MASTER'S THESIS | LUND UNIVERSITY 2016

Automatic dynamic updating of the PalCom middleware for Internet of Things

Christian Hernvall

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-09

Automatic dynamic updating of the PalCom middleware for Internet of Things

(Master's thesis)

Christian Hernvall
christian.hernvall@gmail.com

April 19, 2016

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Boris Magnusson, Boris.Magnusson@cs.lth.se

Examiner: Görel Hedin, Gorel.Hedin@cs.lth.se

Abstract

The number of devices connected to the Internet is rapidly increasing. Especially the number of Internet of Things devices. These devices contain software which, like any other software, needs to be updated. With the great amount of devices to update, this calls for a manageable updating solution which does not depend on user interaction. In this thesis we present a solution to dynamically update an Internet of Things system, by automatically updating its devices. Our solution is based on and implemented for the Internet of Things system PalCom.

Keywords: internet of things, palcom, automatic software updating, dynamic software updating, software update management

Acknowledgements

I would like to acknowledge Björn, Boris, Mattias and Mia for their valuable input when brainstorming solutions. I would also like to acknowledge Mattias, Mia and Gunnar for their help with PalCom related tips and troubleshooting.

Finally I would like to thank all the people around me who have been forced to listen to me talking about this interesting subject too many times, and often contributed as well. Most of all Carolina.

Contents

Chapter 1

Introduction

Internet of Things, often referred to by its acronym IoT, is a term and vision with no single definition, but generally describes how 'things' all around us gets connected. These 'things' are physical objects which by themselves or with the help of another device, are able to generate, consume or exchange data with each other or other devices. All of this happens without the need for human intervention. A key point in the Internet of Things is that these 'things' are not necessarily ordinary computers but can be any everyday object such as a freezer, a cow or a pair of shoes. Any data-carrying, data-capturing, sensing, actuating or processing device can connect itself or a 'thing'. The device can be connected through any type of communication medium, for example Bluetooth, WiFi or both. The network to which these 'things' are connected can look very different. Some 'things' are indeed directly connected to *the* Internet, some only communicate inside a Local Area Network, some only communicate with a single device and so on [30]. Some examples of Internet of Things devices include embedded systems in our home devices, e.g. monitoring the freezer or switching lights in different rooms, the so called smart home [26]. Another example is smart bins and smart traffic lights in a city reacting to their environment and sending useful data about when to empty the trash, or if a light bulb needs changing [4].

The number of connected devices in the world has been predicted to increase at a high rate. Ericsson predicted that there will be 28 billion connected devices by year 2021, of which 15.3 billion are Machine-to-Machine and consumer electronics, in their mobility report from november 2015 [7]. This is a big increase from today's 15 billion connected devices, of which 4.6 billion are Machine-to-Machine and consumer electronics. Cisco made a prediction as well in 2011, saying that there will be 50 billion connected devices by 2020 [9], and Gartner stated in a 2015 press release that they estimated 6.4 billion connected Internet of Things 'things' in use by 2016, growing to 21 billion by 2020 [28]. The

exact numbers differ depending on source, but they all agree that the number of connected devices, most specifically Internet of Things devices, is rapidly increasing.

Internet of Things devices do not necessarily have a traditional graphical user interface. They may be headless systems, which operates without a monitor, keyboard, buttons or other peripheral devices. It can also be the case that the user is not even aware that the system exists, as long as it is working. No matter which category the device falls into, it will need to be updated in order to fix bugs or to release new features. But with the obscurity of interaction with a general device coupled with the large and increasing number of connected 'things', we see that the traditional way of updating devices, which more or less requires user interaction and intervention, is not a manageable solution for Internet of Things systems in the long run.

We need a way to update Internet of Things systems that is *automatic* and *dynamic*. With automatic, we mean that the updating process takes place without requiring user interaction. With dynamic, we mean that the updating process does not disturb or break the services which the system provides or interacts with. This master's thesis examine ways to achieve this, proposes a solution and implements it for PalCom [10]; an Internet of Things system developed at the Dept. of Computer Science at Lund University.

In this master's thesis we examine different ways of achieving automatic dynamic software updating of Internet of Things systems, but we are not looking for a general solution to the problem which is applicable to all possible Internet of Things systems and contexts. Instead, we study the specific case of updating PalCom devices in a PalCom system. If we break down the 'Internet of Things'; PalCom devices resembles the 'things' and the PalCom system, the network and distributed system which these devices are parts of, resembles the 'internet'.

PalCom devices are software, which means that there can be several PalCom devices running on the same hardware unit. Every type of PalCom device is built upon a core, the Palcom kernel, working as an execution environment to which it is possible to add or remove parts, called services and assemblies, dynamically in order to extend the device with features and functionality. The versioning and updating of services and assemblies are completely separated from the versioning and updating of the devices they are running on. This master's thesis does not study the problem of updating the extension parts, services and assemblies, since that subject has already been studied and implemented in PalCom in a previous master's thesis [21]. Instead, we study the problem of updating the PalCom device kernel. The reasons to update the kernel include fixing bugs and adding new features, for example adding support for Bluetooth as another possible PalCom communication medium. We also study the problem of updating these devices given the fact that they are part of a distributed system of other devices, their PalCom system.

We can break down this problem statement into the following questions that this thesis answers:

- What are the important requirements when performing automatic dynamic updating of the PalCom middleware?
- How can we perform automatic dynamic updating of the PalCom middleware?

To answer these questions, we analyze our Internet of Things system to update, PalCom, as well as other related systems and techniques. The related work we study includes dynamic updating solutions such as Kitsune [14], Upstart [1] and OSGi [2], as well as automatic updating solutions such as Android for Work [12] and MobiCare [6]. From our analysis we extract important requirements, from which we design and propose a solution that dynamically and automatically updates the Internet of Things middleware PalCom. As a proof-of-concept, we built a functional implementation of our proposed solution. The implementation is tested to benchmark our solution in regards to device downtime and quickness of the updating process.

Chapter 2

Motivating example

To get a clear picture of what type of system we are working with and why we need a solution, we dedicate this chapter to illustrate a concrete real world scenario where this master's thesis is applicable.

In a health care project, Android smartpads with PalCom is used by ambulance personnel in order to collect values and establish communication to describe the patient and its symptoms to doctors before arriving to the hospital (see figure 2.1). This way, it may be possible to prepare for faster and better treatment of the patient. At the time of writing, there are about 60 smartpads distributed to ambulance personnel.

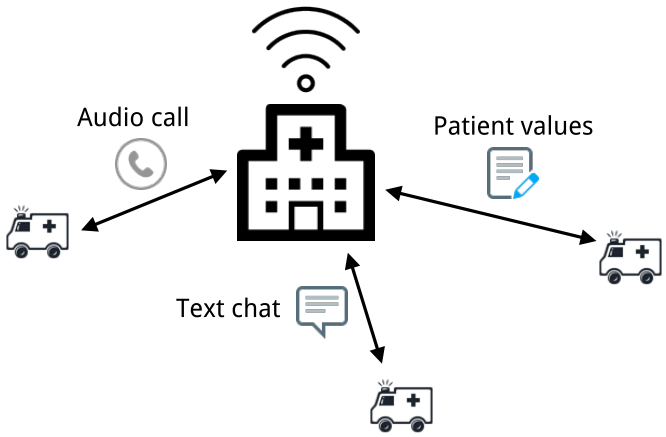

Figure 2.1: The ambulance scenario. Ambulance personnel use smartpads with PalCom to communicate with doctors at the hospital.

In this scenario, we take a closer look at the communication between hospital and ambulance. We refer to figure 2.2. The ambulance personnel use a smartpad, which has a PalCom device installed and running. On the PalCom device, there is a service running called AmbulanceService, which gives the ambulance personnel a GUI to enter patient information, initiate audio communication, etc. This service is also responsible for communicating this information with the HospitalService which is running on a PalCom device on a doctor's desktop at the hospital. HospitalService gives functionality similar to AmbulanceService. In reality, the functionality of AmbulanceService and HospitalService may be spread out on many services. For example one service handling a GUI, one service handling the audio communication, and so on. For the sake of simplicity we only have one service handling all functionality on each PalCom device in this example.

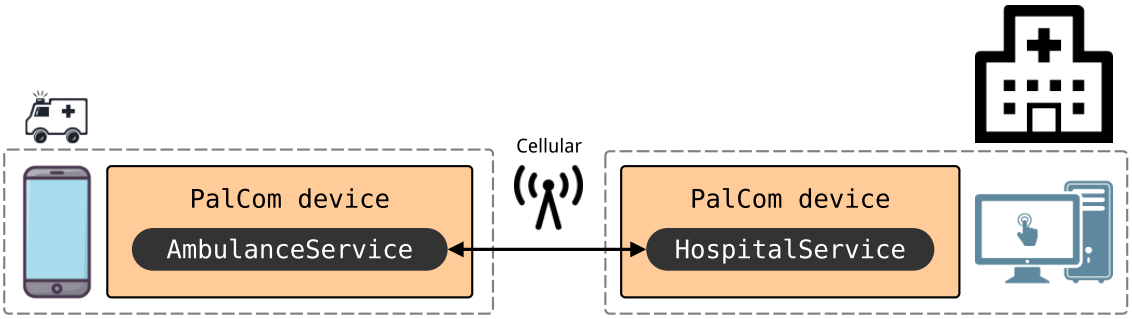

Figure 2.2: Architecture diagram describing communication between ambulance personnel and hospital personnel through PalCom devices and services.

Say that we want to bring more functionality to this scenario. The ambulance personnel like the way they can send patient values, chat via text and make audio calls to each other, but would also like to be able to communicate via video calls in order to describe the situation better. The ambulance personnel would also like to use a PalCom service on their new health watches, that they attach on the patients wrist, so the watches can collect health information and send it live to the hospital. The problem is that the watches only communicate via Bluetooth, and Bluetooth is not supported by PalCom devices as a communication medium. It is the PalCom device that makes it possible for its services to communicate with services on other PalCom devices. There are also some bug fixes to HospitalService and to PalCom device which we want to ship with the same update. Summarized, we want to do the following:

- Update AmbulanceService to support video calls.
- Update HospitalService to support video calls and fix some bugs.
- Update PalCom device to support Bluetooth as a communication medium and fix some bugs.
- Write a new service, let us call it HealthWatchService, which we install on the PalCom devices on the health watches.

We want to end up with the system in figure 2.3.

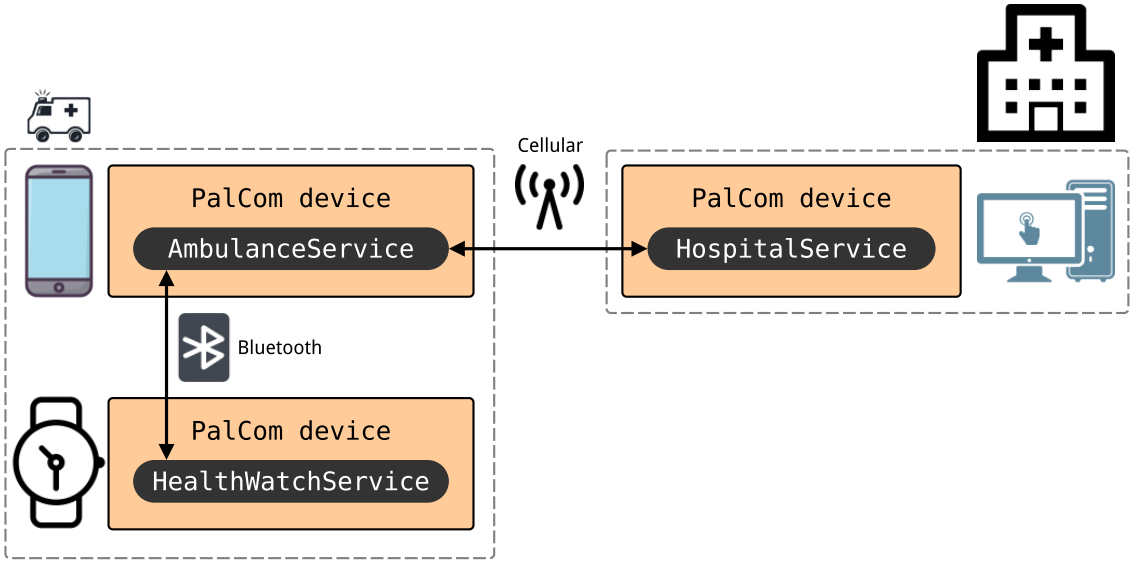

Figure 2.3: Architecture diagram describing communication between ambulance personnel and hospital personnel through PalCom devices and services in the updated version.

Now that we have our new design ready it is time to perform the updates. Thanks to a previous master's thesis [21], it is already possible to automatically update services, so we use that solution to update AmbulanceService and HospitalService. We also write our new service HealthWatchService which also uses ideas in [21] to deploy it on our new health watches. But when it comes to the PalCom devices, there is no way to update them except by manually shutting down the device, replacing the executable with a new version executable, and restarting the device. This has to be done for each PalCom device in the system. It is this problem that our work is addressing: automatically updating PalCom devices in a PalCom system.

Chapter 3

Approach

3.1 Method

The method of this thesis consisted of literature studies of scientific papers and other related projects to find out if existing solutions or methods could be used as a solution or as part of a solution to the problem statement. The PalCom system was also studied, with literature studies of scientific papers, and through informal conversations with PalCom developers at the Department of Computer Science at Lund University.

As a testcase during the thesis work, a proof-of-concept for the proposed solution, as well as a way to get to know the PalCom system in more detail, we worked on a couple of implementations: two pilot projects with only simple functionality and a final implementation of the proposed solution. These implementations were developed in an iterative manner, making it possible to adjust to new requirements and needs during the life of the thesis.

3.2 Technical background

3.2.1 Dynamic software updating

Dynamic software updating, or briefly dynamic updating, is the process of updating a running system without interrupting its execution [16] or without interrupting its service [27]. This is in contrast to traditional updating of programs where the whole program is restarted in order to load the new version.

3.2.2 Automatic software updating

In this thesis, automatic software updating, or briefly automatic updating, refer to the process of *fully* automatic software updating, which means that everything in the updating process happens automatically, with no human interaction from the point in time that an update is made available.

Automatic software updating has no necessary connection to dynamic software updating. It is possible to have an automatic dynamic software updating process as well as a non- or semi-automatic dynamic software updating process.

3.2.3 PalCom

PalCom is an Internet of Things middleware, currently developed at the Department of Computer Science at Lund University. The PalCom project originates from the EU project Palpable Computing [8]. We begin by explaining some terminology:

Unit. A unit refers to a machine running one or more PalCom devices. The unit can for example be a laptop, a smartphone, a server or an embedded computer in a lamp, cow or any other thing. We use this term in order to avoid confusion between physical and virtual devices. PalCom devices are software, so they are always virtual devices. PalCom devices run on physical devices, which we call units.

Now that we have the terminology in place, we explain some general concepts which PalCom is built upon [10]:

Devices. A device represents a running instance of the PalCom kernel. There are many different device types available. All of them uses the same PalCom kernel, but can otherwise differ from one another. Every device is identified by a unique ID, called device ID, which the device broadcasts to other devices on the network. Normally, there is one PalCom device running per unit, but there could be any number of devices running on the same unit. Running many devices on the same unit is practical for testing purposes and/or when there is a need for different device types on the same unit. Devices offer no functionality. This is offered by *services* and *assemblies*, which are described below. A device provides an execution environment for services and assemblies, the ability for them to announce themselves on the network and the ability for them to discover other services and assemblies present on other devices. The protocols used for this are specified in the PalCom kernel.

Services. A service offers functionality. The functionality can originate from the unit which the service's device is running on, such as a thermometer or camera, or could involve only some processing or communication of data. Services are self-describing which means that there is no need to know beforehand how to communicate with and how to use a service, because the service itself will tell you. This is achieved through the use of service descriptions, which lists what commands the service can send and receive. It is the service description that is announced on the network by the service's device, in order to be discovered by other devices. Any

number of services can be added to a device in a modular way. For example, a service on a device running on a unit with a camera could announce commands to take a picture or trigger the flash.

Assemblies. Services have no idea about other devices or services. For this there are assemblies that are used to connect, combine and create logic between services. An assembly is essentially a script defining *configuration* and *coordination*. Configuration describes what services on which devices to connect to. Coordination describes how incoming commands should be handled and how they trigger outgoing commands being sent to other devices. Assemblies can be added to a device in the same way as services. An example use of an assembly could be to link a motion detector to a door opener. A service connected to a motion detector announce commands when it detects motion. Another service connected to a door opener opens the door when it receives the "open door" command. To create our solution we make the assembly listen for commands from the motion detector. When the assembly receives the command, it sends the "open door" command to the door opener. For another good example of using PalCom devices with services and assemblies, with an illustrative walkthrough, see the article "Some like it hot: automating an electric kettle using PalCom" [18].

When practically using PalCom in an environment, a collection of useful tools utilizing the PalCom kernel has been developed to aid both during development and production. Some of these tools include:

Browser. The Browser is an interactive GUI-tool allowing users to explore devices, services and connections between services on the network. It is possible to interact directly with discovered services via dynamically created user interfaces in the Browser. An assembly editor and runner is included as well.

TheThing. TheThing is a generic application which announces itself as a PalCom device. One could argue that TheThing is the standard PalCom device type. Services and assemblies can be dynamically installed, started and removed, using the optional GUI or via a service running on TheThing. TheThing can execute on a desktop, an embedded system, on an Android device (using the fork TheAndroidThing), or any other system with a Java Virtual Machine (or to which PalCom has been ported to).

The reference implementation of the PalCom middleware is in the Java language, but there is also a minimal implementation in the C language [22]. PalCom is not restricted to Java or C, but can be implemented in other languages as well. In order to completely understand the update process, we also need to know how PalCom handles the configuration of devices, services and assemblies:

PalCom File System. More commonly referred to as PalCom FS, this is where all configuration files are stored. It is essentially a folder, called `PalcomFilesystem`, residing somewhere on the system (usually and by default in the user's home folder). The main folder is divided into the `devices` folder which contains device specific configurations, and the `global` folder which contains information and configurations which are not specifically tied to a single device (see figure 3.1).

In the `global` folder there is a folder for each device type on the unit. Inside a

device type folder there are files with device ID:s as names. Every file represents an available device configuration for that device type. The file contains a device ID and a device instance name. The device ID is used to uniquely identify the device and the instance name is a human-identifiable name.

In the `devices` folder there are configurations divided into a folder for each device on the unit, with the device ID as name. Inside a device specific folder there is a `.properties` file, a `services` folder and an `assemblies` folder. The `.properties`-file contains device specific properties and information about which services and assemblies to load. The `services` folder contains services and the `assemblies` folder contains assemblies.

When a device starts, it traverses the `global` folder, into the folder representing its device type. In the device type folder, the device can choose between different device configurations, which appears as files with device ID:s as names. When a device ID is chosen, the `devices` folder is traversed, into the folder representing the chosen device ID. In the device ID folder, the device reads the device properties from the `.properties` file. The services and assemblies specified in the properties will be loaded from the `services` and `assemblies` folders.

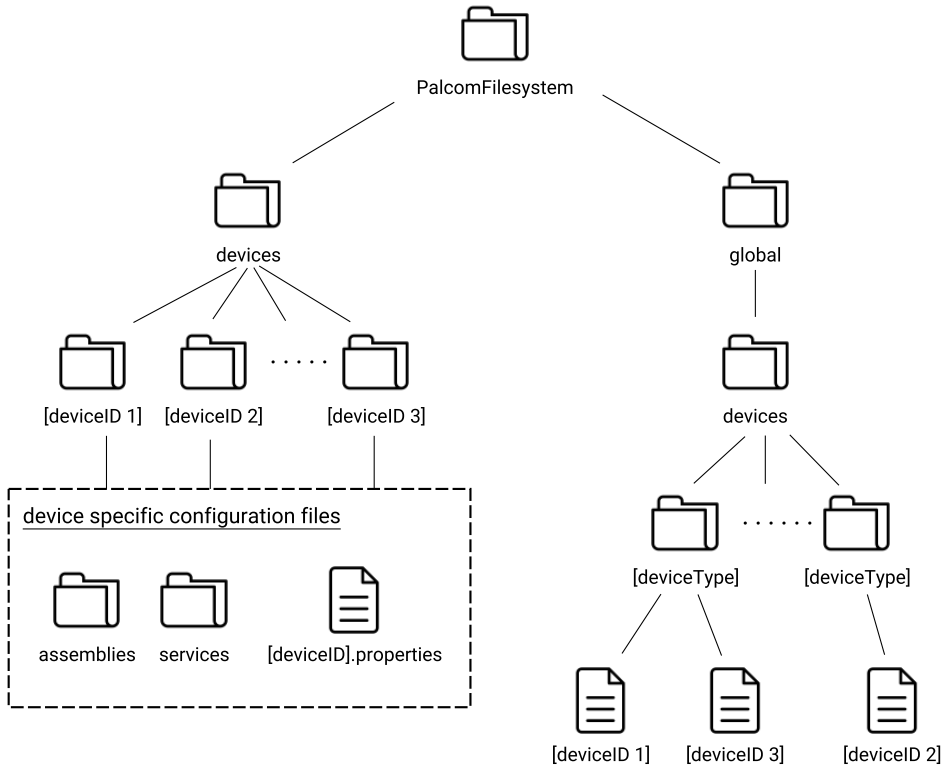

Figure 3.1: A graph showing the general structure of the PalCom filesystem. The [deviceID].properties file is the device specific main configuration file. The assemblies folder contains assemblies and the services folder contains services.

For a deeper introduction to PalCom, see the Introduction in David Svensson Fors' PhD thesis "Assemblies of Pervasive Services"[10], or for a more up-to-date overview, see the PalCom part of the Technical background in Mia Månsson's master's thesis "Dynamic installation and automatic update of Bluetooth low energy devices in Palcom"[21].

Chapter 4

Proposed solution

4.1 Selecting a solution

We will not study nor propose a solution that is best for Internet of Things systems in general. This is beyond this thesis's scope, and we believe that the best solution is largely dependent on the specific Internet of Things middleware used, because depending on the application and requirements of the system, the design of the updating process will be different. For example, an Internet of Things system which require its nodes to be online all the time needs a different updating solution than a system without that requirement. In this thesis we focus on the specific case of using PalCom.

4.1.1 Analyzing our system, environment and requirements

In this section we analyze our system and its environment, in order to know what is important and what is less important when searching for or designing an updating solution.

We are to update PalCom devices, running in an Internet of Things environment, which means:

- Different types of hardware
- Sometimes limited processing power
- Sometimes limited bandwidth and connectivity

When updating Internet of Things devices, we must obviously think about the updating of the individual devices, but we must also analyze how the updating affects the whole Internet of Things system of connected devices. There may be dependencies between devices, real-time requirements, uptime requirements, availability requirements etc. both for individual devices and/or for the whole system of connected devices. We need to understand the system as a whole as well as its parts in order to find a suitable updating solution. We can compare the updating of different parts of PalCom with the updating of the following systems:

- **Traditional program**. When updating, the program requires a restart in order to load the new version of code.
- **Real-time system**. Because of the real-time constraint on these systems we need specially tailored update solutions in order to update the systems while they are running, so called dynamic software updating. This generally includes some kind of hot-swapping of code, migrating states or running redundant hardware in order to always have a system up and running.
- **Distributed system**. A distributed system consists of multiple nodes, which individually can lose contact or go down. Despite this fact, the distributed system can be constructed in a robust way, so that the system itself is not affected by individual nodes being restarted or shut down. The Internet is a perfect example of such a robust distributed system. If one node go down, the Internet does not go down. If one path for delivering messages disappears, another one is used instead.

A PalCom system is a **distributed system**, where PalCom devices are its nodes. The PalCom system is designed to be robust despite devices going down, losing contact, being restarted, etc. All PalCom devices are aware that at any time, they can lose connectivity or contact with other devices. This means that when updating a PalCom device, we can treat it like a **traditional program** and simply restart it to load the new version. But when we consider updating of a PalCom system we treat it like a **real-time system** with a soft real-time constraint. It is not possible to stop the whole PalCom system at once and restart it in order to load a new version. We need to dynamically update the system by individually updating its parts, the nodes, while the system as a whole is still running.

When we talk about updating the PalCom system, we refer to updates that affects all PalCom devices in the system. Technically these updates includes all updates that changes the PalCom kernel, such as changing the API for services or supporting a new communications medium. A PalCom device specific update on the other hand, can for example be a GUI update for the specific device type TheThing. We divide PalCom system updates into two categories: **protocol-breaking** updates and non-protocol-breaking updates. A protocol-breaking update makes changes to the PalCom communication protocols, with the consequence that a device which have performed a protocol-changing update can not communicate with a device which have not performed the same protocol-changing update. If we have a PalCom system where some devices have performed a protocol-breaking update and some have not, we call it a mixed protocol system. Our updating solution must support performing protocol-breaking updates, but it is not the updating solution's job to solve the problem of mixed protocol systems. We get back to how to handle mixed protocol systems in the future work section 4.3.

In conclusion, we have two levels to take care of and treat differently in the updating process: the PalCom system and the PalCom devices. We see to dynamic software updating techniques in order to update the distributed PalCom system. For its parts, the PalCom devices, we treat them as traditional programs that can be restarted at any time.

When analyzing the PalCom project and how PalCom systems work, we can extract some important requirements and scenarios specific to PalCom that the solution must be able to handle, which answers the first of our questions of our problem statement in the introduction, chapter 1:

What are the important requirements when performing automatic dynamic updating of the PalCom middleware?

(A) **Pure Open Source PalCom code base.** The PalCom project does not want to include proprietary code, or to introduce significant dependencies on other code bases or projects. If a proprietary updating solution is used, it is not possible for PalCom developers to make changes, and if the makers of the updating solution makes some breaking changes to the updating solution it directly affects PalCom.

(B) **Unreliable connections.** Devices can go offline without warning, for example because of entering flight mode, shutting down, or simply crashing.

(C) **Unreliable connectivity.** Connectivity can be lost at any time, for example when entering a connectionless area.

(D) **Low bandwidth.** Depending on underlying connection type, for example LE Bluetooth or GPRS, the bandwidth can be very low.

(E) **Heterogenous PalCom implementations.** The updating process must not be platform or language dependent. It does not matter if the PalCom device is implemented in C or Java or any other language for the updating process to work.

(F) **Security.** The PalCom system can be handling sensitive information, such as personal data in health care. Therefore the updating solution must be designed with security in mind.

(G) **No user interaction.** From the point where an update is made available, the updating process must be fully automatic, requiring no user or admin interaction, unless a major error occurs. Due to the big number of devices and the obscurity of interaction with a general device, as explained in the introductory chapter (1), the solution becomes unmanageable if this requirement is not met.

(H) **Unexpected crashes.** Power failure, running out of memory, running out of disk space, and so on, must be handled without the PalCom system getting corrupt. From anywhere in the updating process, the system should always be able to return to a functioning state. This is important because PalCom devices may run on units which are very hard to manually recover.

(I) **Protocol breaking updates.** The updating solution must be able to perform updates that change communication protocols.

On the other hand, we can also extract some things which does not matter so much in PalCom:

(I) **Processing power.** Although the processing power can be low, the updating process does not need to be extremely lightweight in terms of processing power. It does not matter if the updating process is taking a long time to finish.

(II) **Uptime/Availability.** It is acceptable for the device that is being updated to go offline, be restarted or stopped for a limited time period. That devices can go down without warning, for example because of loss of connectivity or if they crash, is assumed and handled by all PalCom devices.

4.1.2 Existing solutions and related work

In order to arrive at the best possible solution for PalCom, we need to study existing solutions and related work. There could very well already be a solution out there that fits perfectly with what we want. And of course, on the contrary, it could be the case that no existing solution is matching what we seek due to some of the requirements we arrived at earlier. In the latter case we can at least hope to learn from the solution and extract something valuable in order to design a new solution for PalCom.

Even though a lot of solutions are language dependent, for example only Java or C(++), thus breaking requirement (E), they can still include useful and generally applicable concepts. If not, they may still be of interest when implementing the updating solution for PalCom in a specific language.

It is important that we remember the difference between dynamic software updating and automatic software updating and their respective application. We will first study dynamic updating solutions, followed by automatic updating solutions and finally solutions which can be put in both categories.

Dynamic updating

In this part we study existing solutions and related work focusing on, or closely linked to, dynamic software updating; the process of updating a running system without interrupting its execution or service. When looking at existing solutions, we must remember that the solution has to be applicable to a PalCom system in order to be relevant. It is not just PalCom devices that has to be updated individually, but a system of devices.

There are a lot of solutions which have a very low level approach, trying to solve the updating problem for programs and systems which are not allowed to ever go offline or have their execution suspended at all. However, because of our loosened requirement on availability and uptime (II), we do not need to take that approach. These solutions, even though they may boost performance, impose unnecessary complexity. All of them also fail to comply with requirements (A), Pure Open Source PalCom code base, and (E), Heterogenous PalCom implementations. Some of them include:

1. **A Technique for Dynamic Updating of Java Software, 2012 [23].** A solution for updating java code dynamically by using class wrappers. When a class is updated, deleted or modified, its class wrapper takes care of the administrative work.

2. **JavaDaptor, 2012 [25].** JavaDaptor is a Java specific solution using the JVM Tool Interface to replace old code with new code and remap all references to the new code.

3. **Partitioning of Java Applications to Support Dynamic Updates, 2014 [5].** As the title explains, this work proposes a way to partition Java applications in order to support dynamic updating.

4. **Upstare, Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction, 2009 [19].** Updates programs by using stack reconstruction. Very low level updating.

5. **Ksplice, 2009 [3].** Ksplice is a tool used to make dynamic updates for OS kernels. It analyzes the differences between the kernel to be updated and a traditional source code patch by comparing the compiled code, rather than source code. For example used to update the kernel in Red Hat Linux.

Some solutions use a state-transfer technique. Although the actual process can look very different, it typically works by starting an instance of the new version of the system, transferring all state information from the old version instance of the system to the new version instance, and finally putting the new version instance in charge. This idea could very naturally be used for PalCom systems because of the fact that all necessary states and settings are stored in the PalCom filesystem, and due to the loosened requirement on uptime (II), it is possible to perform a simple stop-and-update process for the devices. In the most simple scenario we could stop the old version devices (one-by-one or maybe some at a time), copy and/or transform the PalCom filesystem (the states) to the new version, and finally start the new version of devices. Some solutions that incorporates the state-transfer technique are listed below (6-8). Unfortunately also these solutions fall short on the Pure Open Source PalCom code base requirement (A) and the Heterogenous PalCom implementations requirement (E).

6. **Ekiden, 2011 [15].** Ekiden updates whole programs at once by state-transfer. It forks the new version of the program, serializes the old program state and transfers it to new version. A programmer must manually mark the states that should be transferred and mark update points in the code, where an update can occur. C language specific.

7. **Kitsune, 2012 [14].** Can be seen as a later version of Ekiden. As opposed to Ekiden, Kitsune uses dynamic linking to perform state-transfer. Also C language specific.

8. **Rubah, 2013 [24].** A state-transfer solution, similar to Kitsune, but for the Java language.

A slightly different way to approach the problem in the same spirit as above solutions is to first update the code in a stop-and-update way, and then when the new version of code is running, perform a lazy, on-demand, state-transfer, like the following solution:

9. **Javalus, 2014 [13].** Javalus works by suspending a program, updating the code and then continuing the execution. All states and objects are then updated on-demand. The solution is specific to Java and require a Java HotSpot VM.

Because of the fact that there exists a lot of dynamic software updating solutions, a few surveys have been made on the subject providing both general and individual analysis of existing solutions. They also introduce interesting concepts, issues and techniques such as quiescence, state-transfer, transformation functions and rollbacks.

10. **A Survey about Dynamic Software Updating, 2012 [20].** The paper includes some interesting solutions, such as 3.6 and 3.10, which essentially works by starting a new version process, transferring the state from the old version to new version and finally transferring control to the new version, very much like the state-transfer solutions mentioned earlier (6, 7 and 8).

11. **A survey of dynamic software updating, 2013 [27].** This survey has a category for solutions focusing on distributed systems, which matches our system. The following two solutions are particularly interesting:

    - DRACO [29] is a component framework, in many ways similar to PalCom, which supports updates in a state-transfer way. One key difference is that it is acceptable in PalCom to restart a device at any time in order to be updated due to the loosened requirement on uptime (II), whereas in DRACO the components must reach a special so called tranquility state before updating, in order to not break when performing the update.
    - Upstart [1] is an automatic update system for distributed systems. The updating process updates the distributed system's nodes by state-transfer. The idea contains an Upgrade Layer which acts as a communication proxy between an object (similar to PalCom devices) and other objects. The Upgrade Layer handles cross-version calls by using transformation functions. Unfortunately, the work only supports objects calling each others methods, and not general message passing, which is marked as future work. PalCom is built on message passing, which means that this work is not applicable.

Automatic updating

Many applications and operating systems offer some kind of automatic updating function. We are interested in the architectural design of such solutions, if it can be used directly for the PalCom system, or if it is possible to reuse ideas for a new solution.

Traditional operating systems generally include some kind of automatic updating solution. Examples include Microsoft Windows, Apple OS and many different Linux/Unix operating systems. They usually utilize the client-server model which is a usable model regarding our requirements. Unfortunately the solutions are bound to their operating system, thus breaking requirements (A) and (E).

For mobile devices (and lately also in traditional operating systems to some extent) we have seen automatic updating solutions that involves a centralized marketplace for all applications.

12. **Play Store / App Store.** Devices running the Android or Apple iOS operating system generally include a marketplace application, Play Store for Android and App Store for Apple iOS, where it is possible to browse and install applications to the device. These marketplace applications also handles updates. Unfortunately they sometimes require user intervention and are only available for their respective operating system, breaking our requirements (A), (E) and (G). Despite this fact, Play Store and App Store serve as possible models for automatic updates, where in the PalCom system we could have a PalCom update manager device act as a marketplace, and see the other PalCom devices which we want to update as the applications in the marketplace.

13. **Android for Work [12].** As an extension to the Play Store, Google has also released a service called Android for Work, which makes it possible to remotely install, remove and update device applications. These actions can also be performed automatically, and are controlled from an administrative "enterprise mobility management" solution, of which there are many to choose from. Unfortunately this solution falls short on some of the same requirement as Play Store, Pure Open Source PalCom code base (A) and Heterogenous PalCom implementations (E).

Both dynamic and automatic updating

Some updating solutions falls under both the dynamic and automatic software updating categories:

14. **OSGi [2] and Apache ACE [11].** The OSGi specification describes a system used to modularize Java programs, packaging bits of java code into components the same way PalCom packages functionality into services. OSGi has implementations such as Apache Felix and Equinox. It is possible to start, stop and update these components. There is no automatic updating solution for the OSGi system itself, although this can be solved by using the software distribution framework Apache ACE, which can update itself as well. Unfortunately the use of this solution for PalCom would introduce big dependencies on other code bases, thus breaking requirement (A), and it only works for Java implementations, thus breaking requirement (E).

15. **MobiCare, 2006 [6].** MobiCare is a service architecture focusing on health related services and supports dynamic updates. We can not use the solution as is, because it is too specific to health care applications. It is also not possible because of requirement (A), Pure Open Source PalCom code base. But there are some interesting architectural designs that we can reuse. The MobiCare design is client-server based where on the server side there are among others services for device activation, device configuration and remote dynamic device code upgrades. This kind of administrative setup is usable in the design of a new updating solution.

4.1.3 Conclusions based on requirements

When we study the existing solutions with the PalCom requirements (4.1.1) in mind it becomes clear that, to the best of our knowledge, there is no existing solution that can be directly applied to PalCom. This is largely due to requirements (A), Pure Open Source PalCom code base, and (E), Heterogenous PalCom implementations.

Instead of applying an existing solution, we will try to reuse concepts and ideas from these existing solutions in order to design a solution specific to PalCom, which can handle the requirements stated in section 4.1.1.

Because of the loosened requirement on uptime (II), it is possible to use a simpler stop-and-update process (similar to Ekiden (6), Kitsune (7) and Rubah (8)), instead of altering the device kernel during runtime. Also, due to the fact that all configuration and necessary states are stored in the PalCom filesystem, this state-transfer approach is natural with PalCom: stop the old version, transfer the relevant content from the PalCom filesystem, which is practically just a recursive copy, and start the new version.

Regarding the automatic updating solution, the client-server model is a simple and broadly used model which is also suitable for our problem. If the server runs PalCom as well, many requirements can be checked off for the server part by simply reusing existing features of the PalCom kernel. By using PalCom communication we support heterogenous PalCom implementations, requirement (E), from the server's point of view. The built-in detection of disappearing connections and connectivity will greatly help in dealing with requirement (B), unreliable connections, and (C), unreliable connectivity. Also, the use of PalCom tunnels and secure connections solves the security requirement (F). A PalCom tunnel is an SSH tunnel which is implemented into the PalCom kernel, which makes it available for all PalCom devices to use for communication.

4.2 Proposed solution

In this section, we propose our solution to the problem of automatic dynamic updating of the PalCom middleware.

4.2.1 Architecture

Units

A unit refers to a machine running one or more PalCom devices. In the architecture (figure 4.1) we distinguish between *client units* and *server units*. It is perfectly possible for a unit to be both a client unit and a server unit at the same time.

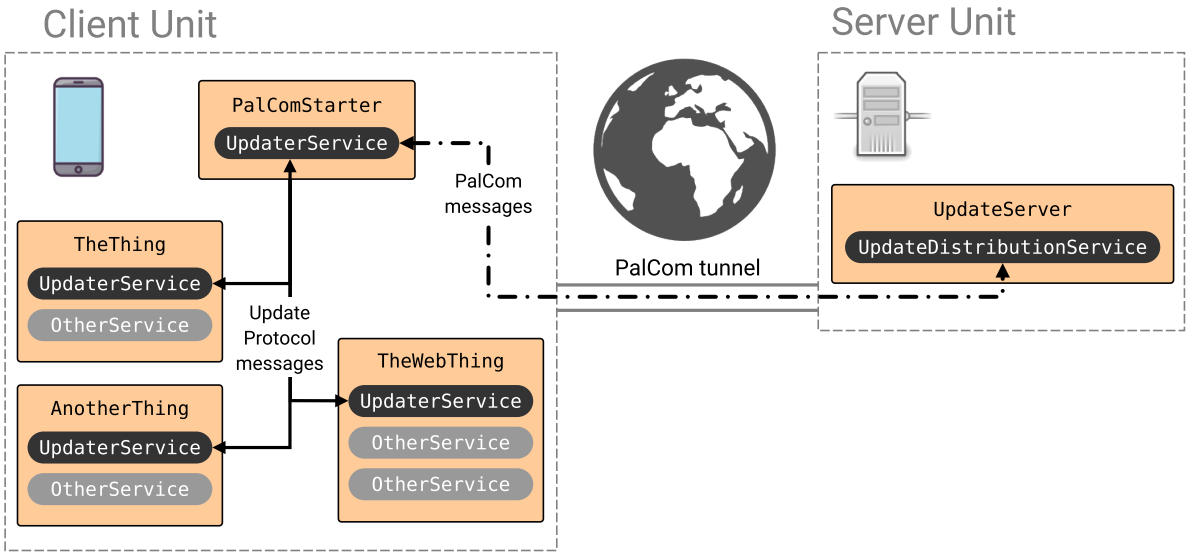

Figure 4.1: Architecture diagram of our proposed updating solution for PalCom. Dotted boxes represent units. Colored boxes inside represent PalCom devices containing PalCom services in rounded boxes. The services in black rounded boxes are the new parts in our proposed solution. Arrows with dashed lines represent PalCom communication and arrows with solid lines represent Update Protocol commands.

Client Unit

On the client unit we have a *PalComStarter*: a lightweight PalCom device which is running an instance of the PalCom service *UpdaterService*. PalComStarter has three areas of responsibility:

- **Starter**: to initially start up all other devices on the local client unit. PalComStarter itself is started by the client unit's operating system.
- **Monitor**: to act as a monitor for all devices running on the local client unit. If PalComStarter discovers any non-functioning device, that device is restarted by PalComStarter. For example if a device has a memory-leak or a fatal bug leading to a crash.
- **Updater**: to update all devices running on the local client unit.

PalComStarter's responsibilities is more thoroughly described later in its own section (PalComStarter). Besides PalComStarter, there is also one or more other PalCom devices (for example TheThing, TheWebThing, etc.) running on the same unit as PalComStarter. From now on, we refer to these devices as *monitored devices*, because they are being monitored (started, stopped and updated) by PalComStarter. Each of these monitored devices has an UpdaterService, in addition to other services or assemblies that may already be present on the device. Even though both PalComStarter and the monitored devices run the same UpdaterService, there is an important difference. The monitored devices' UpdaterService runs in a *monitored mode*, in contrast to PalComStarter's UpdaterService which runs in *monitoring mode*. In the monitoring mode, the functionality to perform starting, monitoring and updating of monitored devices is activated. In the monitored mode, the functionality to be controlled by a monitoring device and to be part of the updating process is activated.

PalComStarter and all monitored devices executes in separate runtime environments on the unit in order to isolate problems. For the Java implementation, this means that each device is running in a separate JVM. The significance of this is that when one device crashes it will not drag other monitored devices or, even worse, PalComStarter with it. If one of the devices crashes, PalComStarter and the other devices will safely continue executing as before in their own environment.

PalComStarter and all monitored devices are executing in parallel. They are not stopped unless explicitly told to do so, for example during update or restart.

Server Unit

In order to keep track of and distribute updates to PalCom clients we have an *UpdateServer*, also a PalCom device, which runs on the server unit. The UpdateServer has an *UpdateDistributionService* which communicates with PalComStarters running on the client units. The communication takes place using PalCom messages transmitted through PalCom tunnels. When developers push out new updates to the UpdateServer, the PalComStarters running on the client units are automatically notified and the updating process takes place automatically.

PalComStarter

A deeper description of the main part of the architecture: PalComStarter.

Starter. The starter role of PalComStarter means that it functions as a bootstrapper for the PalCom system on the client unit. The operating system on the client unit execute a file in the PalCom filesystem, the startup script, which points to the current version of PalComStarter. When PalComStarter is started it will check the monitoring configuration in a file called `monitoring.properties` in the PalCom filesystem (details in 4.2.6 Configuration and file structure) for information about which devices to start, monitor and update. If any of the devices specified to be monitored in the configuration are not installed on the system yet, PalComStarter retrieves executables and configurations for those devices from the UpdateServer. If all the devices it should monitor are already installed on the system, PalComStarter checks if they are currently running. If a device is not running, it will be started in a new instance.

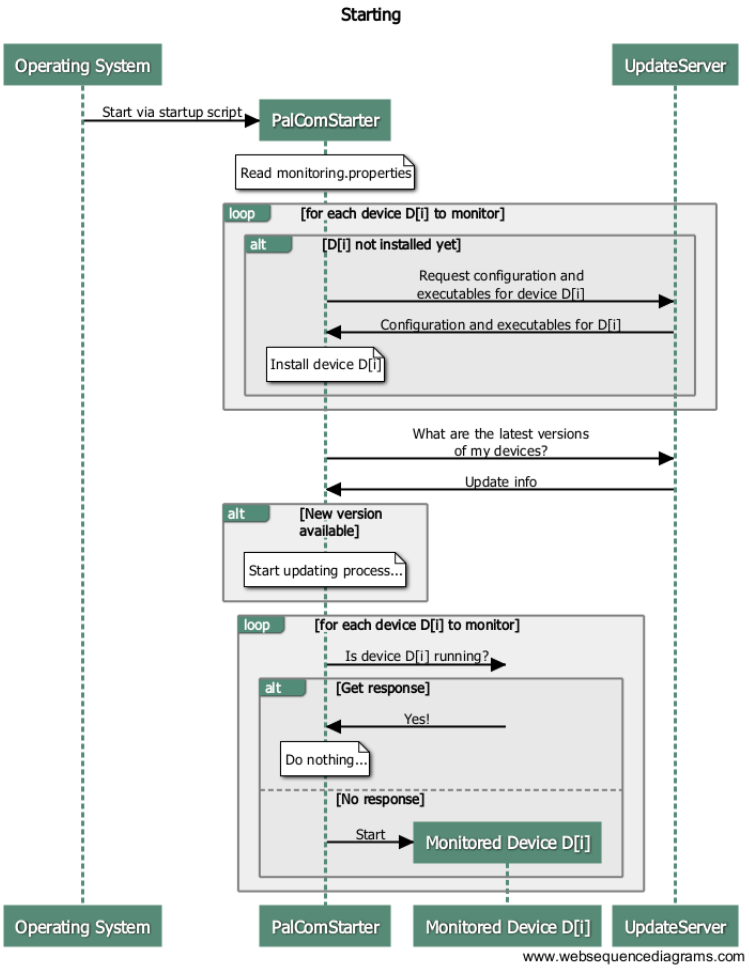

Figure 4.2: Sequence diagram describing the start process used by PalComStarter.

Monitor. PalComStarter is monitoring all devices running on the same unit. The monitoring is achieved through PalCom communication. The PalCom discovery mechanism is used in order to decide if a device is healthy or not. If a device is discovered it is assumed to be healthy. If a device is not discovered it is assumed to be unhealthy and thus restarted by PalComStarter.

Updater. In order to update devices, PalComStarter communicates with the UpdateServer, using a PalCom tunnel if necessary. When a new update is added to the UpdateServer, information about the new update is broadcasted to all PalComStarters. If any PalComStarter is not online when this broadcast is sent, that PalComStarter will still be able to compare its own version with the UpdateServer's broadcasted version when the connection to the UpdateServer is established again.

When UpdaterService has received an update message from the UpdateServer, the update is downloaded to PalComStarter. After the download is complete, the installation phase begins. During this phase, PalCom communication is not used anymore, but instead a separate update protocol is used, which we call *the Update Protocol*, that is specifically designed and intended only for updating. This makes it possible to handle updates between PalCom versions which break PalCom communication, for example if the current and new version use different PalCom communication protocols or due to bugs in one of the versions. It is still PalComStarter that is responsible for the updating process though; it is only switching to another protocol. The Update Protocol is much simpler than the PalCom communication protocols since it is only used locally on one unit, whereas PalCom communication protocols are designed for communication between devices that can be on different units. A reason for using a simple protocol is to reduce the number of possible bugs, and thus try to eliminate the need to update the Update Protocol itself. But if we need to update the Update protocol, we can use the method to update PalCom services as described by Månsson in her master's thesis [21], because the Update Protocol is fully contained in the UpdaterService. In the Update Protocol there are commands for starting, stopping, updating, aborting and to exchange information between devices during the updating process. TCP sockets are used to locally send these commands.

4.2.2 Updating process

The updating process is designed so that there will always be a PalCom device running and in charge of the updating process, in order to be able to abort and rollback to a previous functioning version in case of error. Abortion and rollback is described later in 4.2.4. The updating process can be divided into three stages where each stage denote that one of the devices is in charge of the update process. The different stages are illustrated in figures 4.3-4.8. They are also indicated in the more detailed sequence diagram available in appendix E (figure E.1). The device in charge of the updating process is responsible for deciding when to abort and to perform rollback in case of error during the updating process.

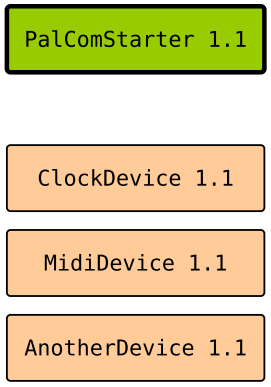

Figure 4.3: Example of update process in update stage one. The device with green background and thick borders is in charge of the updating process.

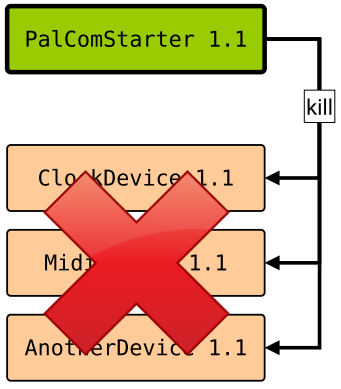

Figure 4.4: Example of update process in update stage one. The device with green background and thick borders is in charge of the updating process.

When the update process begins, in stage one, the current version of the PalComStarter is in charge (see figures 4.3, 4.4 and 4.5). The update data is fetched from the UpdateServer and saved to the PalCom fileSystem. The new versions of monitored devices are started and tested to make sure that they function properly.

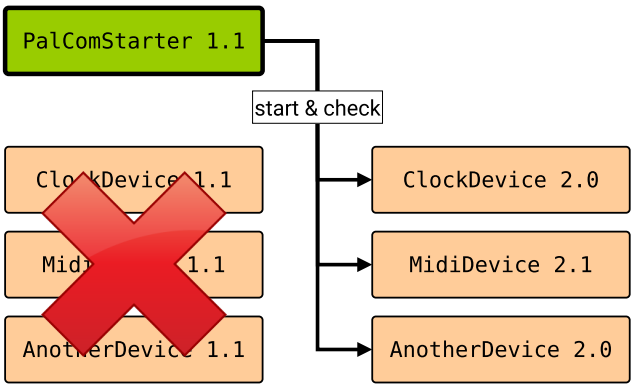

Figure 4.5: Example of update process in update stage one. The device with green background and thick borders is in charge of the updating process.

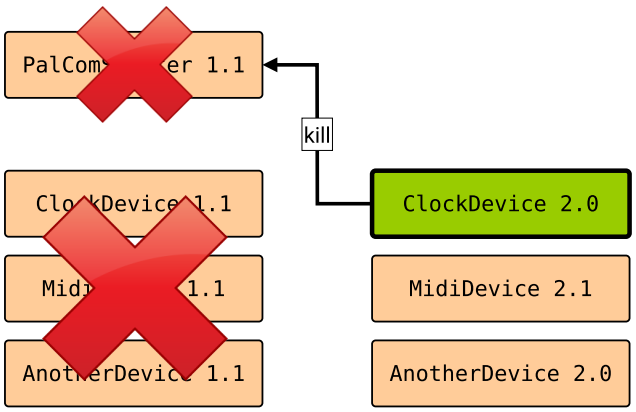

Figure 4.6: Example of update process in update stage two. The device with green background and thick borders is in charge of the updating process.

When PalComStarter has made sure that the new versions of its monitored devices are working, it is time for stage two, where PalComStarter itself must be updated. In this stage, one of the new version monitored devices is in charge of the updating process (see figures 4.6 and 4.7). To choose which monitored device that should be in charge is done by iterating through the unsorted list of all monitored devices. If the first chosen monitored device is not able to be in charge for any reason, another is chosen. If there are no monitored devices that are able to be in charge, the updating process will abort. When chosen, the monitored device in charge starts and tests the new PalComStarter to make sure it is functioning properly.

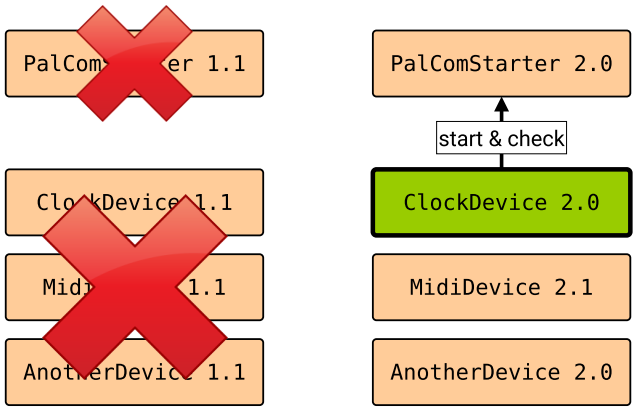

Figure 4.7: Example of update process in update stage two. The device with green background and thick borders is in charge of the updating process.

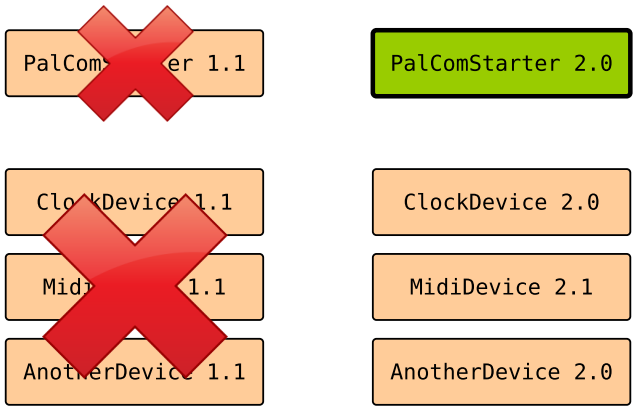

Figure 4.8: Example of update process in update stage three. The device with green background and thick borders is in charge of the updating process.

At last, when the monitored device in charge has made sure that the new version of PalComStarter is working, it is time for stage three. In this stage, the new version of PalComStarter is in charge of the updating process (see figure 4.8). PalComStarter performs cleanup which includes removing old executables and other unnecessary files, and updates the **startup script** which sets the new version as the current version and finishes the update process.

The purpose of the **startup script** is to be a shortcut which the operating system always can use to start the current version of the PalCom system. For example when the operating system starts for the first time, restarts after a power failure, or via some scheduled task or operating system service. It is also used in case of update abort in stage two and three, in order to fallback to the current version of the PalCom system. The startup script is essentially a text file containing the command to start PalComStarter. See section 4.2.6 Configuration and file structure, for more info about the location and content of the startup script.

4.2.3 Different updating scenarios

Depending on the type of update and which devices to update, we may not need to go through the whole updating process. We may only need to perform some of the updating stages. We describe the possible scenarios and what type of updating stages are needed in table 4.1. When only the monitored devices needs to be updated, it is unnecessary to perform update stage 2 and 3 which focuses on updating the PalComStarter. Similarly, when only PalComStarter needs to be updated, stage 1 is unnecessary.

Table 4.1: Updating scenarios and needed updating stages. The device types to update are listed in the first column and the update types are listed on the first row. The updating stages are acquired in the resulting matrix. For example, if a monitored device is to be updated with a non protocol breaking update, updating stage 1 is needed.

Device(s) to update	Non protocol breaking update	protocol breaking update
Monitored device(s)	stage 1	N/A
PalComStarter	stage 2,3	N/A
All devices	stage 1,2,3	stage 1,2,3

It is important to keep in mind that when there is a protocol-breaking update available for any one device type present on the client unit, it must also be available to all device types present on the client unit in order to perform the protocol-breaking update. If the protocol-breaking update is not available to all device types present on the client unit, the protocol-breaking update will not be performed.

4.2.4 Aborting update and performing rollback

If something goes wrong during the updating process, for example due to exceptions that can not be handled, the updating process will be aborted and the system will fall back to the current version. As stated before in the update process part (4.2.2), it is the responsibility of the device in charge to detect errors and perform abort.

When waiting for responses in the update process, or in any other scenario where there is a possibility for the process to be blocked indefinitely, we need to be able to abort if the wait time is too long. To solve this we set maximum waiting times for these scenarios, called abort timers. The abort timers will trigger the abortion of the updating process if these maximum waiting times are exceeded. More specifically, this is done by not using methods and threads that block indefinitely, but instead setting a maximum amount of time to wait. And in those cases where there have to be a thread or method that may block indefinitely, timer threads are used to interrupt the blocked thread if the maximum waiting time is exceeded.

4.2.5 Crash and failure handling

The updating process is designed with a lot of focus on requirement (H): unexpected crashes. Unexpected crashes are handled by leaving the current PalCom filesystem intact during the whole updating process. If something goes wrong during the update, the system will rollback to the current functioning state. PalComStarter will then try to update again at a later time.

If some device in the updating process should crash for any reason we have two possible scenarios. If the device that crashes is not in charge of the updating process, the device in charge will notice this and abort the updating process if necessary. If the device that crashes is in charge of the updating process, it is up to the operating system to restart the current PalComStarter via the startup script.

There are a lot of other errors that can occur during the updating process:

Connection, connectivity or communication error. If PalComStarter's connection to the UpdateServer is lost (due to failing tunnels or something else) or if the devices on the client unit is unable to communicate for any reason, the fallback timers will timeout which triggers update abort.

The server unit crashes. From the PalComStarter's point of view, this is the same as a connection error with the UpdateServer, and thus handled the same way. When the server unit restarts, the UpdateServer will be started again through its startup script.

The client unit crashes. This could be due to power failure, hardware failure, etc. When the client unit restarts, the unit will start the current PalComStarter again through the startup script.

If rolling back to current version do not work. If the updating process is aborted and the current version is not working, manual rescue is needed. This scenario can not be created by the updating process itself, because the current version remains untouched during the whole process. But there may of course be external events corrupting the file system.

Out of disk, and other exceptions. If the disk space runs out during the updating process, the process will abort and clean away all created files. The same goes for any other exception encountered during the updating process.

Out of memory, and other errors. If the updating process runs out of memory, or encounters any other major error, it will try to abort gracefully and return to the current version, although this is not always possible depending on the error. If it is not possible to abort gracefully, the device will be left in a corrupt state which is not allowed. The updating process will therefore shut down the affected device in that case, and it is up to the operating system to restart the current PalComStarter again via the startup script.

4.2.6 Configuration and file structure

In our architecture, we need some way to remember which devices to monitor and update, as well as their version and device type. We also need some place to put the executables of different device types and the device types' different versions. It is favorable to keep all settings and data in one place, so we will naturally use the already existing PalCom filesystem as storage location for this.

Recall the structure of the PalCom filesystem from the theory section about PalCom 3.2.3. If we start with the information about which devices to monitor, let us call it the monitoring properties, a good place to put it is in the *global* folder. That way, it is not bound to a specific device. The path to the monitoring properties will thus be:

```
[path to PalCom filesystem]/PalcomFilesystem/
global/monitoring.properties
```

If we continue with the executables, we see that it is a waste of space to have one executable for each device because if there are many devices of the same type, they could share the executable. It is a better idea to have one executable for each type of device. A good directory to place the executables is therefore the device type specific folders in the *global/devices* folder. In order to know which version of the executable we have, we also need to name them according to some versioning scheme. The path to a device's executable will thus be:

```
[path to PalCom filesystem]/PalcomFilesystem/global/devices/
[deviceType]/[deviceType]-[version].[executable file suffix]
```

We also need some place to put the startup script. The important thing to remember about the startup script is that its path is never allowed to be changed, because it is accessed by the operating system when the operating system wants to start the PalCom system. Therefore, a good place to put the startup script is in the *global* folder. The path to the startup script will thus be:

```
[path to PalCom filesystem]/PalcomFilesystem/global/startupscript
```

The resulting file structure is illustrated in figure 4.9.

Now that we have our file structure figured out, we may ask ourselves what kind of information we need to store in *monitoring.properties* for the architecture to work. PalComStarter must know the following:

UpdateServer's Device ID. PalComStarter must be able to establish a connection to the UpdateServer in order to receive updates and be able to ask for updates. We need to know the UpdateServer's device ID for this.

Monitored Devices' ID. PalComStarter needs to know which devices to monitor and to update. They can be identified by their device ID.

Monitored Devices' type and version. In order to start the monitored devices, PalComStarter must know what type of device it is and the version of the device type. This key pair is used in order to find the appropriate executable in the file system.

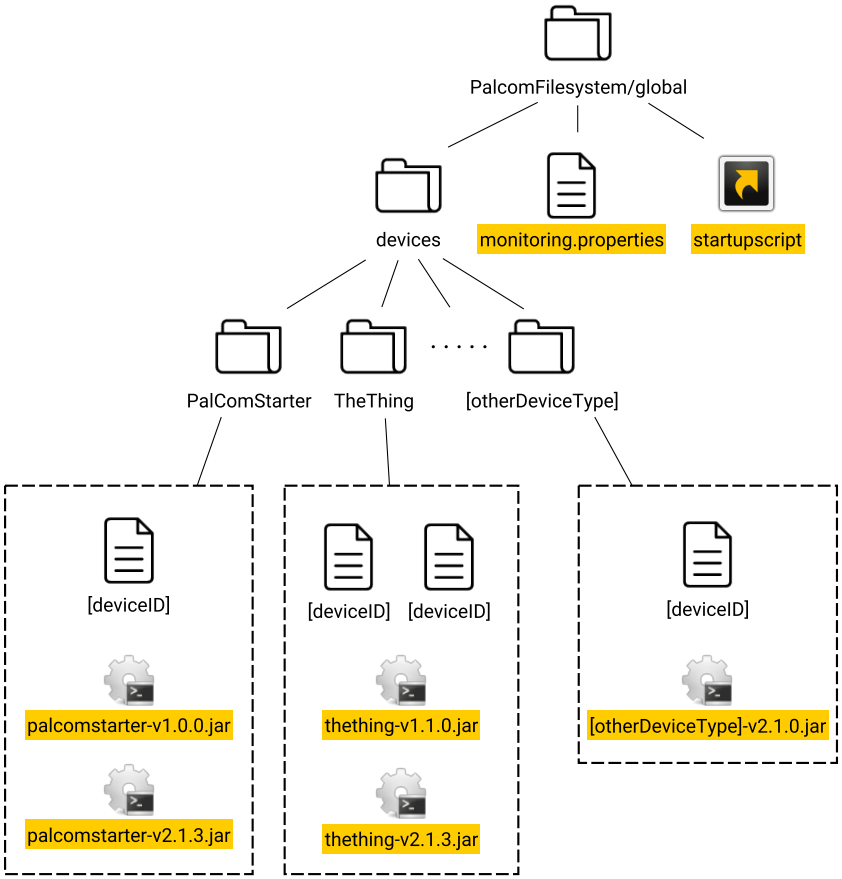

Figure 4.9: A graph showing the file structure of our proposed solution's architecture. The additions made in this architecture are highlighted.

For detailed instructions about how to set up a working system using the architecture, see appendix A.1.

4.2.7 Requirement fulfillment retrospective

We look back to the requirements stated in section 4.1.1 to see if they are fulfilled by our proposed solution:

(A) **Pure PalCom code base.** By writing a new solution tailored specifically for PalCom and reusing existing PalCom features, this requirement is naturally met.

(B) **Unreliable connections.** By using secure communication protocols that ensure that every message arrives, it does not matter if connections to other devices go down. If a connection is lost, the message is resent at a later time. When a connection has been lost for too long, the abort timers will trigger and the update process will abort and fallback to the latest working version. The update process will have to be started again at a later time.

(C) **Unreliable connectivity.** This is handled analogous to the way unreliable connections are handled.

(D) **Low bandwidth.** This can be handled by reducing the size of data being transferred as well as keeping the communication to a minimum. Almost all communication is made on the local device, between PalComStarter and its monitored devices, which means that the network bandwidth is not used at all. All communication between PalComStarter and the UpdateServer is sparse, and also lightweight except the sending of update data from the UpdateServer to the PalComStarter. This is solved by compressing the update data by using delta encoding, which means that only the difference between the current and the new update is sent, instead of sending the whole new update.

(E) **Heterogeneous PalCom implementations.** It is possible to use our update solution in all implementations that support TCP sockets, which are used to communicate with the Update Protocol.

(F) **Security.** PalCom tunnels and authentication are used during communication using PalCom messages. The update protocol is only communicated locally, thus need no protection.

(G) **No user interaction.** After uploading the update to the UpdateServer, the whole update process is performed automatically. In case of major errors, there is of course a need for human assistance.

(H) **Unexpected crashes.** By keeping the old filesystem intact until the updating process is all done and the new versions have been tested, it is always possible to fallback to the old version. Even in case of externally triggered crashes, such as power failure, the system will restart in a functioning state. It is assumed that the old version is the current version until the end of the process when the startup script is updated. The only point in the updating process where the system is vulnerable and could become corrupt when crashing is when the startup script is being edited to contain the command to start the new version of PalComStarter.

4.3 Future work

A great addition to our proposed solution would be the use of protocol proxy layers, as proposed by Ajmani et al. in [1], in the PalCom kernel if possible. This would make PalCom devices backward and forward compatible even with protocol breaking updates in a system of devices with mixed versions, which in turn would reduce the service interruption when protocol breaking updates are released because devices with different versions can then still communicate.

Another great addition to our proposed solution would be to use peer-to-peer (P2P) distribution techniques for the update content. By letting PalCom devices share updates to each other it is possible to lower the network congestion, apply less pressure on the UpdateServer and achieve faster distribution of updates. The impact of this feature is most noticeable in a system with very many devices and where there may be a faster connection between client units than between the server unit and client units.

Chapter 5

Evaluation

5.1 Experimental Setup

The implementation has been manually tested for functionality by running the client unit and the server unit on different Linux and Windows units on the same WiFi.

When testing the performance of our solution we benchmark two things:

Test 1: Average time to update. From the point in time that an update is made available on the UpdateServer, how long time does it take for a client unit to update. This measures how fast the overall updating process is. *Total time* is the total time for the updating process. We will also measure how much of this total time that is spent on sending the update content, the *time to send update content*. *Time to update* is the total time subtracted with the *time to send update content*

Test 2: Average monitored device downtime. In this test we record how much downtime each monitored device suffer during the updating process. This measures how fast the updating process is from a monitored device's point of view. It can also be seen as the service interruption noticed by users and other devices that are depending on the device being updated.

We benchmark these things in three different scenarios:

Scenario A: Update PalComStarter. In this scenario, we only update PalComStarter, which means that only update stage 2 and 3 are performed. In this scenario, the monitored devices do not suffer any downtime and it is thus not interesting to perform test 2 in this scenario.

Scenario B: Update monitored devices. In this scenario, we only update the monitored devices, which means that only update stage 1 is performed.

Scenario C: Update PalComStarter and monitored devices. In this scenario, we update all devices present on the client unit, performing all updating stages 1, 2 and 3. This shows us how long time it takes to perform a protocol breaking update, which means that we update all devices at the same time.

Each scenario is run 20 times in order to get a mean value for test 1 and test 2.

The update content is .jar-files with the size of 2 MB for PalComStarter and 3.4 MB for the monitored devices.

The test system specifications are shown in table 5.1.

Table 5.1: Test system specifications

System:	Laptop
CPU:	2.6Ghz (3.6Ghz turbo) quad-core Intel i7-4720HQ
Memory:	8Gb DDR3
Storage space:	256Gb SSD
Connection:	WiFi 802.11 b/g/n/ac
Operating system:	openSUSE Tumbleweed x86-64

The amount of time it takes for a client unit to update depends on how many devices there are on that unit. In our benchmark we will use a client unit with two PalCom devices in addition to the PalComStarter itself. This represents a typical client unit according to usage scenarios in projects using PalCom in collaboration with the Department of Computer Science at the Faculty of Engineering at Lund University.

5.2 Results

In this section we present the results from the benchmarks specified in 5.1 Experimental Setup. Results are shown in table 5.2.

Table 5.2: Scenario A: Update PalComStarter. Mean values with standard deviation in paranthesis.

	Test 1						Test 2	
	Time to update		Time to send update content		Total time		Monitored device downtime	
Scenario A	0.64 s	(0.05 s)	4.9 s	(1.5 s)	5.5 s	(1.5 s)	0 s	
Scenario B	1.5 s	(0.059 s)	3.4 s	(0.92 s)	4.9 s	(0.93 s)	0.57 s	(0.039 s)
Scenario C	2.5 s	(0.4 s)	7.6 s	(2 s)	10 s	(2 s)	0.59 s	(0.051 s)

5.3 Discussion

In this section we discuss different aspects of our solution that is worth mentioning and interesting things that have come up working with this thesis.

5.3.1 Problem statement

To start with, we must look back to see if we can answer the questions asked in the problem statement. We begin to answer the first question:

- What are the important requirements when performing automatic dynamic updating of the PalCom middleware?

Through specifying and analyzing our system we got a clear picture of important requirements specific to our system, which can be seen in section 4.1.1. Followed by literature studies of existing systems and and related work in section 4.1.2, we learned about more general requirements and techniques related to automatic and dynamic updating. By combining these two sources we can move on to the second question:

- How can we perform automatic dynamic updating of the PalCom middleware?

By introducing our proposed solution in section 4.2, we present a way to solve the problem of automatic dynamic updating of the PalCom middleware, that also fulfills the important requirements found in the previous question. As a proof-of-concept, we also built a working implementation of our proposed solution.

5.3.2 Usefulness of our solution

It is also important to reflect about the usefulness of our solution. As stated in the introductory chapter 1, there are a lot of connected devices in the world and they are predicted to increase rapidly. By using our solution together with PalCom it is possible to solve the problem of updating the big amount of devices present in Internet of Things systems. As a direct practical example, it is worth mentioning the itACiH project [17], which is focusing on developing IT support for advanced health care at home. By using PalCom devices in the home of patients and available to health care personnel, it is possible to remotely collect patient values and establish communication directly between patients and health care personnel. The automatic updating of these devices is a great help. Without it, the updating process have to be done manually for each device. Our solution is also applicable to the health care project mentioned in the motivating example in chapter 2, where ambulance personnel use Android smartpads to proactively collect and send information to hospitals. By using our updating solution it is possible to automatically ship updates to these smartpads, without disturbing the work of the ambulance personnel.

5.3.3 Our solution and possible scenarios

In the section 4.2 Proposed solution, we explained how our solution works. But it is also interesting to see how it solves different scenarios. The obvious scenario our solution solves is to update the PalCom devices running on a client unit, but there may also be other scenarios that may or may not be as obvious:

What if we want to update PalComStarter itself?
As described in 4.2.2 Updating process, the updating includes PalComStarter by putting the monitored devices in charge of updating the PalComStarter.

What if we want to update UpdaterService, UpdateDistributionService, or some other service running on the devices?
As stated in the problem statement in chapter 1, the updating of services and assemblies is a different problem which have been solved in another thesis [21].

What if we want to update the UpdateServer?
By running a PalComStarter on the server unit with the UpdateServer, it is possible to automatically update the UpdateServer as well.

What if we want to update the Update Protocol?
The Update Protocol is specified and only used in UpdaterService, which means that to update the Update Protocol it is only needed to update the UpdaterService service (see previous question on how to do this). Because of the fact that services and assemblies are separated from devices, it is possible to update UpdaterService without updating the device it runs on. This means that when updating the Update Protocol, the Update Protocol itself is not used, which makes it possible to release protocol breaking updates to the Update Protocol.

What happens if there is a protocol breaking update for PalCom?
Our updating solution solves protocol breaking updates by introducing the Update Protocol. PalComStarter and all monitored devices can communicate through the update process, even when the current version's communication protocol is incompatible with the new version's, by using the Update Protocol. The UpdateServer though, have to communicate with PalComStarter through PalCom messages in order to make use of PalCom tunnels and to not make the Update Protocol too complex. To handle protocol breaking updates, two instances of UpdateServer must be online until all devices have been updated. It is possible to solve this in another way, for example similar to [1] by introducing a protocol proxy layer in the UpdateServer for PalCom communication, which we see as a great future improvement. But even though it is a great feature, it is not the intention of our updating solution to support a PalCom system with mixed PalCom protocols for an extended period of time. The intention is that when a new update is released, no matter if it is protocol breaking or not, the client units should update as soon as possible. As stated in the introduction (1), it is not possible for a whole distributed system to update itself at once because some devices may be offline, sleeping, busy with other things, slower than others to update, etc. But when a device is ready, the updating should be performed as soon as possible.

5.3.4 Benchmark results

We believe that the results from our benchmarks in section 5.2 are satisfactory, because of the low device downtime of less than a second. We believe that the user will not experience any notable interruption during the updating process.

It is also interesting to point out that the most time during the updating process is spent transferring the update content from the UpdateServer to the PalComStarter. This is reasonable because of the fact that we did not have time to implement update content delta compression. The update content sent in the tests was complete .jar-files with the size of 2 MB for PalComStarter and 3.4 MB for the monitored devices. If only the diff where to be sent, the size could be greatly reduced to an order of kilobytes instead of megabytes, which in turn would reduce the time to update.

If we set aside the time needed to transfer the update content, we see that the actual updating process is relatively fast (table 5.3).

Table 5.3: Time to update subtracted by time to send update content. Mean values rounded to the nearest millisecond.

Scenario A	Scenario B	Scenario C
0.64 s	1.5 s	2.5 s

PalCom can run on many different systems. From high performing desktop, laptop or server computers to minimal embedded systems with very low processing power and storage space, so it is important to test our updating solution for both kinds of systems. We tried to test our solution on a minimal Android device called Minimal Viable Device, which features lower performance hardware than our test system, but we were unable to make it work due to hardware problems. Given more time, we would like to test our solution on more different kinds of hardware. We would also like to perform larger experiments with our solution and test it in a live situation, such as the in the itACiH project.

5.3.5 Implementation improvements and features

There is a lot that can be improved in the implementation of our solution. For example the way of communicating the Update Protocol can be implemented in a nicer way. For the UpdateServer it is possible to add a lot of commands helping the administrative work of uploading and keeping track of updates, and maybe even adding a web interface. The implemented solution as of now is working, but is not finished by any means, and can always be improved and updated with more features.

Abort timer length

In our solution we use abort timers in order to abort when the wait time is too long waiting for a response or when blocking to send a message. The hard part here is to know what

length to set on these timers. For devices with low bandwidth or unstable connections, the timer length may need to be longer than for devices with stable high bandwidth connections. Also, the timers may need different lengths depending on what action they are tied to. When waiting for update data there is much more data to transfer than when just pinging the UpdateServer, and thus need a longer timer. The timer length may also depend on the processing power of the device. An example of this is when device A starts device B and A waits for an answer from B to know that it have started gracefully. For a low performance device, the time needed for device A to start device B may be longer and thus require more waiting time and a longer timer.

When implementing our solution we have empirically tested different abort timer lengths in order to find a balance between having a long enough timer so that the updating process has enough time to perform its task, and not waiting to long to abort, so that we do not interrupt service for to long. The timer lengths have been tuned to our testing systems running in our testing environment only, which may or may not work for other systems and environments.

What is not implemented yet?

Unfortunately, we did not have time to implement all features of our proposed solution into our proof-of-concept implementation. We see this as future work.

- Use delta encoding to compress update data.
- Make it possible for the UpdateServer to store and distribute update content depending on language of implementation. Currently, there is only support for the Java implementation of PalCom, using .jar-files.
- Distinguish between protocol breaking updates and other updates.
- When PalComStarter starts and reads monitoring.properties, it should check if the devices to monitor are installed or not. If some devices are not installed on the system yet (they have no DeviceID in monitoring.properties), executables and configurations should be downloaded from UpdateServer and new DeviceID:s should be generated for them. This feature presumably resides in a new PalCom service separated from UpdaterService.
- Updating of UpdateServer with the help of PalComStarter. When protocol breaking updates are released, a new UpdateServer have to be started and run in parallel with the old version until all devices are updated.

Chapter 6

Conclusions

In this master's thesis we propose a solution which dynamically updates the middleware of an Internet of Things system. More specifically a distributed system of PalCom devices, a PalCom system. The middleware of the devices are updated in a completely automatic manner by using a monitoring device, called PalComStarter, which is responsible for startup, monitoring and updating of all other PalCom devices on the local machine. The PalComStarter receives updates from an UpdateServer. All devices, protocols and parts are updatable, including the PalComStarter and the UpdateServer, even in protocol breaking updates.

By studying related work, existing solutions and our specified system, we arrived at a number of requirements that needed to be fulfilled. (1) The solution must be fully automatic, requiring no user interaction. (2) It must be able to handle unreliable connections and (3) unreliable connectivity. (4) Unexpected crashed must be handled so that the system always can return to a non-corrupt functional state. (5) Protocol breaking updates must be supported. (6) It must have security in mind, as well as (7) the possibility of low bandwidth connections. (8) The code base must be open source and not introduce significant dependencies on other systems or code bases. (9) The solution must work in heterogeneous PalCom implementations. We designed an updating architecture, our proposed solution, which fulfills all of the above mentioned requirements.

We built a proof-of-concept implementation of our proposed solution in Java. The implementation was benchmarked to evaluate update time and device downtime during the updating process, on a machine with three devices. The resulting update time was 10 seconds in the worst scenario, when updating all devices with a protocol breaking update, with an average device downtime of 0.6 seconds.

With our work on updating PalCom devices together with the work provided by [21] to update services and assemblies, we have achieved a fully automatic and dynamic deployment and updating of PalCom systems.

Bibliography

[1] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 452–476. Springer, 2006.

[2] OSGi Alliance. Osgi architecture. `https://www.osgi.org/developer/architecture/` accessed 2016-02-08.

[3] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs, editors, *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 187–198. ACM, 2009.

[4] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[5] Robert Pawel Bialek, Eric Jul, Jean-Guy Schneider, and Yan Jin. Partitioning of java applications to support dynamic updates. In *11th Asia-Pacific Software Engineering Conference (APSEC 2004), 30 November - 3 December 2004, Busan, Korea*, pages 616–623. IEEE Computer Society, 2004.

[6] Rajiv Chakravorty. A programmable service architecture for mobile medical care. In *4th IEEE Conference on Pervasive Computing and Communications Workshops (PerCom 2006 Workshops), 13-17 March 2006, Pisa, Italy*, pages 532–536. IEEE Computer Society, 2006.

[7] Ericsson. Ericsson mobility report, 2015. `http://www.ericsson.com/res/docs/2015/mobility-report/ericsson-mobility-report-nov-2015.pdf` accessed 2016-02-04.

[8] EU-IST. Palpable computing. `http://www.ist-palcom.org/`, accessed 2015-11-27.

[9] Dave Evans. The internet of things, how the next evolution of the internet is changing everything, 2011. https://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf accessed 2016-02-04.

[10] David Svensson Fors. *Assemblies of Pervasive Services*. PhD thesis, Dept. of Computer Science, Lund University, 2009. http://lup.lub.lu.se/record/1287708.

[11] The Apache Software Foundation. Apache ace. https://ace.apache.org/ accessed 2016-02-08.

[12] Google. Android for work. https://static.googleusercontent.com/media/www.google.com/sv/SE/work/android/files/android-for-work-apps-guide.pdf accessed 2015-11-13 12:49.

[13] Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lü. Low-disruptive dynamic updating of java applications. *Information & Software Technology*, 56(9):1086–1098, 2014.

[14] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael W. Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. *ACM Trans. Program. Lang. Syst.*, 36(4):13:1–13:38, 2014.

[15] Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State transfer for clear and efficient runtime updates. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 179–184. IEEE, 2011.

[16] Michael W. Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.

[17] itACiH. It support for advanced care of cancer patients at home. http://itacih.cs.lth.se/ accessed 2016-02-24.

[18] Boris Magnusson and Björn A. Johnsson. Some like it hot: automating an electric kettle using palcom. In Friedemann Mattern, Silvia Santini, John F. Canny, Marc Langheinrich, and Jun Rekimoto, editors, *The 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '13, Zurich, Switzerland, September 8-12, 2013 - Adjunct Publication*, pages 63–66. ACM, 2013.

[19] Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In Geoffrey M. Voelker and Alec Wolman, editors, *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009*. USENIX Association, 2009.

[20] Emili Miedes and Francesc D Munoz-Escoı. A survey about dynamic software updating. *Instituto Universitario Mixto Tecnologico de Infromatica, Universitat Politecnica de Valencia, Campus de Vera s/n*, 46022, 2012.

[21] Mia Månsson. Dynamic installation and automatic update of bluetooth low energy devices in palcom. Master's thesis, Dept. of Computer Science, Lund University, 2015. http://lup.lub.lu.se/student-papers/record/5471214.

[22] Daniel Nilsson and Mattias Nordahl. *Minimal implementation av PalCom för små enheter*. Department of Computer Science, Faculty of Engineering, LTH, Lund University, Lund, 2013.

[23] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of java software. In *18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*, pages 649–658. IEEE Computer Society, 2002.

[24] Luís Pina, Luís Veiga, and Michael W. Hicks. Rubah: DSU for java on a stock JVM. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 103–119. ACM, 2014.

[25] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Javadaptor - flexible runtime updates of java applications. *Softw., Pract. Exper.*, 43(2):153–185, 2013.

[26] Michael Schiefer. Smart home definition and security threats. In Jana Dittmann and Holger Morgenstern, editors, *Ninth International Conference on IT Security Incident Management & IT Forensics, IMF 2015, Magdeburg, Germany, May 18-20, 2015*, pages 114–118. IEEE, 2015.

[27] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, 2013.

[28] Rob van der Meulen. Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015, 2015. http://www.gartner.com/newsroom/id/3165317 accessed 2016-02-04.

[29] Yves Vandewaude. *Dynamically updating component-oriented systems.* PhD thesis, University of Leuven, 2007.

[30] International Telecommunication Union. Recommendation ITU-T Y.2060. Overview of the internet of things, 2012. http://handle.itu.int/11.1002/1000/11559.

Appendices

Appendix A

Manual

A.1 Installation guide

In order for the system to work for our proposed architecture, some configuration preparations are needed.

A.1.1 PalComStarter

monitoring.properties

For PalComStarter to work, the following properties must be set in `monitoring.properties`:

```
general@updateServerDeviceID     =  [device ID]
deviceTypeVersion@PalComStarter  =  [version]
```

Where [version] is on the form "A.B.C" where A,B,C are natural numbers. A: major (protocol breaking) version. B: minor version. C: patch number.

For every device instance to monitor, the following properties must be set:

```
monitoredDeviceNames@[device instance name]  =  ["enabled" if this device
                                                should be monitored]
monitoredDevice-[device instance name]@ID    =  [device ID, or left blank
                                                if the device is not in-
                                                stalled yet]
monitoredDevice-[device instance name]@type  =  [device type]
```

For every type of device to monitor, the following property must be set:

```
deviceTypeVersion@[device type] = [version]
```

PalcomStarter uses jar-files to start its monitored devices. Every monitored device, as well as PalcomStarter itself, must keep its jar-file in its device type specific folder:

```
[path to PalcomFilesystem]/PalcomFilesystem/global/
devices/[device type]/[device type]-[version].jar
```

The startup script should be located at:

```
[path to PalcomFilesystem]/PalcomFilesystem/global/startupscript
```

The startup script must contain the command to start the current PalComStarter:

```
java -jar [path to PalComStarter jar] -x [PalComStarter device ID]
-f [path to PalcomFilesystem]
```

The `-x` option may be omitted if there is only one PalComStarter configuration and the `-f` option may be omitted if the standard location of PalcomFilesystem is used.

Example configuration

Say that we have the following devices ...

Device instance name	Device type	Device ID
PS	PalcomStarter	C:e8f17688-3203-456d-ac84-deb10c56f49e
TT1	TheThing	C:518126a6-a623-491e-b661-9930e1766141
TT2	TheThing	C:cff30b40-ec25-453e-927a-2042e6c40663
WT	TheWebThing	C:3f265a08-a993-418e-8b2c-d19e0a7e5cd0
AT	TheAndroidThing	C:84a9c2bf-d062-46a8-b479-e5020f68d43a
US	UpdateServer	C:4840b04b-dd38-461a-b40a-ef939f6248ae

... where the device type versions are ...

Device type	version
PalComStarter	1.1.0
TheThing	1.3.2
TheWebThing	1.3.2
TheAndroidThing	1.6.14

... and the path to our PalcomFilesystem directory is ...

```
/home/palcomuser/PalcomFilesystem
```

Our properties in `monitoring.properties` would then be (notice that ":" is escaped with "\" in the configuration file) ...

```
general@updateServerDeviceID=C\:4840b04b-dd38-461a-b40a-ef939f6248ae
deviceTypeVersion@PalComStarter=1.1.0
deviceTypeVersion@TheThing=1.3.2
deviceTypeVersion@TheWebThing=1.3.2
```

```
deviceTypeVersion@TheAndroidThing=1.6.14
monitoredDeviceNames@TT1=enabled
monitoredDeviceNames@TT2=enabled
monitoredDeviceNames@WT=enabled
monitoredDeviceNames@AT=enabled
monitoredDevice-TT1@ID=C\:518126a6-a623-491e-b661-9930e1766141
monitoredDevice-TT1@type=TheThing
monitoredDevice-TT2@ID=C\:cff30b40-ec25-453e-927a-2042e6c40663
monitoredDevice-TT2@type=TheThing
monitoredDevice-WT@ID=C\:3f265a08-a993-418e-8b2c-d19e0a7e5cd0
monitoredDevice-WT@type=TheWebThing
monitoredDevice-AT@ID=C\:84a9c2bf-d062-46a8-b479-e5020f68d43a
monitoredDevice-AT@type=TheAndroidThing
```

... , we would have placed the jar-files in the following locations ...

```
/home/palcomuser/PalcomFilesystem/global/devices/
PalComStarter/PalComStarter-1.1.0.jar
/home/palcomuser/PalcomFilesystem/global/devices/
TheThing/TheThing-1.3.2.jar
/home/palcomuser/PalcomFilesystem/global/devices/
TheWebThing/TheWebThing-1.3.2.jar
/home/palcomuser/PalcomFilesystem/global/devices/
TheAndroidThing-1.6.14.jar
```

... and the content of `startupscript` would be ...

```
java -jar /home/palcomuser/PalcomFilesystem/global/
devices/PalComStarter/PalComStarter-1.1.0.jar
-x C:e8f17688-3203-456d-ac84-deb10c56f49e -f /home/palcomuser
```

To start the system, run `startupscript`.

A.1.2 Monitored devices

For every device specified in PalcomStarter's configuration, `monitoring.properties`, add the service UpdaterService to the device.

A.1.3 UpdateServer

Simply add the service UpdateDistributionService to the device which should act as an UpdateServer. Use the appropriate commands to upload and distribute updates.

A.2 Usage examples

A.2.1 How to add and broadcast an update for a single device type

1. Start a PalCom Browser connected to the same PalCom network as UpdateServer. A PalCom tunnel may need to be configured.
2. Navigate to the `UpdateDistributionService` in the PalCom Browser.
3. Click on the `broadcast update for single device type` command.
4. Enter the following information:
   (a) `device type`, (eg "TheThing").
   (b) `version`, the device type version on the form [major].[minor].[patch] (eg "1.2.5")
   (c) `jar content`, the executable .jar-file, (eg "thething.jar").
5. Click on `Invoke` to broadcast the update to all connected PalComStarters.

A.2.2 How to add and broadcast updates for multiple device types

1. Start a PalCom Browser connected to the same PalCom network as UpdateServer. A PalCom tunnel may need to be configured.
2. Navigate to the `UpdateDistributionService` in the PalCom Browser.
3. Click on the `add jar` command.
4. Do the following for every device type to update:
   (a) Enter the following information:
      i. `device type`, (eg "TheThing").
      ii. `version`, the device type version on the form [major].[minor].[patch] (eg "1.2.5")
      iii. `jar content`, the executable .jar-file, (eg "thething.jar").
   (b) Click on `Invoke`.
5. Click on `broadcast update for multiple device types` to broadcast info about the latest update for each device type to all connected PalComStarters.

A.2.3 How to update the Update Protocol

The Update Protocol is fully contained in the service UpdaterService. Updating of services is not handled in this work. Use the standard procedure of updating services to update UpdaterService. (See [21])

Appendix B

Update Protocol

All commands available in **se.lth.cs.palcom.updaterservice.UpdaterService**.

KILL = "kill" commands receiving device to halt execution.

ABORT = "abort!" commands receiver to abort update.

CHECK_SOCKET = "socket working?" commands receiver to reply with confirmation (see command below) in order to test socket communication.

CHECK_SOCKET_CONFIRM = "socket working!"

CHECK_UPDATE_SERVER = "update server hear you?" commands receiver to ping UpdateServer in order to test PalCom communication and eventual PalCom tunnels, followed by a reply with confirmation (see command below).

CHECK_UPDATE_SERVER_CONFIRM = "update server hear me!"

FINISH_DEVICE_STARTUP_CHECK = "finish device startup check" used by PalComStarter to notify a newly started monitored device that the startup check is finished. Commands receiver to reply with confirmation (see command below).

FINISH_DEVICE_STARTUP_CHECK_ACK = "finish device startup check ACK"

STAGE_TWO = "update stage two" , commands the initiation of update stage two, when the monitored device is in charge of updating PalComStarter to the new version.

STAGE_TWO_SENDING_DEVICE_INFO = "stage two device info" , followed by information needed by the monitored device in stage two to update PalComStarter.

FINISH_STAGE_TWO = "finish stage two" commands the finish of update stage two, meaning that the new version of PalComStarter is in charge of finishing the update process.

Appendix C

PalCom Service Specifications

C.1 UpdaterService

All commands available in **se.lth.cs.updaterservice.UpdaterService**.

C.1.1 General commands

Commands in

KILL = "kill" ,
Commands receiver to halt.

CHECK_UPDATE_SERVER_CONFIRM = "I hear you!" ,
Received from UpdateDistributionService as confirmation when testing connection.

ABORT_UPDATE = "abort update!" ,
Commands receiver to abort update.

Commands out

CHECK_UPDATE_SERVER = "do you hear me?" ,
Request sent to UpdateDistributionService in order to test connection.

C.1.2 Monitor specific commands

Commands in

UPDATE = "update" ,
Parameters: VERSION (text/plain)
Received from UpdateDistributionService when a new update is available. Starts the update process.

STOP_MONITORED _DEVICES = "stopAllMonitoredDevices" ,
Stops all monitored devices.

UPDATE_DATA = "updateData" ,
Parameters: VERSION (text/plain), UPDATE\CONTENT (application/x-jar)
Received from UpdateDistributionService in response to update content request.

DISABLE_MONITORING = "disable monitor" ,
Disables monitoring of monitored devices.

ENABLE_MONITORING = "enable monitor" ,
Enables monitoring of monitored devices.

LIST_MONITORED_DEVICES = "list all monitored devices" ,
Lists the index, device ID and device type of all monitored devices. The response is a concatenated string where each device has the form: "index=[internal index of monitored device] ID=[device ID] type=[type of device]".

KILL_DEVICE_BY_INDEX = "kill device by index" ,
Parameter: MONITORED\DEVICE\INDEX (text/plain)
Kills a monitored device specified by index (which is obtain through the "list all monitored devices" command)

START_DEVICE_BY_INDEX = "start device by index" ,
Parameter: MONITORED\DEVICE\INDEX (text/plain)
Starts a monitored device specified by index (which is obtain through the "list all monitored devices" command)

RESTART_DEVICE_BY_INDEX = "restart device by index" ,
Parameter: MONITORED\DEVICE\INDEX (text/plain)
Restarts (kills and starts) a monitored device specified by index (which is obtain through the "list all monitored devices" command)

RESET_UPDATE_ABORTED_COUNTER = "reset update aborted counter" ,
Resets the counter, which counts the number of times an update has aborted, to zero. This makes it possible to retry to update again, even though the maximum amount of tries has been used.

Commands out

UPDATE_CONTENT_REQUEST = "gief the jar!" ,
Parameters: VERSION (text/plain)
Update content request sent to UpdateServer.

KILL = "kill" ,
Command used to tell monitored devices to halt.

CHECK_LATEST_VERSION = "latest version?" ,
Requests latest version info from UpdateDistributionService.

LIST_MONITORED_DEVICES = "list of all monitored devices" ,
Reply with all monitored devices.

C.2 UpdateDistributionService

All commands available in **se.lth.cs.updaterservice.UpdateDistributionService**.

Commands in

BROADCAST_UPDATE = "broadcast update" ,
Parameters: VERSION (text/plain), UPDATE_CONTENT (application/x-jar)
Used by administrators to broadcast an update.

UPDATE_CONTENT_REQUEST = "gief the jar!" ,
Parameters: VERSION (text/plain)
Received from PalComStarters requesting a specific update version.

CHECK_UPDATE_SERVER = "do you hear me?" ,
Received from PalComStarters and monitored devices testing their connection.

CHECK_LATEST_VERSION = "latest version?" ,
Latest version request from client.

Commands out

UPDATE = "update" ,
Parameters: VERSION (text/plain)
Broadcast sent to all PalComStarters, notifying them about the new version available.

UPDATE_DATA = "updateData" ,
Parameters: VERSION (text/plain), UPDATE_CONTENT (application/x-jar),
Reply to the update content request command

CHECK_UPDATE_SERVER_CONFIRM = "I hear you!" ,
Confirmation reply sent to device testing their connection.

Appendix D

Source code

The source code is available at: `https://github.com/splushii/PalComStarter`

D.1 se.lth.cs.palcom.palcomstarter.*

D.1.1 PalComStarter.java

Extends AbstractDevice to create a minimal PalCom device used as a monitoring device. All updating functionality is in UpdaterService.

D.2 se.lth.cs.palcom.updaterservice.*

D.2.1 UpdaterService.java

The UpdaterService.java specifies the service used both by PalComStarter (a monitoring device) as well as by monitored devices. During the service's startup, it is decided if the service should act as a PalComStarter or as a monitored device. If the device running the service is of type PalComStarter, it will act as a PalComStarter and otherwise as a monitored device. The available commands differ depending on if the service runs as a PalComStarter or as a monitored device. If the service runs as a PalComStarter, the PalComStarterStartThread is created and started. If the service runs as a monitored device, the MonitoredDeviceStartThread is created and started.

Both PalComStarter and monitored devices utilizes the classes SocketSender and SocketListenerThread in order to send and receive Update Protocol commands.

PalComStarter (monitoring device)

The following threads are run only as PalComStarter: PalComStarterStartThread, UpdateStageOneThread, UpdateStageThreeThread, MonitoringThread.

When PalComStarter is started, PalComStarterStartThread is started.

When PalComStarterStartThread has decided that we are fully operational, MonitoringThread is started.

When an update command is received, UpdateStageOneThread is started.

When started in the middle of an update, which is decided by PalComStarter, UpdateStageThreeThread is started.

Monitored device

The following threads are run only as a monitored device: MonitoredDeviceStartThread, UpdateStageTwoThread.

When started as a monitored device, MonitoredDeviceStartThread is started.

When MonitoredDeviceStartThread receives the command to initiate update stage two, UpdateStageTwoThread is started.

D.2.2 MonitoredDevice.java

A structure used by MonitoringThread and during update to represent a monitored device and relevant attributes.

D.2.3 SocketSender.java

A helper-class used to send Update Protocol commands via TCP sockets.

D.2.4 SocketListenerThread.java

A helper-class used to receive Update Protocol commands via TCP sockets.

D.3 se.lth.cs.palcom.updatedistributionservice.*

D.3.1 UpdateDistributionService.java

UpdateDistributionService makes it possible to upload PalCom updates in the form of .jar-files and distribute information about the updates to all connected PalComStarters. Other than this, UpdateDistributionService is a simple service which responds to requests from PalComStarters. For example if a client requests update content, the UpdateDistribution-Service responds with the update content if it is available.

Appendix E

Updating sequence diagram

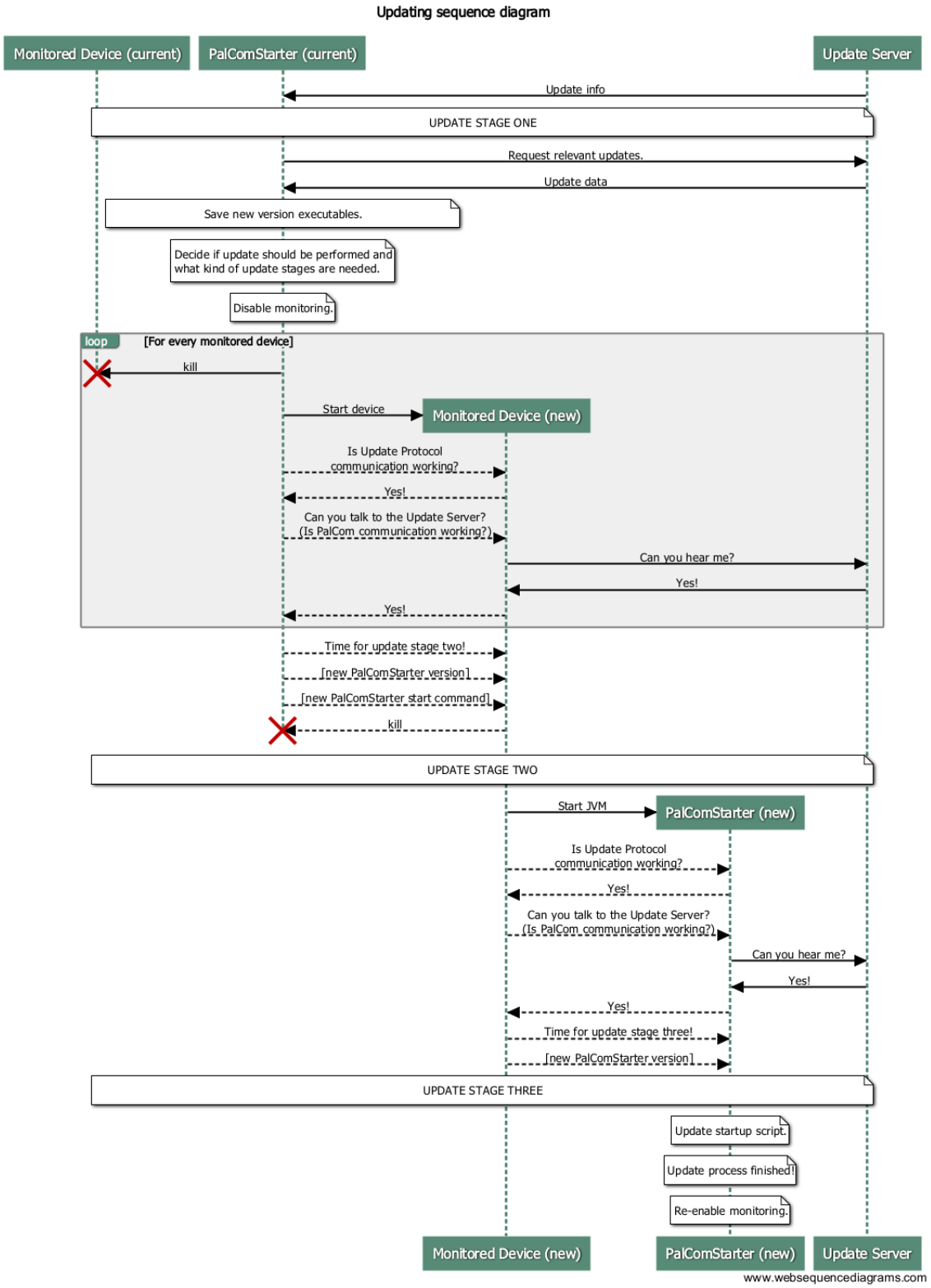

Figure E.1: Sequence diagram describing the update process. Dashed lines represent Update Protocol commands. Solid lines represent PalCom commands and other actions.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-03-18

EXAMENSARBETE Automatic dynamic updating of devices in Internet of Things middleware PalCom
STUDENT Christian Hernvall
HANDLEDARE Boris Magnusson (LTH)
EXAMINATOR Görel Hedin

Låt datorerna uppdatera sig själva

POPULÄRVETENSKAPLIG SAMMANFATTNING **Christian Hernvall**

Det är inte bara våra telefoner som har blivit smarta och börjat koppla upp sig mot internet, utan även våra klockor, vitvaror, TV-apparater och varför inte snart våra husdjur? Men vem är det som ska uppdatera alla dessa saker?

Det fenomen där både våra egna saker och saker runt omkring oss i samhället kopplar upp sig till varandra eller till Internet brukar kallas ”Sakernas Internet”, eller det kanske ännu mer kända engelska uttrycket ”Internet of Things”. En sak kan exempelvis vara en temperaturmätare i hemmet som säger till elementen att öka eller sänka temperaturen, eller en alarmklocka som slår på kaffekokaren automatiskt på morgonen. Dessa saker styrs av datorer som kör program, som behöver uppdateras för att fixa buggar eller för att bygga ut saken med fler och bättre funktioner.

Ericsson har beräknat att det fanns 4,6 miljarder av dessa saker i november 2015 och förutspår att den siffran kommer öka till 15,3 miljarder under år 2021. Med denna ofantliga mängd saker blir det uppenbart att vi inte vill uppdatera alla manuellt. Att uppdatera för hand innebär mycket onödigt arbete som istället kan skötas automatiskt av sakerna själva eftersom de faktiskt innehåller datorer. Datorer är mycket bättre än oss människor på att följa instruktioner och utföra upprepande uppgifter.

Vår lösning gör det möjligt att uppdatera stora mängder av saker automatiskt, utan att användaren behöver röra någonting eller märker att någonting händer. Att uppdatera en sak tar oftast mindre än en sekund, i värsta fall ett par sekunder. Vi har även tagit hänsyn till problemet som uppstår när uppdateringar är protokollbrytande. Att en uppdatering är protokollbrytande innebär att saker som uppdaterats inte längre kan prata med de saker som inte uppdaterats.

Vår lösning är baserad på ”Internet of Things”-lösningen PalCom, en programvara som gör det möjligt för saker att prata med varandra.

Genom att använda vår lösning så finns stor potential att underlätta utvecklingen och förbättringen av dessa saker. I ett samarbete mellan PalCom-projektet och sjukvården har ambulanspersonal fått tillgång till surfplattor med PalCom installerat. Genom PalCom kan ambulanspersonalen knappa in uppgifter om patienten som direkt skickas till sjukhuset, och starta röst- eller video-samtal med en läkare för att snabbt förklara situationen. För tillfället cirkulerar cirka 60 stycken surfplattor ute bland ambulanspersonal. Genom vår lösning kan surfplattorna uppdateras automatiskt, istället för att någon manuellt ska behöva uppdatera varje surfplatta för sig. Den tiden kan istället läggas på viktigare saker, och det blir dessutom möjligt att snabba på utvecklingscykeln genom att uppdatera oftare.