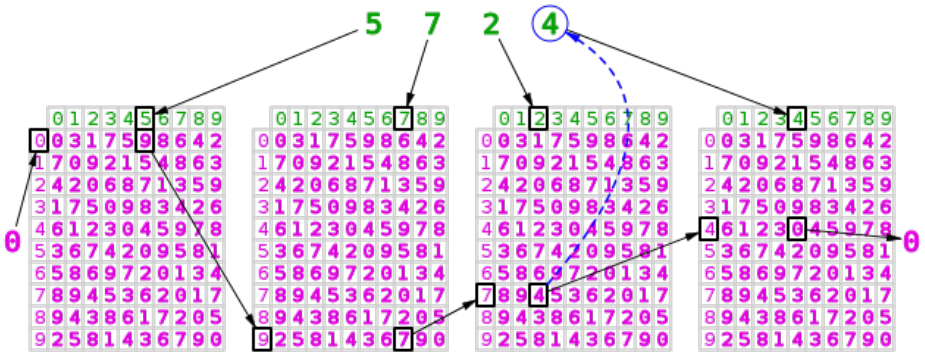


Figure 6.3: Example usage of the Damm lookup algorithm. The algorithm iterates over a given number, looking up the next step in the table. After the lookup is done, one number is chosen as a check number.

Source: https://en.wikipedia.org/wiki/Damm_algorithm

Listing 6.3: A result of iteration of Damm algorithm is given by this formula. Table is a 2D array of values from 0 to 9 as shown on figure 6.3. The value of digit is the current digit of an iterated number (green on figure 6.3 and interim is a current value of the result (violet on figure 6.3). The value of interim is usually initialized to 0.

```
interim = Table[interim][digit]
```

6.4 Factory script

The factory script is executed on a device before a device that should run the gateway software is used for the first time. Factory script prepares an environment of the gateway device to be able to correctly run the gateway software.

The factory script is written in the Python language for the simplicity. It retrieves a MAC address as well as a security ID from a processor of a device. These two values uniquely identify the device. Every device needs to have a unique ID as well. A unique ID

[14] https://en.wikiversity.org/wiki/Damm_algorithm

is generated by the Gateway Factory server. A MAC address of a device and security ID is sent to the server and the server replies with a unique ID. After the ID is returned, it is written to the device's persistent EEPROM memory.

A device must be able to connect to the private network after it has been initialized, thus the factory script saves an SSL certificate which was retrieved from the Gateway Factory server to the device file system (FS). This stage of a device preparation takes place in a private network, to prevent a misuse of data like MAC address, security ID or SSL certificate.

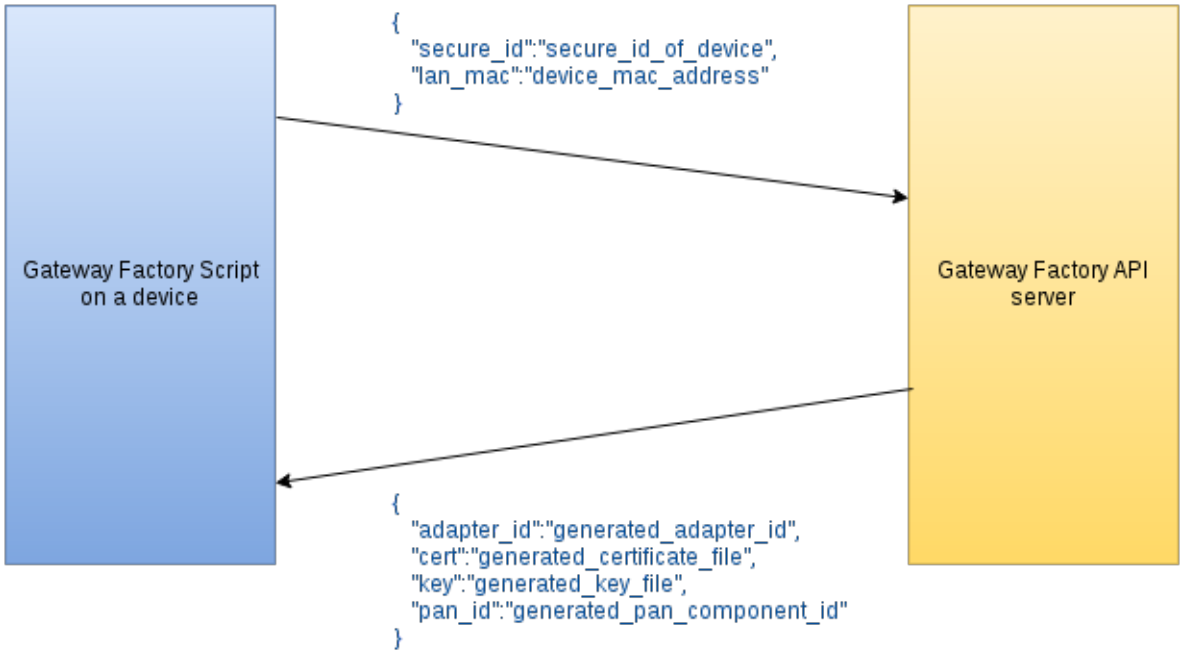


Figure 6.4: HTTP Communication between the factory script and the factory server necessary for the device id and certificate retrieval.

6.4.1 Gateway Manager core

The Gateway Manager core[15] is a library written in the C++ language. It consists of common modules for the Gateway Manager server and the Gateway Manager client. The library provides an implementation of the protocol specific behavior like serialization and event handling, which is the same in both, client[16] and server[17] implementations.

The library implements message handlers for each part of the communication protocol. The communication protocol is divided into 3 smaller protocols: standard, authentication and update (shown in the figure 6.5). A protocol buffers file exists for each protocol, defining the messages that can be send and received by the protocol. When the library is compiled, the protocol buffers files are compiled by the protocol buffers compiler and generate C++ sources, which are afterwards compiled by the standard C++ compiler.

[15] https://github.com/BeeeOn/gateway-man-core
[16] https://github.com/BeeeOn/gateway-man-client
[17] https://github.com/BeeeOn/gateway-man-server

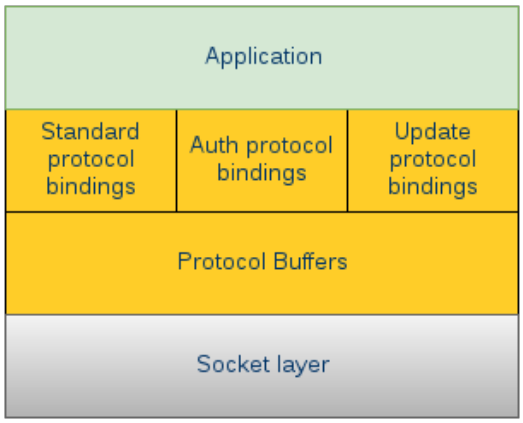


Figure 6.5: Layers of the Gateway Manager core architecture

Standard protocol defines a basic set of informational and command messages. Ping message is one of the well known messages. The Ping message stores the text that should be sent back by the opposite device. This message is mostly sent to keep alive the communication when a timeout occurs on a device. Protocol timeouts may occur when one or the other side is not responding for a given time(usually 250ms). The Report message is an informational message which carries a priority of the message (also known as level) and a message itself. This message should be mostly used when an unexpected event occurs on one of devices. Two protocol control messages are used when errors occur within the protocol: Reset and Restart. The Reset requests a soft-restart by only cleaning buffers and cache of the protocol, the Restart message is sent when the connection should be broken and the devices should reconnect. Two configuration messages are also present in the protocol for getting and setting of the configuration key-value pairs known to an opposite side.

Authentication protocol uses only two messages. The Authentication Request message and the Authentication Response. Authentication Request also has an optional integer value for the wait danger. This value can be set by a server to inform the opposite side it could wait for a certain time in a queue and may be disconnected during the wait. An opposite device should reconnect in a time set by the wait danger if disconnected. The wait danger field is an experimental field that may be set during the performance peaks, possible cyber-attacks or other unexpected events. The authentication response message carries the usual authentication data: id and password.

Update protocol defines two messages: Execute and Execute response. The execute message holds a description of what should be executed and a content(script) that should be executed by the system. The content field may be a Python script, Bash script or anything that is executable by an opposite device. When a device executes the script, the Execute Response message is sent back. It carries the return code of the executed content, a standard output and a standard err output.

Implementation of the gateway manager core library heavily depends on the C++ template system ducktyping. It implements factory methods and message dispatchers for all three sub-protocols. Message dispatcher is given an object that implements the handlers for the protocol that is currently being dispatched. This calls the factory method first, which may decode a message. However if the message could not be decoded, the dispatcher returns and control is given to an other dispatcher. If none of the dispatchers could recognize

the message, it is thrown away.

The underlying implementation on a socket layer is using the length prefix method shown on the figure 6.6 to distinguish between the contents of the packets that are being received. This method prefixes each message by an integer value that is set to the total message length.

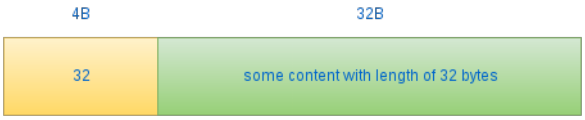


Figure 6.6: A message using the length prefix method with a content of 32 bytes, that is prefixed by the 4 bytes with the value of the message length.

6.5 Gateway Manager server

The Gateway Manager server is built on top of the Gateway Manager core library. Its main purpose is to handle Gateway Manager clients when they connect to it. It needs a PostgreSQL database instance to properly work, as it stores all information in the database.

A database schema was created for the purpose of storing the data of the Gateway Manager server. The schema 6.7 involves five entities. The main entity of the schema is *adapter*. This entity describes a single device and stores the main information such as id, security id or MAC address. It also has an informational owner field that describes who owns the device[18]. This entity also stores data about the components of the device, but they are not directly affecting the Gateway Manager server. The next important entity is *adapter_state*. This entity stores data about the device's current state. The state of a device may change frequently. It has information about the last connection, current and last stable version of a system on a device and its OpenVPN IP address. This entity is weak and only exists it there is an adapter entity present.

The last three entities are used for the purpose of remote execution. The *package* entity saves data about a single file (usually zip), that can be sent to a device for the purpose of remote execution. This entity stores necessary information about the script such as purpose, level of priority (in a case more packages are tagged for execution) or a version that the package supports. The version can be written with the wildcards, e.g. *1.x* means the package can be executed on any device with a version of 1 and any subversion. The *adapter_package* is a temporary store for the packages that should be executed on a given device. Once the package is executed, the record is removed from the database and the result is moved into the *package_result* table. The adapter package entity thus defines a package that should be executed on a device, the purpose of execution and the force flag. The force flag tells the server if it should execute the package even if there are issues present on a device. The *package_result* entity holds data about a package execution, its return code and logs.

A Data Access Object (DAO) pattern is used to access the underlying database. This pattern is widely used to separate the low level data access logic from the business logic [17]. DAOs were created for the purpose of operating with the database entities. Every

[18]The owner field of a device is mainly used during the development to distinguish devices.

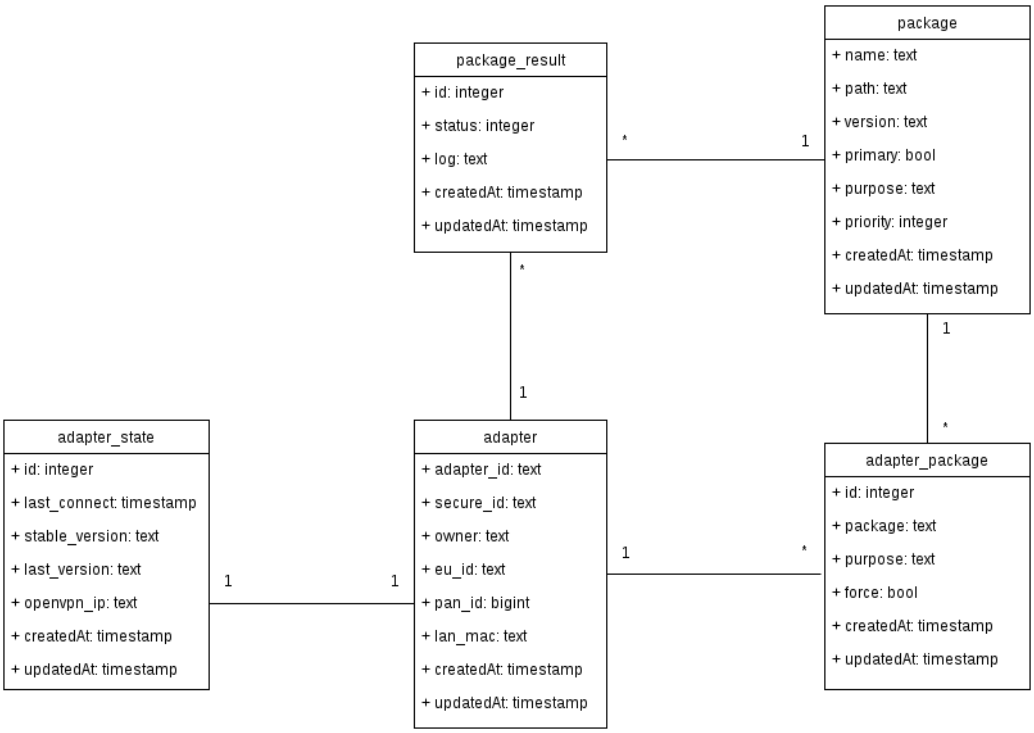


Figure 6.7: ER diagram for the database schema used by the server side of Gateway Manager

DAO is created as a singleton class. This is due to the fact that only a single data access point is needed for every given entity.

The Gateway Manager server accepts TCP connections from devices running the Gateway Manager client. Every device must authenticate right after it connects. The server then updates the state of the device in the database and checks for the packages that should be sent. If no packages are present in the database, the server closes a connection with the device by the Exit packet. The Exit packet tells the device when it should reconnect again. In a case where there are packages to be executed, they are sequentially sent to a device for execution and the results are saved into the database.

6.6 Gateway Manager client

The Gateway Manager client is built on top of the Gateway Manager core library. Its implementation takes advantage mainly of the standard C++ language, the Poco library and the Gateway Manager core library.

The client connects to the Gateway Manager server periodically. This allows the server to check for updates and update the status of a device. The implementation of communication consists of protocol handlers. Each handler method processes one message from the protocol. The client is only responding to requests from the server most of the time. The only message that can be sent without a previous request is the Report. This guarantees that the process is controlled by the server.

The client implements a configuration map that allows an application to register configuration getters. A configuration getter is a function that returns a configuration value

when called. This allows an application to dynamically register the wanted configuration based on a previous configuration, device type or an environment. The configuration getter primarily takes care of:

- loading from different types of configuration storage like EEPROM, file system or memory
- production and debug environment configurations

The Gateway Manager client is managed by a SystemD service. A SystemD unit file is configured to always restart the client after 10 seconds if an issue is detected. This unit file is distributed directly with the client.

6.6.1 Configuration and logging

An external INI[19] configuration file allows to configure the client if necessary. The file shown on the listing 6.4 defines the host and port that should be used as a target Gateway Manager server. It also defines a log level. There are 8 different log levels defined by the Poco library:

- fatal - usually followed by an application tear down
- critical - application is usually able to recover from an event
- error - often used to inform about failure of an important process
- warning - used to warn about possible failures
- notice - may be used when an unexpected event occurs
- informational
- debug
- trace

Poco allows to use so called channels. A channel is responsible for delivering messages (shown in the figure 6.8). A channel has an attached destination. A destination can be a file, a TCP connection or a console.

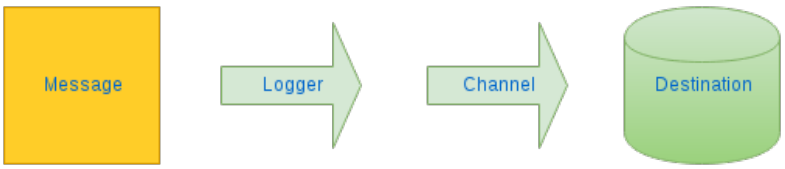


Figure 6.8: A message travels through the logger and defined channel. A channel is attached to a destination. This is where the message is written.

The log levels below notice level are only informational and may contain a lot of information. They need to be properly set up as log size may increase rapidly. A log level is usually set to trace or debug while in development. The warning or error levels are used when in a production environment.

[19] https://msdn.microsoft.com/en-us/library/windows/desktop/ms717987(v=vs.85).aspx

Listing 6.4: An example configuration file for the Gateway Manager client

```
[connection]
host=localhost
port=8877

[log]
level=trace
```

The requirement of the configurable log level is based on a previous production experience. Production log level defaults to warning level. The client may run in a production environment for days or weeks without an issue. However if an issue occurs, it is necessary to have the device start monitoring with a lower log level, e.g. debug.

6.6.2 Execution and update

A process of package execution and update is handled by the Update protocol message handlers. This process is initiated by receiving the Execute message from the server. The message describes what should be changed and carries the script that will be executed.

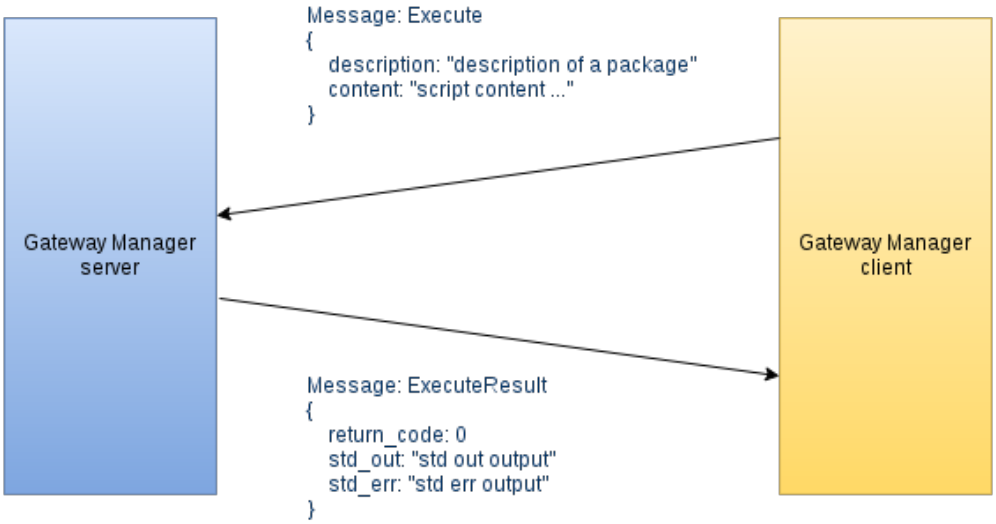


Figure 6.9: Only two messages are exchanged during the communication that handles execution. The incoming message includes the description and content of the script. The outgoing message contains the results.

The process is shown in the figure 6.9. The Execute request message is sent by the server. The client executes a given content and responds back with the Execute Result message. The Execute Result message contains logs from the run and a return code. The opkg package manager is often used to easily update existing packages or install new ones.

Chapter 7

Applicability of the Gateway Manager system

This section covers usage possibilities of the Gateway Manager system. There are two main scenarios for the system to be applied in. The first scenario is configuration reading. This allows a server to read desired configuration from a device. The second scenario is script execution. The script execution on a device allows the system to primarily:

- access a file system
- access already installed applications and an underlying OS
- change configurations, repair an environment
- install or update new applications

7.1 Configuration reading

The Gateway Manager client allows to define a configuration map that can be dynamically adjustable. This means, new configuration mappings can be added during the runtime if needed.

A simple experiment of a configuration retrieval is shown by the log 7.1. The client is running in the OpenVPN network on IP 10.1.0.103. The system running on a device has version 0.6. There are three main parts that are shown in the log 7.1. The first one is the report saying the device successfully authenticated to the server. The second part consists of GetConfigRequests, where a dynamic map is used to retrieve the configuration values from the device. The last part is a Restart message. Once the server finishes the work on the device, it sends the Restart message. The application shuts down and will be restarted by the SystemD service in a configured amount of time.

Listing 7.1: A log that shows the process of configuration retrieval on one of debug devices

```
[Ping] Dispatched, query [Hello]
[Report] Dispatched
Authentication: Successfully authenticated
[GetConfigRequest] Dispatched
[GetConfigRequest] Value for [openvpn_ip] was requested.
[GetConfigRequest] Found [openvpn_ip:10.1.0.103]
[GetConfigRequest] Value for [beeestro_version] was requested.
```

```
[GetConfigRequest] Found [beeestro_version:0.6]
[Restart] Dispatched
```

The server handles ale responses sent by the connected client as shown in the listing 7.2. It receives the configuration sent by the client and shuts down the communication.

Listing 7.2: A log that shows the process of configuration retrieval on the server side

```
[unk->Ping] Dispatched, query: [Hello]
[Auth] Handling for the adapter: 1737240720266367
[State@1737240720266367] Updating last connection status
[1737240720266367->HandleGetConfigResponse] Dispatched
[1737240720266367->HandleGetConfigResponse
Received configuration openvpn_ip:10.1.0.103
[1737240720266367->HandleGetConfigResponse]
Received configuration beeestro_version:0.6
[Shutdown@1737240720266367]
```

7.2 Simple update script execution

A device can be updated using the Execute message from the update protocol. This example of logs 7.3 and 7.4 shows what is happening during the execution of a simple opkg update script.

The log 7.3 shows a successful authentication of the client. The execution of the script happens next and the results are sent back to the server.

Listing 7.3: A log that shows the script execution log on a device

```
[Ping] Dispatched, query [Hello]
[Report] Dispatched
Authentication: Successfully authenticated
[Execute] Dispatched, Description: Package update via opkg package manager
[Execute] Temp file created on the location: /tmp/tmp24105aaaaaa
[Execute] Done, sending results
[Restart] Dispatched
```

The server log 7.4 shows the process of execution handled by the server. First, the device is authenticated. The Execute message is sent next and the ExecuteResult message is received once the script has been executed on the device. The standard output and standard error logs are returned back to the server.

Listing 7.4: A log that shows the process of a script execution on a device from the perspective of server

```
[unk->Ping] Dispatched, query: [Hello]
[Auth] Handling for the adapter: 1737240720266367
[State@1737240720266367] Updating last connection status
[1737240720266367->ExecuteResult] Dispatched
[1737240720266367->ExecuteResult] return code: 0
[1737240720266367->ExecuteResult] std_out:
Downloading http://cloud.beeeon.com/feed_jethro//all/Packages.gz.
Updated source 'remote-beestro-all'.
Downloading http://cloud.beeeon.com/feed_jethro//armv7a-vfp-neon/Packages.gz
Updated source 'remote-beestro-armv7a-vfp-neon'.
Downloading http://cloud.beeeon.com/feed_jethro//olinuxino_a10lime/Packages.
    gz
Updated source 'remote-beestro-olinuxino_a10lime'.
```

```
[1737240720266367->ExecuteResult] std_err:
[Shutdown@1737240720266367]
```

7.3 Possible extensions of the Gateway Manager system

There are two main methods that can be used to manipulate a device. The first is a configuration mapping. The second is a script execution. The configuration mapping system works with already defined functions compiled in the client's binary. However, the script execution system is able to execute scripts that are defined by the user. This allows the server to execute scripts e.g. to:

- repair an environment of a device
- gather data provided by a device
- change configurations that are not controller by the mapping

Considering the user experience, there is a possibility to create a web interface. This interface should be able to communicate with a server that would access the system's database. It should allow a user of the system to execute necessary scripts on selected devices.

7.4 Conclusion on the system applicability

Two scenarios of the Gateway Manager system were introducted in the sections 7.1 Configuration reading and 7.2 Simple update script execution. However a number of scenarios may greatly vary based on requirements and a deployment environment. There are many possibilities for the configuration mapping system. It can be used in specific situations like status tracking for the subsystems. Some of the possible extensions were then introduced. These extensions are focused mainly on the options of script execution and user experience.

The testing scenarios were executed on the debug server and device used within the BeeeOn system. The system testing started on May 5. 2016 and the scenarios were executed on May 16. 2016. No issues regarding the system were detected during the test run.

Chapter 8

Conclusion

The goal of this work was to implement, test and discuss a solution that is able to track and update embedded devices remotely. There are many existing solutions that can do so locally. The implementation of the system focused on their usage and distribution of information they provide. There are two implemented methods to manipulate a client running on a device. The first method is able to retrieve a configuration of a device. The second method allows to execute scripts and get back their results. Both of these methods are automatically used by the implemented server when necessary. The A10-OLinuXino-LIME embedded device was used during the 10 day testing. No fatal issues were detected during the test run. Minor adjustments to the configuration mapping were made.

A core library that unifies the common behavior of the server and the client was implemented. The library implements the protocol between the server and the client. Specific message handlers were then implemented for the server and client separately.

There are many possible extensions to the system. They include automatic issue detection and repairing or a web interface. A web interface would allow more users to easily use the system without deep background knowledge.

The system is meant to be used on the BeeeOn project within the smart home ecosystem. It allows an automatic update and monitoring of deployed devices. An update of the devices was originally done manually. This system allows to execute the update scripts en masse and remotely even on devices without a public IP address.

Bibliography

[1] Clearkimura (askubuntu user). Syslinux vs grub. http://askubuntu.com/questions/651902/what-is-the-difference-between-grub-and-syslinux, 2015. [Online; visited 9.5. 2016].

[2] BeeeOn. Beeeon: About. https://beeeon.org/domains/beeeon.org//index.php?title=Main_Page, 2016. [Online; visited 16.5.2016].

[3] Alex Da Costa. Grub2. https://help.ubuntu.com/community/Grub2, 2016. [Online; visited 9.5. 2016].

[4] CZ.NIC. Project turris. https://www.turris.cz/en/, 2016. [Online; visited 9.5. 2016].

[5] Git. Git. https://git-scm.com/, 2016. [Online; visited 16.5. 2016].

[6] Applied Informatics Software Engineering GmbH. Poco. http://pocoproject.org/features.html, 2006-2016. [Online; visited 9.5. 2016].

[7] GNU. Bash reference manual. https://www.gnu.org/software/bash/manual/bash.html#What-is-Bash_003f, 2014. [Online; visited 9.5. 2016].

[8] Google. Protocol buffers. https://developers.google.com/protocol-buffers, 2016. [Online; visited 9.5. 2016].

[9] Google. Protocol buffers developer guide - how do they work? https://developers.google.com/protocol-buffers/docs/overview#how-do-they-work, 2016. [Online; visited 10.5.2016].

[10] Google. Protocol buffers language guide (proto3) - packages. https://developers.google.com/protocol-buffers/docs/proto3#packages, 2016. [Online; visited 10.5.2016].

[11] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. LeeSeshia.org, 2011. Available online at http://leeseshia.org/releases/LeeSeshia_DigitalV1_08.pdf.

[12] Maciej Sobczak Mateusz Loskot, Vadim Zeitlin and Stephen Hutton. Soci - the c++ database access library. http://soci.sourceforge.net/doc/3.2/rationale.html, 2013. [Online; visited 9.5. 2016].

[13] Olimex. A10-olinuxino-lime specification. https://www.olimex.com/Products/OLinuXino/A10/A10-OLinuXino-LIME/open-source-hardware, 2016. [Online; visited 9.5. 2016].

[14] Margaret Rouse. Embedded system. http://internetofthingsagenda.techtarget.com/definition/embedded-system, May 2009. [Online; visited 9.5. 2016].

[15] The Lua team. Lua: Getting started. https://www.lua.org/start.html, 2016. [Online; visited 9.5. 2016].

[16] The RedBoot team. Lua: Getting started. https://sourceware.org/redboot/, 2016. [Online; visited 8.5. 2016].

[17] Tutorialspoint. Data access object pattern. http://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm. [Online; visited 11.5. 2016].

[18] Rob van der Meulen. What is the difference between fpga and asic? http://www.gartner.com/newsroom/id/3165317, 2007. [Online; visited 14.5.2016].

Appendix A

Content of the CD

- /BP.pdf - Bachelor's thesis text
- /BP/src - directory that contains the source text of bachelor's thesis
- /gateway-manager - directory that contains source files for the Gateway Manager project