MASTER'S THESIS | LUND UNIVERSITY 2015

Dynamic installation and automatic update of Bluetooth low energy devices in PalCom

Mia Månsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-15

Dynamic installation and automatic update of Bluetooth low energy devices in PalCom

Mia Månsson
ada09mma@student.lu.se

June 8, 2015

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

Chronic kidney disease is becoming an increasingly common illness. The medical background for this thesis comes from the prediction that frequent monitoring of body parameters may help prevent progression of the disease. For other chronic diseases other parameters are interesting to monitor and the need is typically dynamic. This thesis presents a solution for dynamic installation of Bluetooth low energy-enabled health care devices and automatic update of such services in the PalCom middleware. PalCom is aspiring to be a helpful tool in the pursuit of improving the quality of health care and provides functionality for taking measurements at home for remote monitoring. However, as most similar systems, PalCom only allows deployment of predefined BLE devices. The contribution of this work is a solution to install BLE-enabled health care devices in PalCom on demand, as well as a solution for automatic update of these services.

Keywords: PalCom, Internet of Things, automatic update, dynamic installation, Bluetooh low energy

Acknowledgements

I would first of all like to thank my supervisor professor Boris Magnusson for guidance throughout the project and for the invaluable support in writing this report.

I would also like to thank Björn A. Johnsson, Mattias Nordahl, Tommy Alatalo, Knut Mårtensson and Fredrik Nilsask for their input and for answering any questions I have had.

Contents

Chapter 1
Introduction

The work described in this report emerged from the idea of letting patients, who suffer from chronic kidney disease (CKD), have connected devices at the home, taking measurements and continuously presenting those measurements for medical personnel. A careful monitoring may help detect abnormalities sooner and prevent progression of the disease, and consequently enhance life quality of the patients.

The concept of *Internet of Things* (IoT), which refers to the idea of having various type of devices identifying themselves and communicate with other devices, is becoming an important tool in healthcare. Such devices should blend ubiquitously in the background, but still be ready to assist and making life easier for humans. A closely related term is *ubiquitous computing*, which was coined already back in year 1991 by the computer scientist Mark Weiser [20]. While ubiquitous computing refers to having computers everywhere, always present in the background, does *ubiquitous communication* similarly mean "enabling anytime, anywhere communication of anything with anything else"[10]. This work aims at utilize this concepts.

This work is a part of the itACiH (IT support for Advanced Care in home) project. The original focus of the itACiH project was to develop support for home-based medical care for terminally ill cancer patients, but have expanded to include other illnesses, such as support for home-based care for patients with chronic diseases.

1.1 Report structure

The first chapter of this report gives an explanation to the background which leads to the problem statement. In the second chapter I will give an introduction to the PalCom middleware, playing a big role throughout this work. I will also explain the current setup for remote monitoring that is used together with PalCom, as well as some previous work that been giving helpful insights in this work. In the following chapter I'll describe a solution serving as a proof of concept for dynamic installation of BLE devices and automatic update in PalCom. In the fourth chapter I will evaluate the solution. Lastly I will conclude this project and mention some future work.

1.2 Medical background

Medical staff spend a lot of time on paperwork and much of the documentation, for example equipment readings, are first done by pen and paper for later registration in an *electronic health record* (EHR) system. Readings that are not done by pen and paper usually require the use of software from the manufacturer of the device [19]. This force medical personnel to switch between numerous different systems, making it difficult to get an overview of all data belonging to one patient. Having wireless measurement devices connected to a coherent interface, will result in less time spent on paperwork for medical staff, and hence saved resources.

Chronic diseases are becoming increasingly common and also a challenge for health care to manage the increasing number of patients. One example of a chronic disease is Chronic kidney disease (CKD).

1.2.1 Chronic kidney disease

CKD is an irreversible disease and is estimated to be found on 8-16% of the population worldwide [12].

The main task of the kidneys is to filter blood, but they also play an important part in regulating blood pressure levels, red blood cells and concentration of salts and minerals in the body. Without properly functioning kidneys death is only a few days away. [8]

The blood filtration capacity of the kidneys naturally decrease with age, but usually not so rapidly to cause problems. The risk of CKD increases when strain is put on the kidneys, for example diabetes, poor diet or lack of exercise. Treatment is usually focused on slowing down the ever-decaying function of the kidneys. Early identification of CKD can help delay, and hopefully avoid, dialysis and transplantation. [12]

CKD progresses for a long time, not uncommonly for 10-20 years. A progressed stage of CKD may lead to dialysis and kidney transplantation.

1.2.2 Kidney function and blood pressure

Blood pressure levels are closely connected to the function of the kidneys. High blood pressure may indicate low kidney function, which can cause build up of fluid and therefore weight gain. It is essential to maintain low blood pressure as high blood pressure levels will not only further injure the blood filtration in the kidneys, but also lead to a greater risk of cardiovascular disease. [8] Having a frequent monitoring of bodyweight and blood pressure may help to be more proactive in detecting changes in the patients condition. The expectation is that this will prevent, or at least delay, other a lot more intrusive and resource demanding treatments like dialysis and kidney transplantation.

The blood pressure levels vary due to many factors. A measurement taken at one particular point in time may therefore not be representative for the average everyday blood pressure of the patient, for example being in a hospital can be stressful for some patients and temporarily raise blood pressure levels. This is also known as the *white coat effect*. [15]

1.2.3 Taking measurements from home

Patients suffering from CKD or other patients that are at risk or developing it, usually require regular checkups at the hospital in order for medical personnel to take measurements and from the results decide if treatment and medication needs to be adjusted. Between those visits, big changes might occur and the samples made at the hospital may not even be representative of the patient's current condition.

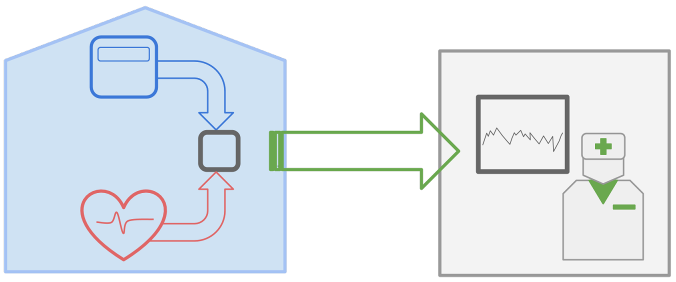


Figure 1.1: Connected devices at home ready to send measurements to medical personnel.

In order to let patients take measurements more frequently, it's not feasible to simply increase the number of visits to the hospital. It would be ideal to have measurements taken regularly every day, and have the measurement instantly available for medical personnel. Putting the equipment in the home of the patients save time both for patients and medical personnel. Such devices may need to be at the patients home for a few weeks or in other cases for months and years. In contrast to manually register measurements, the measurements are sent from the devices at home to a monitoring central. From the monitoring central the data is easily accessible for medical personnel. The system alerts the personnel if there is big changes in the collected values - since this might indicate changed condition and thus the need of medical attention. For patients showing early signs of CKD, it may be enough to measure blood pressure and weight, additional devices at home may be necessary if the disease progresses aggressively.

In the itACiH project, a system for medical personnel at hospitals as well as for mobile care teams is developed. Among other things, the system allows medical personnel to view and write patient data from tablets.

1.3 Problem statement

In order to carry out the idea described above, a solution for installing devices for remote monitoring is needed. When medical personnel brings for example a scale or a blood pressure monitor to the home environment of the patient, the device should "just work". This means that it should be fast and easy to deploy the device to send measurements to a monitoring central. As the number of patients increase, updates and configuration can not be made manually on every single device, instead this needs to happen autonomously.

It is not always known before-hand which kind of device will be used, and this will also most likely change over time. For example as the condition of the patient changes, it may be necessary to measure other body parameters, thus also the need of installing new types of devices.

The primary goal is to investigate how to build a mechanism to dynamically deploy medical devices and perform automatic updates in the context described above.

The main questions to be answered in this report are the following:

- How can Bluetooth low energy devices be dynamically deployed in PalCom?
- How can such services be updated without user interaction?

Security is important in medical care and PalCom support secure tunnels through SSL for confidentiality and integrity of data. However, this was not a part of this thesis.

Chapter 2
Technical background

2.1 PalCom

The infrastructure in figure 2.1 is realized using a middleware called PalCom, the name originates from *palpable computing*, which refers to a tangible system that should be easy for users to understand and use. The system provides communication in the form of services from various devices in a comprehensible way for the user. The aim of PalCom is to allow users, also without previous programming experience, to combine services from devices that originally were not intended to communicate with each other. PalCom is used in the ongoing itACiH-project. The itACiH system is being developed in cooperation with LTH and the university hospital in Lund and includes support for communication between medical personnel at hospital, mobile care teams and medical devices at home. [2] PalCom is implemented in java and was started as an EU project in 2004.

There are a few important concepts in the PalCom system:

Device corresponds to physical hardware that can provide and execute one or more services. A *PalCom* device can for instance be an android device or a PC, but it could also be a simulated device which corresponds to a smaller resource constrained device such as a blood pressure monitor. Each device has a unique deviceID in order to identify itself on the PalCom network. The deviceID will stay the same independently on the addressing of the underlying network technology.

Service has got in commands and out commands which the user can interact with through a PalCom browser or through an assembly. A service can for instance be used to extract and distribute equipment readings from various measurement devices.

Assembly is a user defined set of rules that dictates which services should be used and how they should behave together. In this way services on the same or on different devices can exchange data.

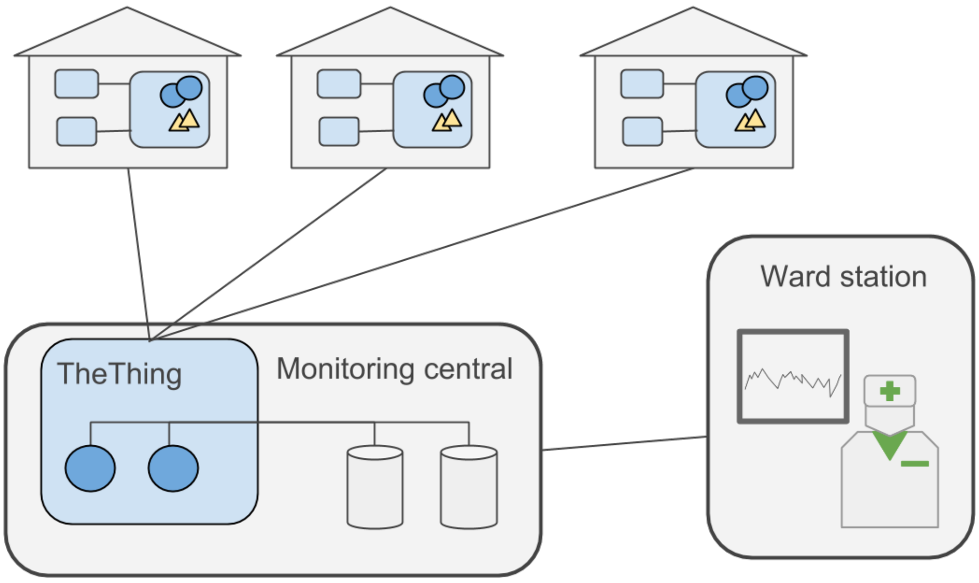


Figure 2.1: The hubs running at the patients' home are connected through PalCom tunnels to a TheThing on a monitoring central. Each hub can then send measurements to a receiver service on the monitoring central for storage (cylinders), these serivces (circles) are connected though assemblies (triangles). An itACiH application retrives and presents the measurements for medical personnel.

TheThing is a java application working as a PalCom device, it can load and run assemblies and services. TheAndroidThing is an equivalent for android, and in this work TheAndroidThing is used as connection point for measurement devices at the patients' home.

PalCom browser is also a java application which is used to explore devices and the services they provide. From the PalCom browser services can be used manually and in our case it can be used by administrators to release new updates. It also serves as a tool for building new assemblies.

The discovery of the devices is based on a heartbeat mechanism, in which a smaller message (i.e. a heartbeat) is regularly sent between devices currently available. To connect devices on different networks, for example over network address translation (NAT), a *PalCom tunnel* can be created. A remote device can then be connected just as any other device on the local network. PalCom also supports routing, which connects groups of devices. Any device that is connected to more than one network may act as a router and then it connects also the other devices on the different networks. [9]

2.1.1 Services

PalCom services reside on PalCom devices and can thus communicate independently of the network technology at hand. [18]. To implement a new service the new java class usually extends the abstract class `AbstractSimpleService` [14].

A service can either be bound or unbound, which means the service is bound to a particular type of device or it can execute on any device and does not depend on a particular hardware.

Services are self-describing thanks to service description, which explain to other devices on the network how to use this particular service. The service description is sent over the network and contains information about available commands, parameters and its serviceID. The serviceID identifies the service and also contains version information. [9]

2.1.2 Assemblies

Different PalCom services can work together through an assembly. An assembly is similar to a script and defines how PalCom services should interact. It takes care of configuration and coordination, in other words, it states which services on which devices that should communicate and how they should communicate. [9]

The feature called *self* can be assigned to a device that is used in a assembly. Instead of pointing to a particular deviceID, self denotes the device that the assembly is running on. In this way the assembly is more flexible and can be moved to work on different devices as long as this device provides the same services that the assembly originally was built for. An example of such an assembly can be seen in appendix A. In PalCom assemblies can be loaded into TheThing using an AssemblyManager.

2.1.3 Dynamic class loading

A service typically consists of a number of java-classes. Dynamic class loading is a way of loading new classes to a running program during execution. In java, classes can be loaded using a classloader. The classloaders are arranged into a hierarchy which dictates in which order to search for a class. If there are two classes with the same name, the class found first will be used. [3]

In PalCom services can be loaded dynamically through jar- and .class-files using a so called ServiceManager. In my solution, this functionality is used both during installation of a new device and during update.

2.2 Bluetooth low energy

In this work, scales and blood pressure monitors communicate through Bluetooth low energy (BLE), which also is known as Bluetooth Smart. BLE is a low power technology released in 2010 and designed for small data transfers, in contrast to the classic Bluetooth which is more suitable for continuous data transfering. [11] BLE is therefore suitable for devices where low power consumption is important and when data transfers are kept short.

A BLE device can play four different roles:

Central role initiates connections to peripherals.

Peripheral role advertises itself and can connect to a central device.

Broadcaster sends non connectable advertising messages.

Observer receives broadcasted advertising messages.

Scales and blood pressure monitors are acting as peripheral devices, while an android device (the hub), is acting as a central device.

The structure is a star topology with a central device and one or many peripheral devices. The central device can be connected to several peripheral devices, while a peripheral device can be connected to only one central device.

BLE is built around the generic attribute profile (GATT), which defines how devices exchange data. To identify what kind of functionality a device can provide, the following categories are defined:

Profile is a set of services.

Service contains characteristics.

Characteristics describe what data that can be sent and received.

Descriptor contains additional information about a characteristic, for example the unit of a measurement.

In order to extract data from for instance a scale, the extractor, in our case a PalCom service, must know in beforehand which GATT-services and characteristics that will be present on that particular scale.

The BLE standard allows for communication in *unpaired mode* as well as *paired mode*. The paired mode does provide encryption and authentication. In Bluetooth version 4.1 the key exchange protocol is flawed and susceptible to passive eavesdropping and man-in-the-middle attacks, unless an out-of-band method is used. [16] In December 2014 version 4.2 was released and should now provide better protection against such attacks. [17] In the prototype used in this work, Bluetooth version 4.1 is used and connections between periopherals and hubs are made in an unpaired mode. In future versions, it would of course be preferred to use Bluetooth version 4.2 in a paired mode.

2.2.1 Bluetooth low energy on Android

Android 4.3 (Jelly Bean) provides an API for BLE that allow applications to communicate with BLE devices. The API does not support abstractions on GATT profile level, instead services and characteristics are explored using callbacks from the API. Jelly Bean only provides API for implementing central- and observer-devices. Since android version 5.0 (Lollipop) there is also support for implementing broadcaster devices and peripheral devices.

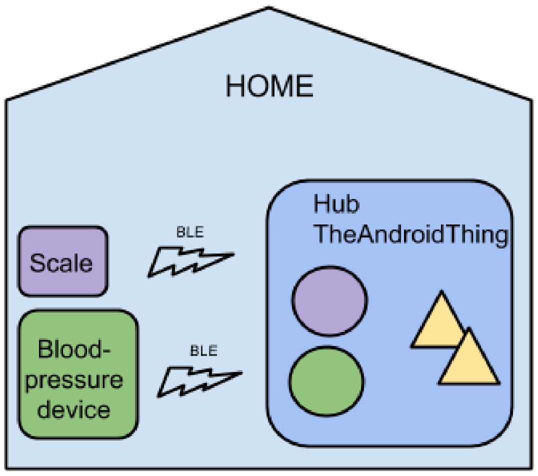


Figure 2.2: Simplified architecture of the setup of the hub in the kidney project. Each BLE peripheral is connecting to PalCom through a intepreter service (circles), which in turn is used by assemblies (triangles) to send data to a monitoring centeral.

2.3 The setup

Through the itACiH-project a pilot study was conducted, letting patients have a BLE scale and blood pressure monitor at home using PalCom. A few patients were selected to test the system at home, an overview of this setup can be seen in figure 2.2. On an android tablet - *the hub*, runs TheAndroidThing which detects BLE peripherals, in our case various measurement devices. The peripherals are typically not a PalCom device, thus in order to communicate with a peripheral, a dedicated PalCom service has to act as an interpretor. Apart from services, the hub also needs assemblies to connect each service with a receiver service on a TheThing at the monitoring central (fig. 2.1). In the pilot study, BLE-services and assemblies on the hub were hard coded in a branch of the TheAndroidThing. If a new type of peripheral is to be connected, a new version of TheAndroidThing has to be installed.

2.4 Related work

Companies such as Withings and Beurer, provide wireless solutions for health monitoring at home, but their solutions normally support feedback for private use and gives feedback to the user only, in contrast to transmitting and presenting measurements to medical personnel. Such solutions typically have predefined devices that can be connected, thus no support for dynamic installation of devices.

Continua [1] defines standardizations and guidelines for personal health systems. They also certify devices, for example sender and receiver devices, but at this point only few types BLE devices, such as a blood pressure monitor, but not yet a BLE scale. In june 2015 new guidelines will be released about how to reduce pre-configuration on a device in order to obtain plug-and-play interoperability.

There exists some well-known systems that provide automatic updating, for example browsers like Chrome and Firefox. Google for example, argues that it is not the user's responsibility to keep software up to date. In this work we take a similar approach - neither patients nor medical personnel should be involved in software updates. Since Android 2.3 applications have the possibility to automatically receive updates through Google Play. With their solution the entire application has to updated, in this work we only want to update a small portion of the application.

When updating only certain components of a system, dependency conflicts may arise. S Eisenbach et al. [7] and R. Andersson [4] discusses DLL (Dynamic link libraries) hell in Windows, but the more general term *dependency hell* arises in other situations as well. The problem occurs when a configuration needs a new version of a shared module, in our case a PalCom service, while other configurations still require the old version. This cause the configuraions using the old version to fail. Solutions to this problem includes maintaining a versioning system, protecting the module from being overwritten or using two versions side-by-side

U. Asklund et al. [5] presents a solution to this problem using the unified extensional version model, which is utilized in this work. The model maintains a versioning number both for entities and the configurations using the entities. The article explains how versions can form an directed acyclic graph of modules and configurations, and how changes in entities propagate through the configurations. A key feature of this model is that it is possible to have different versions of components coexisting at the same time.

R. Bialek et al. [6] have evaluated the functionality and the steps needed to perform dynamic updates. According to their study, components should preferably be updated when they are not operational, but in some cases, with very long execution times, it may be necessary to perform a hot swap. In such cases it is required for the programmer of the component to denote where in the execution it is safe to perform the update. This concept have been utilized in this work in order to decide when to switch to a new version of a PalCom service.

Chapter 3
Approach

This work was carried out using an evolutionary approach, where an initial idea is discussed and analyzed. Through an iterative process the work was improved and refined, and at the end of each iteration the project was evaluated in order to decide how to take the next step. As a first step I implemented a prototype to get a grasp on PalCom and to get a fundamental understanding of how to identify, store and load services in PalCom. As a second step I implemented my idea on actual BLE devices and BLE services originating from the pilot study (section 2.3), where only preinstalled BLE devices was used. In this chapter I will analyze the problem in greater detail, describe the result and some of the design choices that where made during the implementation.

3.1 Analysis

The data sent from a peripheral, in our case a measurement device, may either be interpreted remotely or interpreted locally. If the interpretation takes place remotely, the hub would act like a mediator between the peripheral and the remote interpretor. The hub would be able to communicate with any peripheral as long as the remote interpretor knows how to communicate with this particular type of peripheral. This solution is however not suitable for our problem, because the hub and the peripheral will be located at the patients home, and constant Internet connectivity can not be guaranteed. In offline mode it is not always enough to simply store the received data from a peripheral and send it to the interpretor when online again, because whenever the hub is expected to send messages to the peripheral the communication may fail. For instance, the Beurer BF800-B scale used in this thesis expects to receive certain protocol specific data in order to continue transmitting measurements to a central device. For this reason, the interpretation has to take place locally, which leads us to having a solution where services are distributed on demand. With this solution, the hub would of course still need access to Internet when it detects a new type of peripheral, but there after it would not depend on Internet connection in order to receive measurements. Instead, when the hub is offline, measurements can be stored and sent to the monitoring central later.

In order to install a BLE peripheral on the hub, we need a PalCom service which acts as an interpreter maintaining the connection to the peripheral, and an assembly connecting the interpreter to a receiver service on the monitoring central. When a new version of a BLE service is to be deployed, also a new assembly is necessary.

In the following two sections, I will analyze different scenarios of how installation and update can happen.

3.1.1 Dynamic installation

Dynamic installation of services and assemblies can happen in a few different ways. An assembly is in our case mapped to a group of hubs and a particular BLE peripheral. An other point to note is that each assembly uniquely specifies the services it is using. Whenever an assembly is deployed on a device, the services it needs may already be present, not present or partially present. In case a service is missing, a copy of that service needs to be installed on the device, in our case this is typically bound services that need to execute locally.

3.1.2 Coordinating versions

When updating assemblies and services, the most natural procedure would be to release a new service, and later build the assembly where the new service is used. Lastly the new assembly will be released and distributed to the hubs. Using this approach, the hubs will first load the new version of the service and then receive the assembly.

Three approaches that may be used are:

1. **Deploy the new version and stop the old**

   If we immediately stop the old version and deploy the new version of a service, assemblies using the old version will stop working, and the application will not work as expected until all the assemblies for the new version arrive.

2. **Wait to deploy the new version** Let the old version of a service run until all assemblies for the new version have arrived, see figure 3.1. When all assemblies for the new service are present, we switch to the new version and stop the old. This approach may cause delay until a new version of a service can be deployed, as it can not be assumed that all assemblies using a specific service will be updated at the same time.

3. **Have different versions running simultaneously** In this way different service versions may be used simultaneously, which allows for a gradual update where different assemblies may use the two versions, see figure 3.2. However, for some bound services i.e. services depending on a certain kind of hardware, it is not always possible to have more than one service at the time using the hardware.

Having several versions running at the same time does not completely solve the problem for BLE services. Although PalCom allows different versions of the same service to

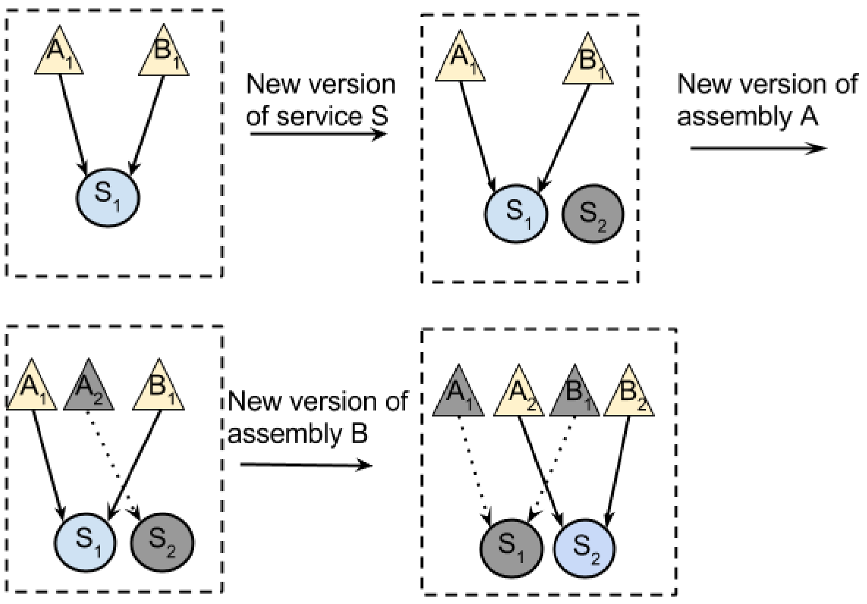


Figure 3.1: Switching to version two of service S when all assemblies (A and B) have been updated and assemblies and services in version one are stopped (dark grey).

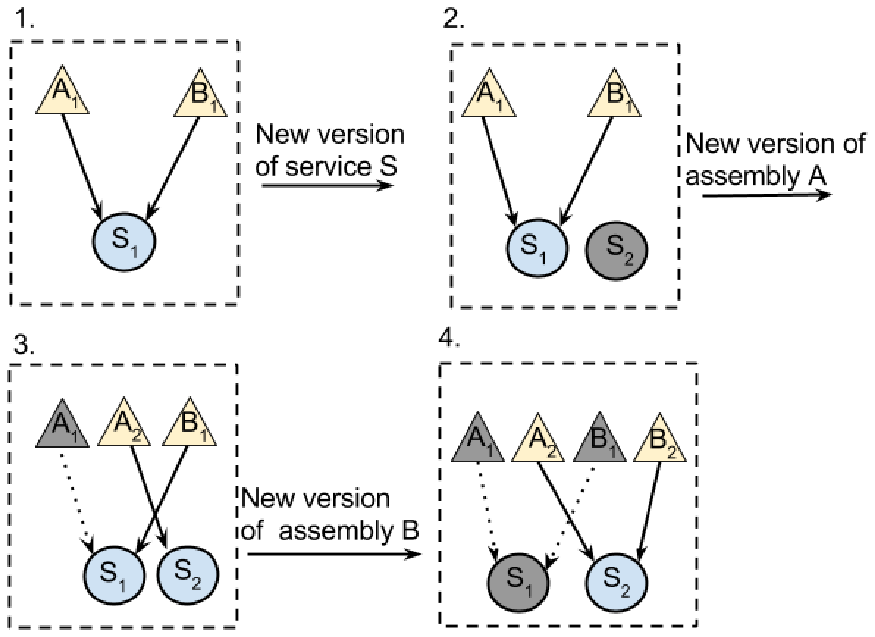


Figure 3.2: Switching to new versions when two versions are executing simultaneously. Notice in step three, where A_2 uses the new version of S while B_1 still use the old version. Having two versions running simultaneously is not always meaningful for some bound services such as BLE services, where only one of the two versions can be connected to the BLE peripheral.

run simultaneously, for BLE services only one of them can be connected to the BLE peripheral. In our case a BLE service is mapped to one assembly, and we can use the rather straight forward approach depicted in figure 3.3. Service T is depicting a service running on a remote device, while an assembly (A1) and a service (S1) are running locally.

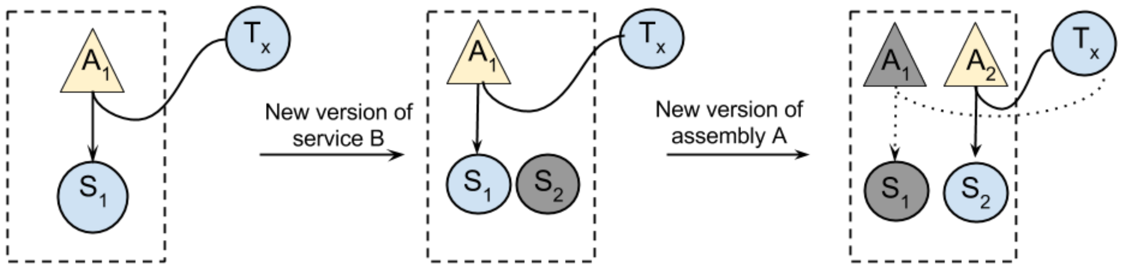


Figure 3.3: Procedure when updating a service (S). In this case only one assembly is using the services.

A BLE service should not disconnect itself from its peripheral until it is safe to do so, i.e. when no assembly is using the service. On the other hand, a BLE service should not connect to a peripheral until there is an assembly using it, which means a PalCom connection have been opened to the service.

3.2 Implementation

In this section follows a presentation of an implementation for dynamic installation and update of BLE services in PalCom. This implementation serves as a proof of concept to the problem statement.

3.2.1 Architecture of the proposed solution

An overview of the implemented solution is depicted in figure 3.4. I implemented two PalCom services `RemoteUpdateService` and `RemoteBLEAssemblyService`, which can run on a PalCom device and supply other PalCom devices, in this case the hub, with services and assemblies. The commands belonging to these services are listed in appendix C.

On the the hub I implemented functionality for managing installation and update of services and assemblies; a BLEManager and UpdateManager, both also consist of a PalCom service. The BLEManager existed already in the pilot study described in section 2.3, but have been modified in this implementation.

This solution is utilizing the AssemblyManager and ServiceManager in PalCom and listen to resources from the PalCom network through the DiscoveryManager.

In order to install a new BLE peripheral in PalCom and configure it to send measurements to a given receiver, two PalCom components are needed; an assembly and a service. The required service is the BLE service that manages communication with the BLE peripheral. The assembly script contains instructions about where to send measurements collected from a peripheral.

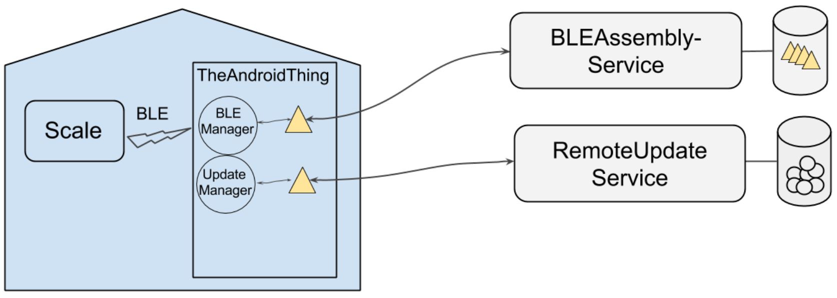


Figure 3.4: An overview of the main components of the implementation. Where assemblies (triangles) and services (circles) are distributed on demand.

3.2.2 Remote services

A storage and distribution mechanism is needed to distribute services and assemblies to the hubs. For this the PalCom services `RemoteUpdateService` and `BLEAssemblyService` was implemented. These two services may run on the same device or on different devices. The reason why these two functionalities are kept separate is that each assembly already uniquely specifies which services it needs (in contrast to a service which provides functionality, independently of where it is used).

Both services store all their data in the PalCom file-system, which allow them to run on any TheThing or TheAndroidThing.

Service distribution

`RemoteUpdateService` has got commands for administrators to store, test and release services, as well as commands for invoking service requests and receive jar-files. The service is taking advantage of the discovery mechanism in PalCom, similar to how the PalCom browser detects devices and explore their services. `RemoteUpdateService` can then determine which services are running on which device, and will thus only send push-notifications to those devices affected of an update.

A new version of a service should be stored within an instance of `RemoteUpdateService`. It can then be tested by some selected users, and later be released. Update notifications are sent in the following scenarios:

1. A new version of a service is released - notifications are sent to all devices using an old version of the service.
2. A device connects - a notification sent to this newly connected device if an old version of a service is executing on the device.
3. A service becomes available on a already connected device - a notification is sent to the device the detected service is executing on, if there is a newer version of the service available.

In PalCom we want to maintain the distinction between the functionality a service provides, and configuration and coordination in assemblies, which is why `RemoteUpdateService` is only allowed to send updates to devices that are already connected to this service through an assembly.

Assembly distribution

`RemoteBLEAssemblyService` has got commands for administrators to store assemblies and commands to provide devices with assemblies. A device which detects an new BLE peripheral should send an assembly request containing the name of the peripheral and its hub name. If the service has got a matching assembly stored, it sends this assembly to the device.

3.2.3 The hub

`BLEManagerService` is a service running on TheAndroidThing on the hub, and is working as a gateway for BLE communication. It scans for BLE peripherals and if it finds an unknown peripheral, it initiates the installation by sending an assembly request for the found peripheral. When the assembly arrives, it is loaded using the AssemblyManager. If the BLEManager discovers an already installed BLE peripheral, it redirects it to the corresponding BLE service, which then initiates a connection to the peripheral.

An UpdateManager with a corresponding PalCom service is also implemented on the hub in order to provide functionality for managing services. It is responsible for the second step in the installation; ensure that all services needed are present. The UpdateManager is notified whenever an assembly is started on the device, and by parsing the assembly, it can list which services are missing on the device. It then sends requests for services not yet present, and when a service arrive, it will load it using the ServiceManager in PalCom. The UpdateManager listens to events from the discovery manager in PalCom in order to gather information about services executing locally.

Performing updates

In PalCom multiple versions of services can execute simultaneously, which then allows for gradual updates, as explained in [5]. This does not solve the problem for bounded services, such as BLE services. Even though it is possible to have two versions of a BLE service executing, only one at a time can communicate with the peripheral. In this implementation, the services themselves take care of synchronizing the start and stop with their BLE peripheral, depending on whether there is something using it, that is if there is an assembly connected to the service or not.

A BLE service therefore terminates the communication with its BLE peripheral when there is no PalCom connection to it. The different operating states of an BLE service is depicted in figure 3.5. The BLEManager stops the old version of an BLE-assembly when a new version arrives. This cause the old BLE service to stop, and allow the new service version to initiate a connection to the peripheral.

In order to minimize the functionality in each subclass of `BLEService` to only include communication and interpretation with a specific type of BLE peripheral, I imple-

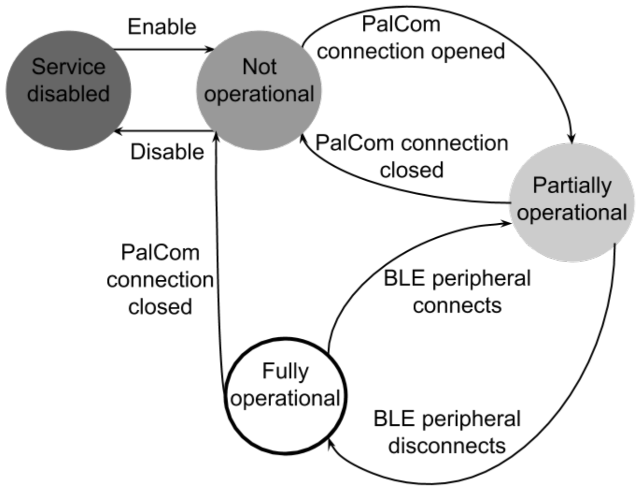


Figure 3.5: The different operating states of a BLE service.

mented the synchronizing of start and stop of the BLE connection in the super class `BLEService`. This class in turn extends the `AbstractSimpleService`. A BLE service should therefore extend this class instead of the `AbstractSimpleService`, which normally is the class to extend when implementing a service in PalCom.

3.3 Using the solution in practice

In this chapter I will show a concrete example of an installation and update of a BLE scale (Beurer BF800-B) in PalCom.

The setup is depicted in figure 3.6. Both `RemoteBLEAssemblyService`, `RemoteUpdateService` are running on a remote device acting as a server, and a PalCom tunnel is set up between this server and the hub. The BLEManager and UpdateManager and their corresponding services are running on TheAndroidThing on the hub.

A PalCom tunnel is also prepared between the hub and a monitoring central, which is where measurements from BLE peripherals should be sent.

3.3.1 Installation of the Beurer scale

Figure 3.7 illustrate the service view in TheAndroidThing on the hub, and also the corresponding state seen from the PalCom browser. Here we can see the two assemblies that are preloaded on the device. One assembly is connecting `UpdateManagerService` to `RemoteUpdateService` and the other is connecting `BLEManagerService` to `RemoteBLEAssemblyService`. `BLEManagerService` and `UpdateManagerService` are loaded internally as a part of TheAndroidThing, such services are not displayed in this view, but if we look from the PalCom browser, we find those services under the android section.

In picture 3.8 the Beurer scale has been detected by the BLEManager, which then sent an assembly request to `RemoteBLEAssemblyService` in order to receive and load the assembly for the Beurer scale.

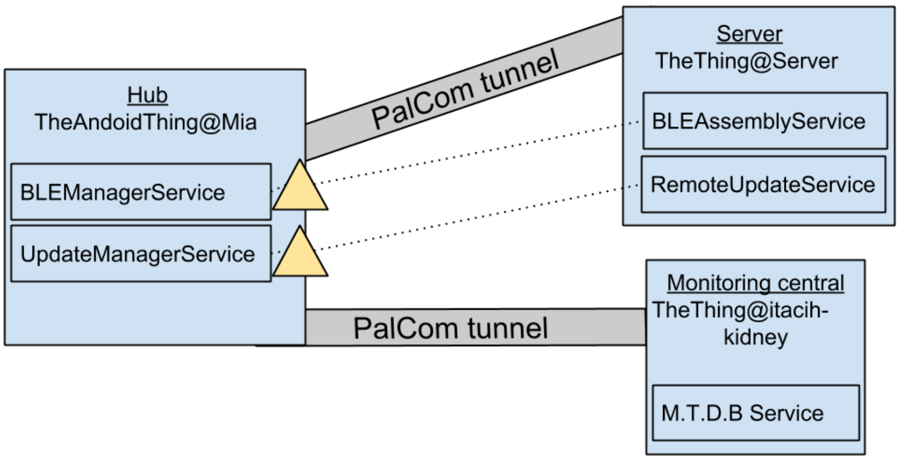


Figure 3.6: The setup prepared for automatic installation and update of BLE services. The two triangles indicate assemblies connecting the services on the hub to their corresponding service on the server.

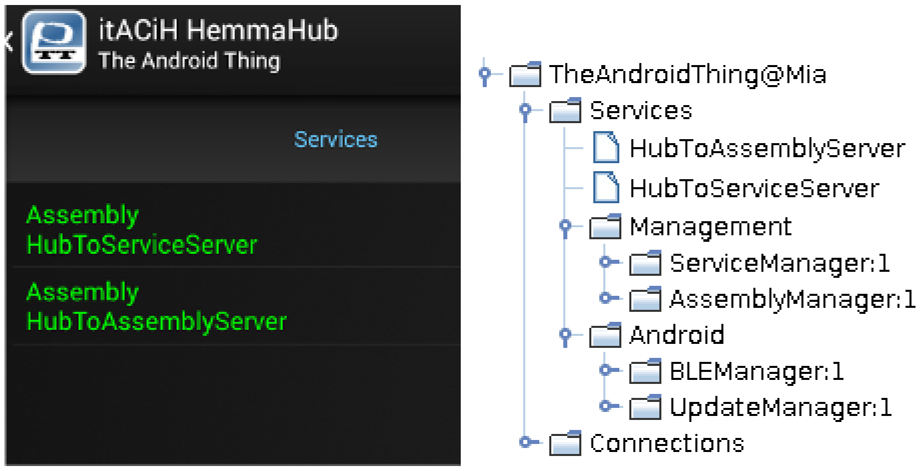
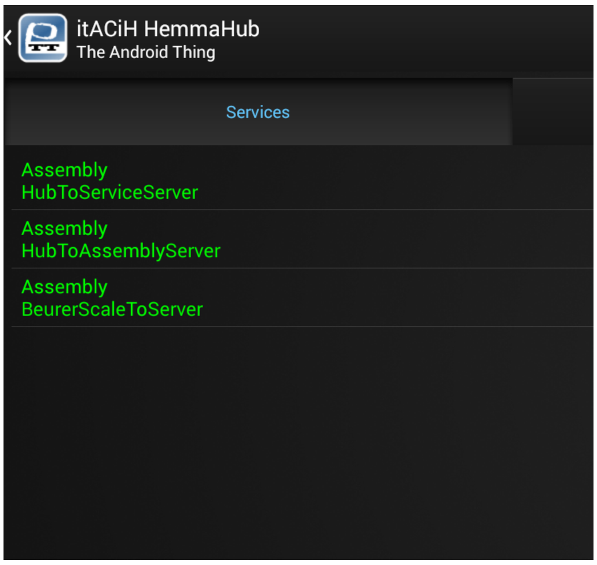


Figure 3.7: Inital view when no BLE service is installed. Two assemblies have been loaded to connect to the remote services. To the left is the view from TheAndroidThing on the hub, and to the right is the corresponding view from the PalCom browser.

Next, the UpdateManager checks the received assembly for missing services that need to run locally. In this case, a service called `BeurerScaleService` will be missing. The `UpdateManagerService` sends a request for this service to the `RemoteUpdateService`. In figure 3.9 the service has arrived and by the color of it (orange) we can see that it is currently only partially operational - it have not yet initiated communication to the scale, but there is an assembly connected to this service, in this case from the "BeurerScaleToServer" assembly. A short moment later the service has established a connection to the BLE peripheral, and in the view the color of the service has switched from orange to green and is now fully operational.


itACiH HemmaHub
The Android Thing

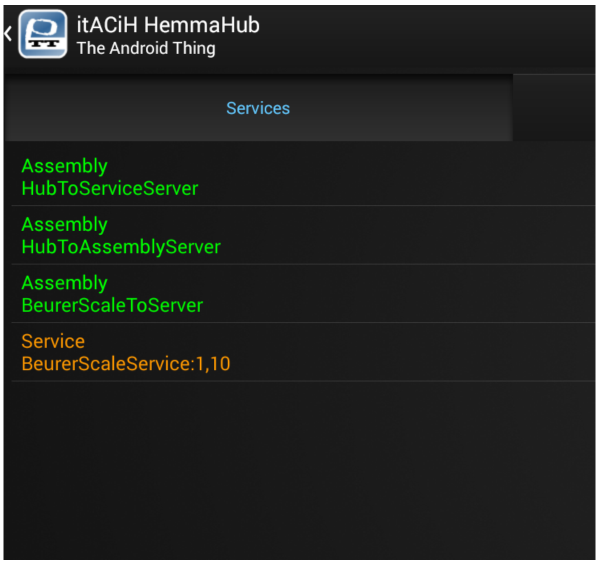
Services

Assembly
HubToServiceServer

Assembly
HubToAssemblyServer

Assembly
BeurerScaleToServer

Figure 3.8: An assembly has arrived for the Beurer scale (BeurerScaleToServer).


itACiH HemmaHub
The Android Thing

Services

Assembly
HubToServiceServer

Assembly
HubToAssemblyServer

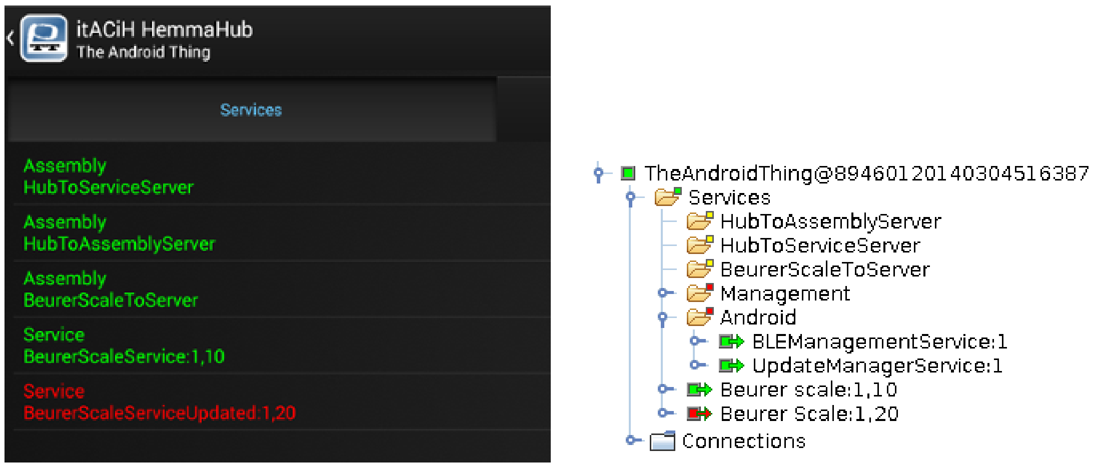
Assembly
BeurerScaleToServer

Service
BeurerScaleService:1,10

Figure 3.9: The BLE service for the scale has arrived. Since the BeurerScaleToServer assembly uses this service, the state of the service goes from not operational (red) to partially operational (orange), and after the connection is setup up to the scale, fully operational (green).

3.3.2 Updating the Beurer scale service

When we want to update the Beurer scale service, we store the new version in the `RemoteUpdateService` on the server device. When releasing the service, `RemoteUpdateService` will send update notifications to connected devices that are using older versions of the service. The hub in our example will receive such a notification, which will cause it to reply with a service request in order to receive a jar file containing the new version. In figure 3.10, the new version has arrived, but is not yet in a operational state, since there is no assembly connected to it. The old version is still maintaining the connection to the scale.



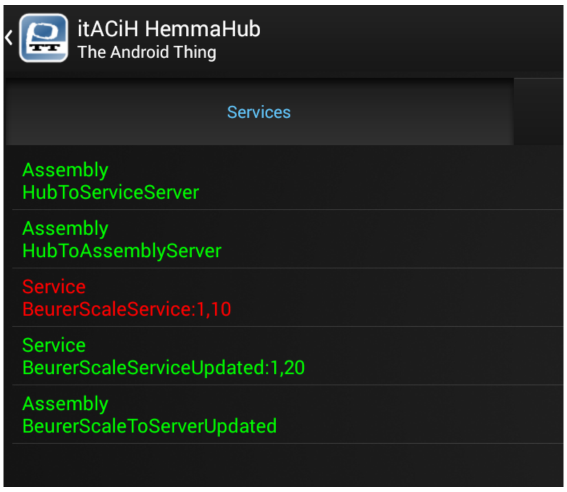
```
itACiH HemmaHub
The Android Thing
Services

Assembly
HubToServiceServer
Assembly
HubToAssemblyServer
Assembly
BeurerScaleToServer
Service
BeurerScaleService:1,10
Service
BeurerScaleServiceUpdated:1,20
```

```
TheAndroidThing@89460120140304516387
  Services
    HubToAssemblyServer
    HubToServiceServer
    BeurerScaleToServer
    Management
    Android
      BLEManagementService:1
      UpdateManagerService:1
    Beurer scale:1,10
    Beurer Scale:1,20
  Connections
```

Figure 3.10: A new version of `BeurerScaleService` is loaded and currently in a non-operational state, because no assembly is connected. The prevous version is still fully operational and thus connected to the scale. To the right we see the same hub from the PalCom browser.

When administrators have built the assembly for the new service version, the new assembly will be stored and available on the `RemoteBLEAssemblyService`. The BLEManager on the hub receives a notification that a new assembly for the Beurer scale is released. It then compares its current version of the Beurer scale assembly and send an request for the new version. When the new assembly version arrives, the BLEManager stops the previous version of the assembly and start the new.

The new assembly will connect to the new version of the service which then goes into the partially operational state. Remember that the BLEManager stopped the old assembly, so our old service does no longer have any assembly using it. This forces it to close the connection to the scale and go into a non operational state, thus letting the new version connect to the scale.



Assembly
HubToServiceServer

Assembly
HubToAssemblyServer

Service
BeurerScaleService:1,10

Service
BeurerScaleServiceUpdated:1,20

Assembly
BeurerScaleToServerUpdated

Figure 3.11: The update of the Beurer scale service is now completed. Version 1.10 is in a non-operational state (red) while version 1.20 is fully operational (green) and has taken over the communication with the scale.

Chapter 4
Evaluation of the implementation

In this chapter I will evaluate the implementation described in the previous chapter. First I will discuss how my solution solved the problem statement. I will evaluate changes from administrator perspective as well as from user perspective. I will also measure the installation time of a BLE peripheral and mention some scalability aspects.

The questions to be answered was:

- How can bluetooth low energy devices be dynamically installed in PalCom?
- How can such services be updated without user interaction?

4.1 Dynamic installation

The BLE peripherals are dynamically installed using the two remote services described in the previous chapter. When a hub detects a new type of BLE peripheral, it requests the remote assembly service in order to receive the assembly that is associated with the BLE peripeheral in question. From the assembly, the device determines which service and version is required. If some service is missing, the device can ask other connected PalCom devices whether they have the required service or not. In the current implementation there is not yet support for connecting two identical peripherals, i.e. peripherals using an instance each of the same PalCom service.

`RemoteUpdateService` will provide the requested services including the BLE service acting as a intepretator between PalCom and the BLE peripheral. From here on the BLE peripheral is ready for use together with the hub.

The paring procedure from the users perspective is no different from when the BLE peripherals were pre-installed, apart from some delay when receiving the necessary services and assembly from the remote services. The magnitude of this delay will be evaluated in a following section.

From an administrators perspective it will not anymore be required to update the entire TheAndroidThing in order to deploy a new BLE peripheral. Instead all BLE services and assemblies are stored once within the remote services. This enables the non pre-planned deployment of BLE peripherals, where services and assemblies are retrieved on demand.

4.2 Automatic update

Whenever an update is ready for release, the new version of the service and the assemblies using it can preferably be stored in one central server executing `RemoteAssembly-Service` and `RemoteUpdateService`. The server distributes the new versions to all connected devices that are using older versions of the service.

The BLE services are used by one assembly, this together with the explicit version numbering of assemblies helps the hub to manage different versions in a consistent way.

One can argue that it is a weakness that this solution only consider one assembly per peripheral. The decision to only have one assembly connecting was originally something that was implied by the user case in the pilot study, since there was only one assembly per peripheral. In hindsight the system would be more versatile if it was prepared for managing several assemblies. But as long as only one version of a service can be used at a time, the new version of the service can't be deployed until also all assemblies are updated. One possible solution could be to have several BLE services communicate with the same peripheral as if they where connected to the same peripheral, when in fact only one of them is. The other services will instead communicate with the BLE-peripheral through the BLE service that is connected to the peripheral. Then several versions of a BLE service can run simultaneously, and updates then can happen gradually instead of waiting until the old service is not anymore in use.

4.3 Installation- and update preparations

The two services `RemoteUpdateService` and `RemoteBLEAssemblyService` will have to be deployed on a PalCom device and all other PalCom devices interested in receiving updates should have an connection, i.e an assembly, to those services. When a new service is to be released, the programmer may first want to test the service among some selected users. This is possible when the new service is added to the `RemoteUpdate-Service`. It will then only distribute the service to selected devices, see the manual in appendix C.

When adding a new service to the `RemoteUpdateService`, the serviceID has to be manually added. A chain is only as strong as its weakest link, and a better solution would be to not leave this task to humans, but perhaps let the service deduce the serviceID in some other way from the jar-file.

Administrators will also have to supply the name of the BLE peripheral together with the assembly in order to store the assembly in `RemoteBLEAssemblyService`.

Before the hubs are ready for use , TheAndroidThing needs to be installed and in addition, assemblies to connect to the remote assembly- and service- storage must be loaded. Loading assemblies during installation cause some slight extra work. Those assemblies only need to be loaded once, and the same assembly can preferably be used by all hubs that should be accessing the same storage. An other possibility is to attach such assemblies in the assets directory in the apk-file.

4.4 Installation time

To get an indication of the deployment time when a BLE device is brought home to a patient, I measured the installation time of a BLE peripheral. I let the hub use a 4G network to connect to the remote storage services and then I measured the installation phase from when a BLE peripheral is detected, until it is installed in PalCom. For this experiment I used the same Beurer scale as in the using scenario in section 3.3. Note that these measurements are meant as an indication of the delay when deploying services dynamically at the patients home, and not as an exhaustive evaluation of the efficiency.

I divided and measured the installation in four different steps:

Step 1 Duration from when the assembly is requested, until the assembly arrives.

Step 2 Duration for loading the assembly, including sending request for one missing BLE service.

Step 3 Duration from when the BLE service is requested, until it arrives.

Step 4 Duration for loading the BLE service.

In five tries the installation time was on average 3.3 seconds, see table 4.1. The size of the assembly that was transmitted was 5.7 KB and the jar file containing the service was 7.3 KB.

Run number	Step 1 (ms)	Step 2 (ms)	Step 3 (ms)	Step 4 (ms)	Total (s)
1	404	88	243	3093	3.8
2	587	130	863	1776	3.4
3	893	123	185	1895	3.1
4	296	179	394	2391	3.3
5	290	155	513	1999	3.0

Table 4.1: Installation time of five different tries over a 2.5/1.5 Mbit/s connection.

From the result in table 4.1, we can conclude that loading the service is the bottleneck. If we wanted a faster installation, it may help to request the service as a first step and while loading the service, sending request for the assembly. For this to work, the structure of both the remote services and the hub would have to be slightly different from my implementation, since the installation steps would be reversed. However, seen from the users perspective the measured installation times is fast enough.

Chapter 5
Conclusions

Improving health care is an important topic and there is still many concepts of the IoT that may be utilized in health care. In this work we investigated how to simplify the installation- and update process of BLE devices to be used in the patients home, and how it can be done in a fairly straightforward way using PalCom. Using the solution presented in this report enables non-preplanned deployment of medical devices at the patients home through an automatic installation process. The installation time was measured and in five tries the installation took on average 3.3 seconds.

The automatic update solution allows for a more convenient usage of the installation, where both services and assemblies can be automatically updated. The solution updates services and assemblies using the unified extensional version model [5]. To overcome the limitation that only one version of the same BLE service can be connected to the same peripheral, I used the concept described in R. Bialek et al. [6] where the programmer of the component is responsible for denoting where in the execution the update should be performed. In this solution, the components translates to PalCom services. For BLE services this was solved by letting the service itself close the connection with its peripheral when there is no PalCom connection to the service, since this means that it is not used, and should thus allow the new version to execute.

5.1 Future work

This work focused on BLE devices and installation and update of BLE services in PalCom.

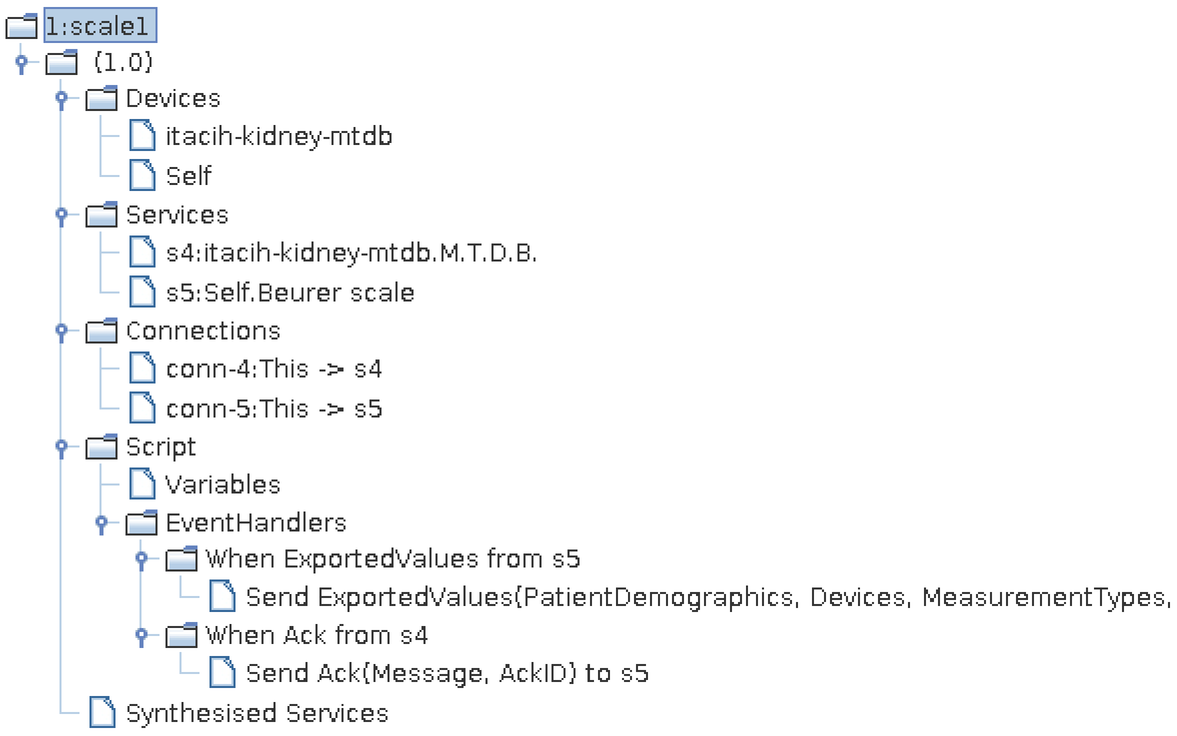
Integrating my solution with the PalCom kernel is left for future work, but there is still work to be made in the area of automatic updates of assemblies and services. For instance, update several assemblies and not only consider one assembly per peripheral. Assemblies may also be used by other assemblies, and this leads to updating tree structures of assemblies, where the PalCom services can be seen as leaf nodes. For this a general naming convention for assemblies, similar to serviceID for services, would be beneficial.

To use my solution on a larger scale it could be helpful to have tools for the configuration management process, for example some kind of serviceID generation tool.

The BLE services in PalCom implemented this far have used no pairing, thus no authentication of BLE peripherals, and little integrity and confidentiality of data. In future versions of BLE services, a pairing mechanism, such as Near field communication (NFC) may be necessary. This could be a way of providing better security, but still demand little user interaction.

Appendices

Appendix A
Assembly



```
1:scale1
  (1.0)
    Devices
      itacih-kidney-mtdb
      Self
    Services
      s4:itacih-kidney-mtdb.M.T.D.B.
      s5:Self.Beurer scale
    Connections
      conn-4:This -> s4
      conn-5:This -> s5
    Script
      Variables
      EventHandlers
        When ExportedValues from s5
          Send ExportedValues(PatientDemographics, Devices, MeasurementTypes,
        When Ack from s4
          Send Ack(Message, AckID) to s5
    Synthesised Services
```

Figure A.1: An example of an assembly seen from the PalCom browser. This assembly is connecting the Beurer scale BLE service with a reciever service (M.T.D.B) running on a remote PalCom device.

Appendix B
Equipment

The following equipment was used in this thesis.

Hub Sony Xperia Z2 tablet (KitKat)

Scale Beurer BF800-B

Blood pressure monitor Beurer BM 85

Appendix C
Administrator guide

When setting up the servers for storing assemblies and services, the `RemoteBLEAssemblyService` and `RemoteUpdateService` should be loaded on a PalCom device, for instance TheThing, see TheThing users manual [13]. Through a PalCom browser, you can invoke the commands described in the following sections.

C.1 Implementing BLE-services

By extending the class BLEService, new BLE services can be created. They can be implemented as any other PalCom services, and by following the usual ServiceID naming convention of PalCom:

A serviceID consists of the following space separated parts, where each part contains a device-ID, an unique running number and initials of the developer:

1. Creation-ID: this is assigned in the first version of a service and is never changed for future versions.
2. Update-ID: is also assigned in the first version, but will be changed for for each new version.
3. Previous-ID: in every new version, this reflects the previous version. For the first version this is set to null.
4. Merged-ID: is used if a new version is a merge from two previous versions, otherwise null.

All start and stop coordination with the BLEManager in PalCom are implemented in the super class and does not need to be handled in the new class.

If you want to deploy the service using either my solution or manually through the service manager in PalCom, the constructor of the new service must follow usual pattern for services in PalCom:

```
public BeurerScaleService(AbstractDevice device, String instance){
super(device, SERVICE_ID, PROTOCOL, VERSION, TAG, instance,
             HELP_TEXT, PERIPHERAL_NAME);
```

The peripheral name that is handed over to the super class should be the name that the peripheral present itself with during advertisement, this is also the name that is used by the `RemoteBLEAssemblyService` to map a BLE peripheral to the correct assembly. Callbacks for handling characteristic reads and write etc. should be overridden from the super class `BLEService`.

C.2 RemoteUpdateService

This service store and distributes PalCom services. The following commands are available:

C.2.1 In-commands

Store service (creationID, updateID, previousID, mergeID, versionName, jar file) This command takes a jar file containing a service and store it with the associated serviceID and versionName.

ServiceID consists of `creationID`, `updateID`, `previousID` and `mergeID`. `creationID` and `updateID` are obligatory while `previousID` and `mergeID` may be left empty depending of the version history of the service. `versionName` should be entered in digits in format X.XX (major, minor). The version having the highest number is interpreted as the latest version. The service will response with a `Store response`. If the service was successfully stored, it will be possible to test the service. The service may be replaced as long as it is not released, thereafter can no changes be made to this particular version of the service. After the service is tested, the service can be released with the command Release service.

Release service (creationID, updateID, previousID, mergeID) This command can be called for any service that is stored with the serviceID consisting of `creationID`, `updateID`, `previousID` and `mergeID`. The command results in a **New version(serviceID)** being sent to all connected devices that are using an older service, as well as a status message sent to the invoker.

Get service (serviceID, isTesting) Responds with **Service(jar file, service ID)** or **NoService** if the requested serviceID could not be found. To receive non-released services, give the "isTesting" parameter any value $\neq 0$, otherwise leave empty or set to 0. This command is typically used in assemblies for automating the installation of services. For simplicity when building assemblies, should therefore the full `serviceID` used, with space separated versionparts i.e. `creationID updateID prevousID mergedID`.

C.2.2 Out-commands

The out-commands that the service send in reply to the incoming commands are the following:

Store response(Status message) This a reply to the incoming **Store service** command. The parameter contains information if it was possible to store the service - and if not a message explaining what went wrong.

New service version (ServiceID, version) This is a update notification sent when a service is released with **Release service**. The parameter contains the serviceID and version name of the new version. `serviceID` is sent in with space separated versionparts i.e. `creationID updateID prevousID mergedID`.

Status (Message) Status message sent to the invoker of **Release service** to let the user know if the release was successful or not.

Service(ServiceID, version, jar-file) The response to **Get service** if the requested service was found.

No service The response to **Get service** if the service was not found.

C.3 RemoteBLEAssemblyService

This service store and distribute assemblies. Each assembly is connected to a BLE name, typically the product name. The service is keeping a version number for each combination of hub - bleName.

C.3.1 In-commands

Store service(BLE name, hub type, file) This command is invoked to store an assembly. The BLE name is the name that the BLE peripheral present itself with during advertisement. The hub type is a name that can be used to allow the use of different assemblies depending on which context the hub is used.

Get assembly(BLE name, hub type) When invoking this command the service will reply with the command **Assembly** if an assembly was found for corresponding hub type and BLE name.

Get latest assembly version (BLE name, hub type) Results in a **Latest assembly version** if an assembly was found for the specified combination of BLE name and hub type.

Remove assembly (BLE name, hub type) Remove the assembly associated with the hub name and BLE name.

List assemblies Listing the stored assemblies and their corresponding device- and hub names.

C.3.2 Out-commands

Store response(Message) Response to command store service. Includes a message weather it the assembly was successfully stored or not.

Assembly(Assembly file, BLE name, version number) Reply to the invoker of **getAssembly** if the assembly was found.

No assembly Reply to the invoker of **getAssembly** if the requested assembly was not found.

New assembly version(BLE name, hub type, version number) This is sent to all connected devices after an assembly is stored trough the command **Store service**.

Latest assembly version(BLE name, hub type, version number) The reply to the **Get latest assembly** command. Includes the version number of the assembly corresponding to the BLE name and hub type supplied for the incoming command.

Connection opened Is sent to a newly connected service.

C.4 UpdateManagerService

This is a service running on TheAndroidThing and is responsible for requesting and loading services that need to run locally. It detects if a service is missing when a new assembly is loaded on the device. The following commands are available:

C.4.1 In-commands

Service(ServiceID, version name, jar file) The incoming service is loaded on this device if it was previously requested and is still not loaded on the device.

New version(ServiceID, verson) Notification that there is a new version of a service that is running on this device available. Results in getService(ServiceID) message if the is not already present and there is not a newer version running.

C.4.2 Out-commands

Get service(ServiceID) Requesting all connected devices for a service with the specified serviceID. Is invoked by this service when an assembly with a serviceID marked "self" is loaded on this device, or when it receives a **New version**-command.

C.5 BLEManagerService

This is a service running on TheAndroidThing and is responsible for fetching and update a assembly associated with a detected BLE peripheral. The following commands are available:

C.5.1 In-commands

Assembly(BLE name, assembly, assembly version) The assembly connected to the specified BLE peripheral, from the parameter `BLE name` is loaded on this device if it was previously requested.

New assembly version(BLE name, hub name, version) Notifies the service that there is a new assembly version available. The `hub name` can be set as a parameter on TheAndroidThing (see the user manual for TheThing), If there is no parameter set, the name of the hub is "default". If `hub name` is the name of this hub and is has an older version of the assembly, a **getAssembly(BLE name, hub name)** is sent.

Latest assembly(BLE name, version) Expected as a response after **Get latest assembly version** has been sent. If `hub name` is the name of this hub and it has an older version of the assembly, a **getAssembly(BLE name, hub name)** is sent.

Connection opened Indication that a remote service has connected to this service, which cause this service to ask for the latest versions of its installed assemblies.

C.5.2 Out-commands

Get assembly(BLE name, hub name) Requesting all connected devices for a service with the specified serviceID. The command is invoked when a new BLE peripheral is detected.

Get latest assembly version(BLE name, hub name This command is invoked when a new connection is opened to this service.

Bibliography

[1] Continua alliance. `http://www.continuaalliance.org/products/design-guidelines`. Accessed: March 13, 2015.

[2] itacih, it support for advanced care of cancer patients at home. `http://itacih.cs.lth.se`. Accessed: September 22, 2014.

[3] Java documentation, secure class loading. `http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc5.html`. Accessed: Januari 23, 2015.

[4] Rick Anderson. The end of dll hell. *MSDN Magazine*, 2000.

[5] Ulf Asklund, Lars Bendix, Henrik B Christensen, and Boris Magnusson. The unified extensional versioning model. In *System Configuration Management*, pages 100–122. Springer, 1999.

[6] Robert Bialek and Eric Jul. A framework for evolutionary, dynamically updatable, component-based systems. In *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*, pages 326–331. IEEE, 2004.

[7] Susan Eisenbach, Vladimir Jurisic, and Chris Sadler. Feeling the way through dll hell. In *Proceedings of the First Workshop on Unanticipated Software Evolution (USE 2002), Malaga, Spain*, 2002.

[8] C Elinder et al. *Skattning av njurfunktion*. Statens beredning för medicinsk utvärdering, 2013.

[9] David Svensson Fors. *Assemblies of Pervasive Services*. PhD thesis, Department of Computer Science, Lund University, 2009.

[10] Pervasive Healthcare. Guest editorial introduction to the special section on pervasive healthcare. *IEEE transactions on information technology in biomedicine*, 8(3):229, 2004.

[11] Antonio J Jara, David Fernandez, Pablo Lopez, Miguel A Zamora, Antonio F Gómez-Skarmeta, and Leandro Marin. Evaluation of bluetooth low energy capabilities for tele-mobile monitoring in home-care. *J. UCS*, 19(9):1219–1241, 2013.

[12] Vivekanand Jha, Guillermo Garcia-Garcia, Kunitoshi Iseki, Zuo Li, Saraladevi Naicker, Brett Plattner, Rajiv Saran, Angela Yee-Moon Wang, and Chih-Wei Yang. Chronic kidney disease: global dimension and perspectives. *The Lancet*, 382(9888):260–272, 2013.

[13] Boris Magnusson and Jon Lindholm. Thething users manual. `http://palcom.cs.lth.se/Palcom/Users_Manuals_files/theThing-users-manual-3.1.12.pdf`, 2014. Accessed: June 8, 2015.

[14] Boris Magnusson and Jon Sturk. Implementing services in palcom. `http://palcom.cs.lth.se/Palcom/Download/Poster/2014/5/21_Version_3.1.12.html`, 2013. Accessed: June 8, 2015.

[15] Giuseppe Mancia and Alberto Zanchetti. White-coat hypertension: misnomers, misconceptions and misunderstandings. what should we do next? *Journal of hypertension*, 14(9):1049–1052, 1996.

[16] Mike Ryan. Bluetooth: With low energy comes low security. In *WOOT*, 2013.

[17] Bluetooth SIG. BLUETOOTH SPECIFICATION Version 4.2. https://www.bluetooth.org/en-us/specification/adopted-specifications, 2014. Accessed: May 25, 2015.

[18] David Svensson Fors, Boris Magnusson, Sven Gestegård Robertz, Görel Hedin, and Emma Nilsson-Nyman. Ad-hoc composition of pervasive services in the palcom architecture. In *Proceedings of the 2009 international conference on Pervasive services*, pages 83–92. ACM, 2009.

[19] Upkar Varshney. Pervasive healthcare. *Computer*, 36(12):138–140, 2003.

[20] Mark Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-06-04

EXAMENSARBETE Dynamic installation and automatic update of Bluetooth low energy devices in the Internet of Things
STUDENT Mia Månsson
HANDLEDARE Björn A Johnsson (LTH), Boris Magnusson (LTH)
EXAMINATOR Görel Hedin (LTH)

Smidigare sjukvård på tre sekunder

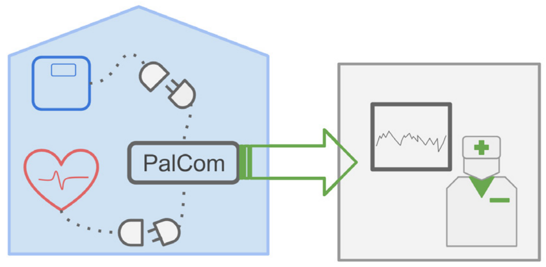
POPULÄRVETENSKAPLIG SAMMANFATTNING **Mia Månsson**

87% av alla dödsfall i EU orsakas av kroniska sjukdomar. För att ge kroniskt sjuka bättre vård gör ”Internet of Things” sitt intåg i sjukvården och flyttar samtidigt hem sjukvården till patientens vardagsrum.

Trådlösa mätinstrument som *“bara fungerar”*
Genom att sätta upp mätinstrument hemma hos patienten hoppas man kunna ge kroniskt sjuka patienter bättre vård. Den traditionella lösningen för att använda sådan utrustning kräver att utrustningen är förinstallerad. Inom sjukvården skapar detta en stor begränsning, eftersom olika typer av utrustning behövs efterhand som patientens behov förändras. Att försinstallera all tänkbar utrustning kan jämföras med att ha en gigantisk ordbok för alla världens språk, fast man bara behöver översätta ord mellan svenska och engelska. Inte heller vill man byta ut hela ordboken bara för ett ord blivit tillagt i något språk.

I mitt examensarbete har jag utvecklat en lösning för att installera trådlösa mätinstrument först när det visar sig att de behövs. Installationen sker automatiskt när ett mätinstrument, till exempel en blodtrycksmätare eller våg, har tagits hem till patienten. Ett sådant installationsförfarande visar sig ta cirka tre sekunder över en 4G- uppkoppling. Dessutom kan uppdateringar av installationen ske automatiskt. Mätinstrumenten kommunicerar med Bluetooth low energy, som är en mer energisnål variant av vanlig Bluetooth.

Kronisk njursvikt
Patienter med kronisk njursvikt är ett exempel på patienter som skulle kunna bli hjälpta av att utföra mätningar hemifrån. Njurarna är kroppens reningsverk och för att inte slita ut dem i förtid är det viktigt att blodtrycket inte är för högt. Men att gå till en läkare en gång i halvåret, eller ens en gång i veckan, för att mäta blodtrycket säger inte mycket om hur blodtrycket faktiskt är i vardagen. Ett annat varningstecken på att njurarna inte fungerar som de ska är plötslig viktuppgång. Därför kan man behöva mäta blodtryck och vikt varje dag i patientens hemmiljö, för att få ett så tillförlitligt resultat som möjligt och upptäcka förändringar tidigare.



Mätinstrument installeras i PalCom för att kunna utföra mätningar hemifrån.

Installera mätinstrument i PalCom
Lösningen för att använda mätinstrument hemifrån är byggd utifrån Internet of Things–lösningen PalCom, som är en programvara som knyter ihop olika apparater för att få dem att kunna kommunicera med varandra. PalCom har sedan tidigare stöd för kommunikation mellan sjukhus och förinstallerade Bluetooth low energy instrument. Inom vården finns ett stort antal olika instrument, många gånger av olika fabrikat och med sitt eget sätt att överföra data. Den “ordbok” som krävs för att installera till exempel en blodtrycksmätare, kan i PalComvärlden översättas till en service och en assembly. Min lösning går ut på att låta PalComenheter själva starta installationen genom att fråga efter en viss service och assembly. Andra PalComenheter får i sin tur agera bibliotek och ansvarar för att dela ut dem. På liknande sätt kan också uppdateringar av services och assemblies skickas vidare till de PalComenheter som är berörda av uppdateringen. Det enda som krävs hemma hos patienten är en androidenhet, t.ex. en surfplatta, där PalCom finns installerat.