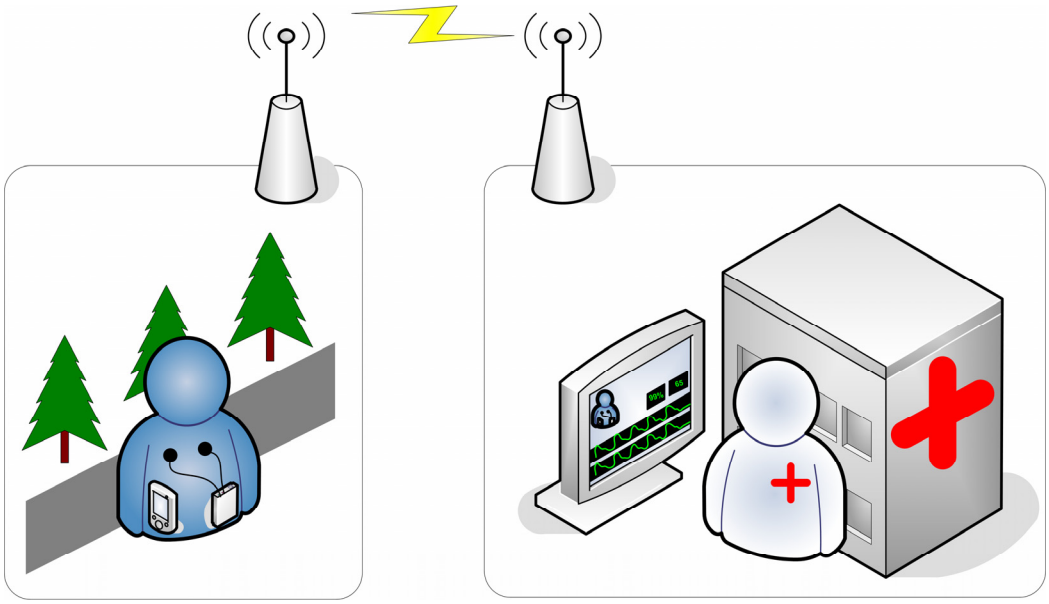
On the design of a mobile e-health platform – towards deployment flexibility

Niels Backx

Thesis for a Master of Science degree in Telematics
from the University of Twente, Enschede, The Netherlands,

15 February 2006



GRADUATION COMMITTEE:
Dr. Ir. A.T. van Halteren (University of Twente)
Dr. Ir. I.A. Widya (University of Twente)
Ir. B. Peet (Yucat B.V.)

On the design of a mobile e-health platform – towards deployment flexibility

Niels Backx

Thesis for a Master of Science degree in Telematics
from the University of Twente, Enschede, The Netherlands,

15 February 2006

UNIVERSITY OF TWENTE,
Faculty of Electrical Engineering, Mathematics and Computer Science,
Department of Computer Science,
Division of Architecture and Services of Network Applications

Abstract

Almost all industries benefit from the information technology era, including the healthcare industry. In the area of telemedicine, high bandwidth communication technologies enable the transmission of a patient's vital signs from one location to the other, for analysis and monitoring. Recent developments in mobile communication technologies allow remote mobile patient monitoring: a patient can be monitored without being bound to a specific location. Several projects have been targeting this area, including the MobiHealth project, which aimed at developing a versatile mobile monitoring platform that was suitable in many remote monitoring scenarios. Such a platform is called a *mobile e-health platform*.

Although MobiHealth aimed at developing a mobile e-health platform, its primary goal was proving the feasibility of such a platform using current mobile communication technologies. The implementation of the MobiHealth prototype is not flexible enough to support a variety of scenarios, without requiring extensive additional design and implementation effort. However, there is a need for such a flexible platform within the Awareness project and possibly future projects.

The mobile e-health platform is a software infrastructure that supports a patient equipped with a mobile device called Mobile Base Unit (MBU) that gathers vital sign data from sensors using short-range wireless technologies (i.e. Bluetooth). The device can communicate using 2.5/3G mobile communication technologies, like GPRS and UMTS, which provide sufficient bandwidth for a set of vital signs, but can be unreliable and are often restricted by the operators. To address these problems and other challenges, the design uses the University of Twente's Mobile Service Platform, which is based on the JINI Surrogate Architecture.

Our design focus is flexibility and to support this, we identified four flexibility aspects for the platform: (1) the sensors that can be connected to the MBU, (2) the lower level communication protocol used for communication, (3) the functionality included in the MBU and (4) the exact service exposed to client applications. This flexibility allows the design to support various usage scenarios, for which the mentioned aspects must be addressed separately.

The major changes in comparison with the MobiHealth implementation are: (1) the way sensor data is internally handled and (2) the introduction of discrete data types that allows transmission of events and data blocks separately from the sensor data stream. We also used (3) a more component-based design approach, which resulted in a clear structure and allows future developers to focus on a specific component and enables separating responsibilities in a project.

The design presents a major improvement compared to the MobiHealth platform and contains sufficient flexibility to support many scenarios in which mobile patient monitoring can play an important role. We partially implemented this design in a prototype, which demonstrates the new features and shows the feasibility of this design. However, the prototype does not include all designed functionality nor is it tested extensively. When fully implemented, the system can be used in research projects like Awareness or can be used as a basis for a commercial product.

Table of Contents

List of Figures

List of Tables

Preface

The area of telemedicine is an interesting melting pot of medical science and telematics. This thesis is related to only a part of the possibilities that the information and communication technology have opened up for the healthcare industry. Besides the applications that are currently in use or being developed, the future may present even more interesting applications that can increase both quality of care and, in the end, quality of life.

After five and a half years of studying for a Master's of Science degree in Telematics, this concluding research has enabled me to bring into practice what I have learnt and also brought me into contact with an interesting area that showed me how my knowledge can be used for a real-world application.

In the past eight months I have been able to conduct this research at Yucat Mobile Business Solutions, to which I want to express my gratitude for enabling me to do this work. I especially thank my supervisor Barry Peet and also Frank Thiele for their valuable feedback and reviews of my work. Furthermore, I would like to thank the employees and interns of Yucat for the discussions we had and pleasant working environment they provided.

I also thank my academic supervisors, Aart van Halteren and Ing Widya, for their suggestions, feedback and our valuable meetings.

In conclusion, I sincerely thank my family for their everlasting support and enthusiasm.

Niels Backx

Driebergen, the Netherlands,
10 February 2006

1 Introduction

This thesis presents our research on the design of a mobile e-health platform. Such a platform can be used for a variety of situations in which remote mobile patient monitoring is useful. The thesis focuses on the technical aspects of the design, but touches upon less technical aspects as well. The research was carried out for Yucat Mobile Business Solutions [YUCAT] to provide them with a versatile platform that can be used in current and future research projects and possibly for commercialization.

This chapter presents the background (Section 1.1) and motivation (Section 1.2) for the research and defines our main objective, which is supported by our research questions (Section 1.3). It also provides the approach for our research (Section 1.4) and the last part describes the thesis structure (Section 1.5).

1.1 Background

The healthcare industry has always benefited from using the latest technologies; mostly for increasing quality of care and to contain costs [ZAJ99]. Especially IT is gradually changing the healthcare industry and helps providing health information services and containing costs [GAN04]. The area where healthcare is supported by the use of telecommunication technologies is called *telemedicine*, which started with simple remote consultation of a medical specialist by phone or videoconference, or by sending patient medical data to a remote location for assessment. Nowadays, it is also possible to transmit live patient data from one hospital to another hospital where a specialist is available, using high-bandwidth telecommunication lines, or less distant, having ubiquitous access to patient data throughout a hospital [NEL97]. Using broadband Internet to the home, even real-time home monitoring of patients is possible.

Telemedicine has three driving forces [ZAJ99]: the first is *access to care*, telemedicine enables assessment of patients in rural areas who otherwise would not have access to this specialist care; the second force is *cost containment*, health care expenditures are increasing yearly and technological development often increase the cost; telemedicine can reduce cost by earlier discharge and reduced travel expenses. The third driving force is *quality of care*, for example, efficient information presentation and remote consultation help general practitioners in making a better and quicker assessment of a patient's illness. Intelligent systems that accumulate and process patient data from a variety of sources also play an important role for this assessment [BAR99].

Three factors limit the usage of telemedicine applications; first, healthcare professionals are often not aware of the possibilities of new technologies (although patients and healthcare professionals have positive experiences with telemedicine and do not oppose using telemedicine) [LIN99]. Second, most telemedicine applications are technology-driven, which means the applications originate from the new technologies, not from the healthcare

domain request, thus its usefulness must be proven and third, insurance companies often require that applications must be proven to be clinically effective before compensated for [ZAJ99].

Mobile data communication technologies like GPRS and more recently UMTS are being deployed throughout the world. Moreover, personal computing devices are decreasing in size and provide portable computing capabilities. These technologies offer the same bandwidth capabilities as early home-monitoring applications used; so combining home-monitoring with these technologies open the latest area in telemedicine: *mobile patient monitoring*. With mobile patient monitoring, a patient has portable monitoring and communication equipment and his vital signs can be sent to his healthcare professional, reducing the time a patient spends in the hospital [KON02].

Current mobile monitoring systems focus on periodic sending of patient data, or use older technologies with lower bandwidth [KYR03]; often, these systems also focus on a single type of sensor. The European MobiHealth project [MOBIHEALTH], which ended in 2004, aimed at developing a generic platform for continuous mobile patient monitoring that can be used in areas like monitoring during sports training, home monitoring and clinical research [KON02]. This platform links several sensors into a Body Area Network and transmits the sensor data over a single wireless link to the healthcare professional. We call the generic platform as aimed for in the MobiHealth project a *mobile e-health platform*.

1.2 Motivation

The MobiHealth project trialed the applicability, user-acceptance and feasibility of a mobile e-health platform and its results indicated that there is a need for such a platform and that a stable product is very useful [MH/D5.1]. A generic platform should be flexible at several aspects, like the set of sensors connected to the patient, the functionality provided by the platform or the user-interface of the application the healthcare professional uses to see the vital signs. This enables the platform to be used for a variety of new services in different scenarios. These aspects are defined as implementation flexibility.

Although the mobile data communication technologies are becoming more widespread and although their bandwidth increases; they still suffer from fluctuating speeds and availability due to changing coverage because the user is mobile. A typical set of vital signs data can often be sent using a standard UMTS connection or a good-coverage GPRS connection; however, if the connection degrades, some of the vital signs must be dropped to maintain the real-time property of the data [WID03]. For more complex sets of vital sign data, like a 12-lead ECG, it is often required that as much data as possible is sent, to allow correct assessment of a patient. Both situations demand certain flexibility in the set of vital signs being transmitted at run-time. This flexibility is defined as run-time flexibility.

The prototype developed for the MobiHealth trials does not contain enough flexibility to be used effectively used in the various areas. Moreover the design and implementation of this prototype is badly documented and complex; restricting the possibilities to adapt the prototype for a specific scenario. The prototype and results of MobiHealth are used in two other projects: the Freeband Awareness [AWARENESS] project and HealthService24 [HS24]. Within the Awareness project, the mobile e-health platform plays an important role in demonstrating the usage of the service and network infrastructure for context aware mobile applications. These

context-aware applications use context information (like location or presence) to provide services or execute complex tasks; more information on context-awareness can be found in [DOC04].

To support integrating the Awareness functionality in the mobile e-health platform and to work towards a platform that is usable for future projects and possibly commercial deployment, it is necessary to revise the MobiHealth platform. In this research, we will revise the MobiHealth platform and focus on the flexibility requirements, such that it can be used in the Awareness project as well as for future projects.

1.3 Objectives and scope

This section describes the main objective for our research and determines the scope. The main objective is split into three research questions that must be answered to accomplish our objective. Furthermore, we define some challenges that are encountered in the design of a complex distributed platform. Using our research questions and design challenges, we identify some evaluation criteria to evaluate our end-result.

1.3.1 Main objective

Based on our research motivation we define our main objective as follows:

> "*Design a mobile e-health platform that can be implemented and deployed using available technologies and contains sufficient run-time and implementation flexibility to support a variety of scenarios in which remote patient monitoring can increase quality of care, using the design of the MobiHealth system as a reference.*"

The result of this research is a structural design of 'our' mobile e-health platform that defines important components and scenario independent aspects of such a platform. We have included '*using available technologies*' to ensure it can be implemented in the short term. The design presented in this thesis contains the basic functionality for the mobile e-health platform and when implemented, it will present a very light-weight version of the platform. Additional functionality that is required for more complex scenarios is left for future work.

Due to time constraints, we will focus on defining the main components and their behaviour, but will not elaborate on the component-implementation. However, we will verify and test our findings by implementing a prototype. This prototype does not contain extensive error-handling nor is it extensively tested with regard to performance, scalability and stability. The design of the healthcare professional application and the patient's user-interface are also ignored for this research, since this requires extensive interaction with the users and is very scenario dependent.

1.3.2 Research questions

To reach our objective, we aim to answer the following research questions; the first two are related to the design of the platform and the latter is used to verify this design. For each question, we have included the main tasks that are necessary to give an answer to these questions.

(1) What are the requirements for the mobile e-health platform that supports the main functionality that is required in a variety of scenarios?

a. List scenarios that cover different aspects of remote patient monitoring and that are representative for a variety of remote patient monitoring situations.
b. Perform a requirements analysis for the mobile e-health platform, using scenario definitions and technical restrictions.

(2) *To what extent can the current design and implementation of the existing MobiHealth system be used to work towards a design and implementation of a mobile e-health platform that supports the required flexibility?*
a. Analyse the MobiHealth design and implementation.
b. Design the mobile e-health platform, using available technologies.
c. Implement a prototype of the design.

(3) *Is the design suitable for the scenarios as defined in (1) and what is necessary to implement a specific scenario?*
a. Verify the design using the requirements from (1b) and use the prototype to demonstrate the capabilities.
b. List the tasks that are left for scenario developers / system integrators.

1.3.3 Challenges

The mobile e-health platform is a complex distributed application and designing such an application involves design challenges, some of which have already been addressed within the MobiHealth project. This section identifies the main challenges and lists some problems that add extra challenges to our research.

Challenge 1: *Gain sufficient domain knowledge to be able to understand the requirements of the end-user.*
We have limited domain knowledge on the subject of healthcare and telemedicine; this is a potential problem, because it is important to understand what the end-user requires and to identify priorities in the platform. Although it is impossible to become an expert on telemedicine within the limited timeframe that we work in, we can gain some domain knowledge by studying literature and talking to experts. The deliverables for the three related projects HS24, Awareness and MobiHealth provide a good source for this domain knowledge, especially the usage scenarios and trial descriptions give us insights in what the system is used for. Several medical parties, like medical centres and research institutes, were involved in these projects.

Challenge 2: *Deduce sufficient implicit design knowledge that is embedded in the MobiHealth implementation, such that it can be used for our research.*
As said, the MobiHealth project forms the base for our design effort; because we did not personally contribute to this project, we have to study the results of the MobiHealth project. Due to the lack of design and implementation documentation, we have to study the implementation thoroughly to deduce the design and the implicit knowledge that is included in this design.

Challenge 3: *Work towards a run-time and implementation flexible platform that can be used for a variety of scenarios, while minimizing design complexity.*
The MobiHealth implementation lacks some features that are required for a mobile e-health platform that can be used for a variety of scenarios; therefore we must work towards a flexible platform that allows developers to assemble an implementation for a specific scenario. Flexibility can add up to the complexity of the application,

therefore we have to determine what aspects of the platform must be flexible, both on run-time as implementation flexibility.

Challenge 4: *Minimize resource usage at the PDA to allow additional features*
To keep the system portable and mobile, we will use a Personal Digital Assistant (PDA) as the main processing unit. PDA's have limited resources (although their capabilities increase rapidly). If our base platform would consume too much processing and storage resources of the PDA, future extensions such as digital signal processing are inhibited. Also battery consumption plays an important role for a mobile device; if the battery needs to be charged too often, it will decrease the mobility of the patient. Battery consumption depends on both wireless communication and processor usage.

The evaluation criteria in the next section help us verifying if we have overcome these challenges.

1.3.4 Evaluation criteria

This section defines evaluation criteria that are derived from the main objective, research questions and challenges. The criteria are used to evaluate the design of the mobile e-health platform and help answering the research questions and verifying if we have accomplished our main objective.

Flexibility
Flexibility is one of the main focus points in the design; the following criteria are used to evaluate the flexibility of the design:

- The designed platform must support the scenarios and trials that are defined for the MobiHealth and Awareness projects, since the HS24 trials are very similar to the MobiHealth trials, we will not include these for our evaluation and assume that they are supported when the other scenarios are;
- The design must support run-time flexibility with regard to enabling and disabling sensors on-the-fly;
- The design must include implementation flexibility, such that it is possible to use different sensors for each scenario and use scenario specific healthcare professional applications;
- When the design is implemented, it must support a very simplistic scenario, which can be seen as a light-weight version of the platform.

Resource usage
Since the system must be able to run using available technologies and based on the fourth challenge that resource usage should be minimized; we define the following evaluation criteria with regard to resource usage.

- The prototype implementation must be able to run at the device that is currently used to run the MobiHealth system;
- CPU usage of a light-weight platform in regular operation (comparable to Awareness trial, without signal processing) must be below 60% at the device currently used in the projects, to leave room for other applications and functionality, like signal processing.

1.4 Approach

This research follows a standard approach, consisting of an analysis, design, implementation and evaluation phase. The steps we have identified within these phases are listed below.

1.4.1 Analysis phase

The first step is analysing the work on the MobiHealth platform, because this is the origin of the current application. Awareness deliverables and HealthService24 deliverables also form an important input for our analysis. After this, we will thoroughly study the implementation of the system and try to derive the design from this implementation. The results of this analysis are both domain knowledge and current system knowledge and the results are presented in Chapter 2.

The second step is defining the leading scenarios and some generic scenarios, which we can be used as a reference throughout the research. We use a simplification of the leading Awareness scenarios and trial documentation for the other projects, the descriptions can be found in the first part of Chapter 3.

The next step is to identify the requirements for the platform; we have explicitly chosen to do this after current system analysis, so we can use our domain knowledge. For the requirements phase we will interview experts, talk to developers of the current system and study requirement-related deliverables. We use use-cases to identify the actors, systems and main system behaviour. The results are described in the second part of Chapter 3.

1.4.2 Design phase

For the design phase, we will start by identifying the high-level structure; defining the distributional aspects of the application and use a top-down approach to define the sub-components. We will focus on designing the part that will be deployed on the mobile device; however, we will also look into backend design.

We will then try to implement (parts of) the components to evaluate their design. This enables us to improve the design and can be seen as an iterative process.

The top-level results for the design phase are described in Chapter 4 and we will elaborate on the lower level components in Chapter 5 and Chapter 6.

1.4.3 Implementation phase

The implementation phase consists of combining and improving the prototype components from the design process and developing new components from scratch, working towards a prototype platform implementation. The next step is to test this implementation with regard to the evaluation criteria as defined in this chapter. We will describe the most important parts of the implementation and testing in Chapter 7.

1.4.4 Evaluation phase

In the evaluation phase, the prototype will be tested with regard to resource usage and functionality compared to the MobiHealth implementation. The platform design and the results of these tests will be reflected on our evaluation criteria; furthermore, we will try to answer our research question and discuss how we have overcome our challenges. The last part of this research consists of evaluating our main objective and discussing the future work on the platform. The results of the evaluation are presented in Chapter 8.

1.5 Structure

This thesis is structured as follows:

- Chapter 1 gives an introduction to our research and poses our main research objective, our research questions and challenges;
- Chapter 2 discusses the current implementation and the technologies used in this implementation;
- Chapter 3 contains the requirements analysis on which the design is based;
- Chapter 4 introduces our high-level design and the core components in the system, using a top-down approach;
- Chapter 5 elaborates on the design of the mobile base unit, the mobile part of the application;
- Chapter 6 elaborates on the design of the surrogate and discusses some parts of the backend design;
- Chapter 7 shows the main challenges in our prototype implementation;
- Chapter 8 concludes this research with an evaluation, conclusion and discusses future work on this subject.

2 Existing design and implementation

This chapter presents the results of our analysis of the current design and implementation of the mobile e-health platform. The foundation for the implementation that is currently used in the Awareness project [AWARENESS] was built in the MobiHealth project [MOBIHEALTH]; the HealthService24 project [HS24] also uses the results of the MobiHealth project (and source code). Awareness and HS24 have different goals, but share source code. The design we examined for this chapter has been used for the first beta release of the HS24 and was the core for the Awareness project at that time. For the remainder of this thesis, we will refer to this design as the MobiHealth design, or simply MobiHealth.

We will start our analysis by showing the high-level design, where the main entities in the system are described. Then we discuss the different connections in the system, including the technologies used. The third part describes the most important parts of the implementation of the system. Then we give an overview of how the MBU application evolved and we end with a conclusion on the current design and implementation.

2.1 High level design

Figure 2-1 shows the overall system design for MobiHealth; its main components are the Mobile Base Unit (MBU), the backend network and the healthcare application. The MBU is the device that the patient carries, the healthcare application is an application that a healthcare professional uses to view the data sent by the MBU and the backend network is the backbone to which the MBU and the healthcare application connect. The backend network is usually protected by a firewall, to secure the patient information. This firewall must allow the MBU to connect to the surrogate host and must allow the healthcare application to connect to the backend service.

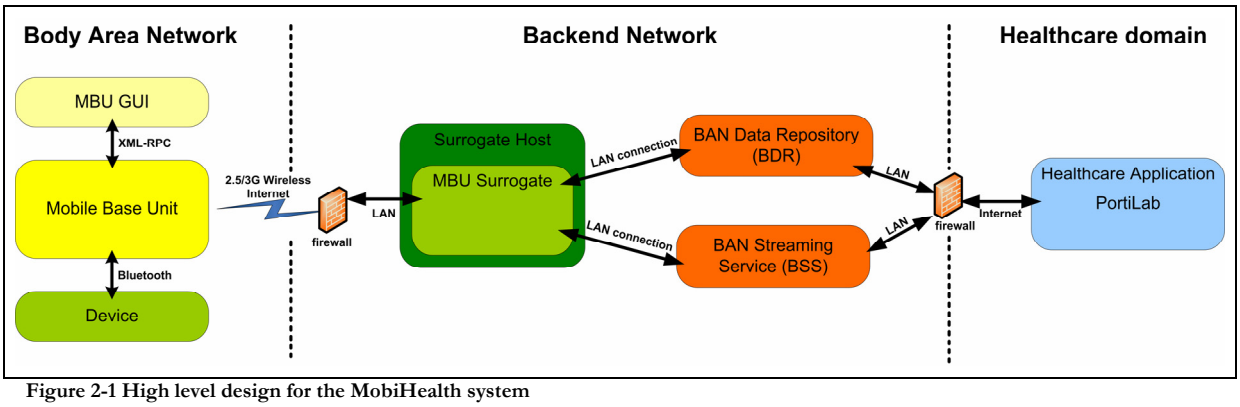


Figure 2-1 High level design for the MobiHealth system

The initial system design aimed for flexibility for multiple devices and different types of back-end functionality, but since the TMSi Mobi-device and the associated TMSi PortiLab application were already available [TMSI], the actual implementation mainly focuses on collaborating with these components.
Figure 2-2 shows the communication protocols used between the components of the MobiHealth system. The MSP interconnect and RMI push protocols are the most important ones, since these are defined within the system and can be changed if necessary, whereas the Bluetooth connection between the (Mobi) sensor box and the MBU uses the TMSi fibre-protocol [BRO03]. RMI is Remote Method Invocation, which is a Java standard for remote procedure calling, the RMI push mechanism indicates that the surrogate calls a method at the backend service. If it would be the other way around, it would be called RMI pull. The backend protocol is arbitrary and depends on the backend service and healthcare application. Currently, this backend protocol is a simple raw TCP stream of sensor data.

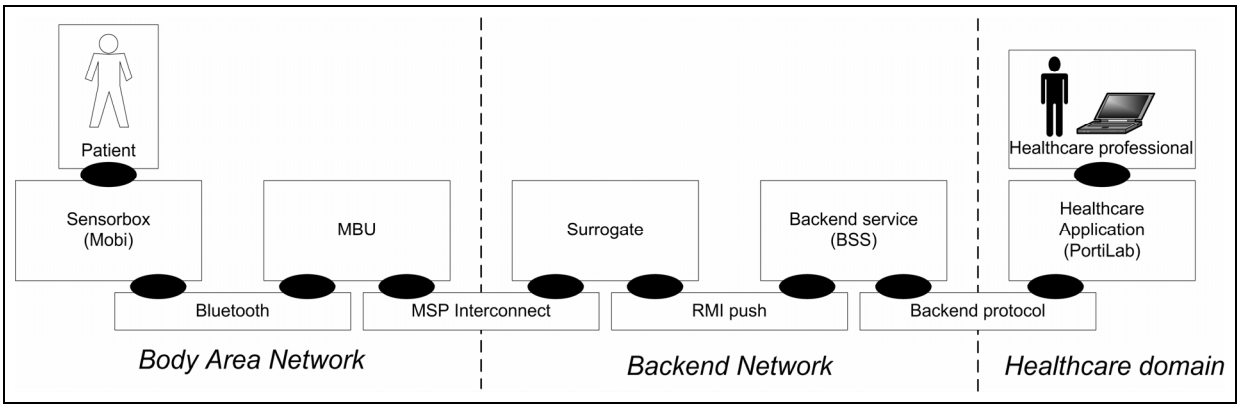


Figure 2-2 Connectivity overview for the MobiHealth system

The patient side of the system is often called the *Body Area Network*; the services that are always online are located in the *Backend Network* and the healthcare application usually runs in the domain of a hospital or general practitioner (GP), which is called *Healthcare Domain*.

2.1.1 Mobile Base Unit

The *Mobile Base Unit* or MBU is the PDA or other mobile device that runs at the patient-side of the platform. It is connected to the devices that provide sensor data on one side, and it is connected to the Internet over a 2.5/3G network on the other side. It is also possible to use other wireless technologies, like WiFi.

The devices that provide sensor data can be of different types, actuators are not supported. For patient data, only the Mobi devices by TMSi are being used extensively. Yucat has implemented GPS functionality that shows the location of the MBU in GPS coordinates, but this is currently not embedded in the system. Devices can be connected to the MBU by any communication means available on the MBU; however, each device requires a connectivity module. The Mobi only supports Bluetooth communication, which requires platform dependent (non-Java) communication with the operating system. Yucat has implemented a library called the BTStreamer that facilitates this communication for a Windows Mobile device.

Data measured from the devices is propagated into the MBU JavaCore, the JavaCore can do signal processing, compression et cetera and prepares the data to be sent to the surrogate using the Interconnect protocol as described below.

For usability, the MBU also contains a Graphical User Interface (GUI). The GUI is decoupled from the JavaCore to allow different implementations of the GUI for different scenarios. It also allows the GUI to be implemented in a different programming language than Java. The communication between the JavaCore and the GUI is implemented using XML-RPC, a standardized simple remote procedure call technology that uses XML to encode the requests [XMLRPC].

2.1.2 JINI Architecture and Nomadic Mobile Services

In a nutshell, *JINI* is a service discovery architecture designed by SUN, which enables developers to create adaptive networks and services; it is a specific implementation of the Service Oriented Architecture (SOA). In JINI, a service registers itself on start-up at the JINI Lookup Service (LUS), so it can be discovered. A client locates the service in this LUS and downloads a proxy to communicate with that service [JINIWEB]. Section 6.2.1 elaborates on JINI services in a mobile e-health platform.

Nomadic mobile services are services that are offered by a mobile device; they are called nomadic because regular mobile services are often associated with services that are offered to mobile devices. A nomadic mobile service may constantly change location and as a result may have a changing network address and frequent loss of connectivity. Mobile devices also have limited or expensive bandwidth, so data exchange should be minimized. The JINI architecture defines some strict criteria (e.g. fixed IP address, no network address translating (NAT)) for a service to join the network; a nomadic mobile service does not meet all these criteria and can therefore not join a JINI network on itself. Therefore SUN has introduced the Surrogate Architecture as an addition to JINI [SUN03].

The surrogate architecture defines an architecture that allow services to join a JINI network, which otherwise could not join. Within the architecture, a so-called surrogate joins the JINI network on behalf of the real service. This surrogate uses the facilities of the surrogate host, which provides an environment for executing the surrogate from a host-capable machine. The environment offered meets all criteria that are required for joining a JINI network. The surrogate communicates with the (nomadic) service via an interconnect protocol and is loaded in the surrogate host when the service is running. Service users locate the surrogate of the service in the JINI network and communicate with the service via the surrogate.

In MobiHealth, the MBU and the related MBU surrogate provide the nomadic mobile service. The HTTP interconnect protocol provides the communication between the MBU and the surrogate (explained in section 2.2.2). More information on the JINI architecture and Nomadic Mobile Services can be found in [TOL05] and [ESU05].

JINI was chosen to solve some technical challenges for MobiHealth platform and which are applicable for a mobile e-health platform in general, these challenges are listed below [DOK03]:

- MBUs are ambulatory and can come on-line and go off-line at any point in time; a list of on-line MBUs must be available at the backend. JINI solves this problem via the lookup server that provides discovery methods.
- The MBU has limited resources and can therefore not join a JINI network; the surrogate architecture allows the MBU to focus on its core functionality and to delegate service offering to its surrogate.

- The MBU can not offer a reliable connection to backend applications; using the surrogate architecture, this problem only needs to be solved once;
- The mobile e-health platform must be scalable and should support potentially thousands of MBUs; JINI networks are designed to be scalable.

2.1.3 Backend services

Backend services are used as a bridge between the healthcare applications and the surrogate. Currently, the two most used servers are: the BAN Streaming Service (BSS) and the BAN Data Repository (BDR). The first enables a user to view the sensor data real-time in PortiLab by putting the raw data on a socket. The second stores raw data for every session in which the MBU sends data, so it can be reviewed later (offline). The services are implemented as JINI clients, meaning they can use the LUS to locate an MBU (represented via the surrogate) and subscribe for its data.

The BDR waits for new surrogates to join the LUS and immediately subscribes for the sensor data, it writes the incoming data to a file. This file is published when the MBU session finishes, a healthcare professional can select the MBU and the session from within PortiLab (see 2.1.4), and simply downloads the session file from a web server. PortiLab then displays the sensor data as a streaming recording of the sensor data.

The BSS waits for a healthcare professional to login to PortiLab and then requests a list of currently available MBUs in the LUS. The healthcare professional selects an MBU and the BSS subscribes for sensor data at the surrogate for the selected MBU. The BSS opens a socket and PortiLab connects to this socket and receives the sensor data.

2.1.4 Healthcare professional application

The healthcare professional application shows the sensor data for the healthcare professional. Currently, the only application used is TMSi PortiLab, which is a closed-source application. This application has extensive filtering capabilities and is fully compatible with data that the TMSi Mobi devices generate. Unfortunately, it is designed for data from one single device at a certain frequency; a plug-in module facilitates communication to the backend services and is not very flexible. For example, it is not possible to add or remove sensors during a session.

Some research has been done on other types of healthcare professional applications, for example using web services [PRU04], but none of these are currently used in trials. An extensive portal is currently being developed in the scope of the Awareness project.

2.1.5 Standard operation

Figure 2-3 shows the general operation for the system. All entities in the backend network are already present and initialized (the Surrogate Host, the Lookup Service and the Backend Service). When the MBU is initialized, it registers itself at the surrogate host. The surrogate host instantiates a surrogate object for the MBU and this surrogate registers itself at the JINI Lookup Service. When this is completed, the MBU will start sending sensor data to the surrogate (it is possible to delay this sending until a healthcare professional requests data).

At some moment in time, a healthcare professional wants to see how the patient is doing, so he starts his healthcare application. This application contacts the backend service (in this case the BSS, since he wants to see real-time data) and specifies what patient he wants to see. The backend service contacts the lookup service, asking for the surrogate that is linked to the MBU of the patient. The lookup service returns a proxy for this surrogate and the backend service can now subscribe for the sensor data. The surrogate receives this subscription and relays the sensor data that it receives from that moment to the backend service. The backend service in its turn relays the sensor data to the healthcare application, where the doctor can see the data.

After a while, the doctor shuts down his application, which results in an unsubscribe operation to the surrogate. The surrogate then stops relaying data to the backend service. The MBU will continue to send data to the surrogate.

When the patient wants to stop sending data, he presses the stop button on his MBU and the data transmission stops. The MBU sends a stop-message to the surrogate, which will stop running and will be destroyed by the surrogate host. The MBU is then also shut down, while the servers in the backend network keep running.

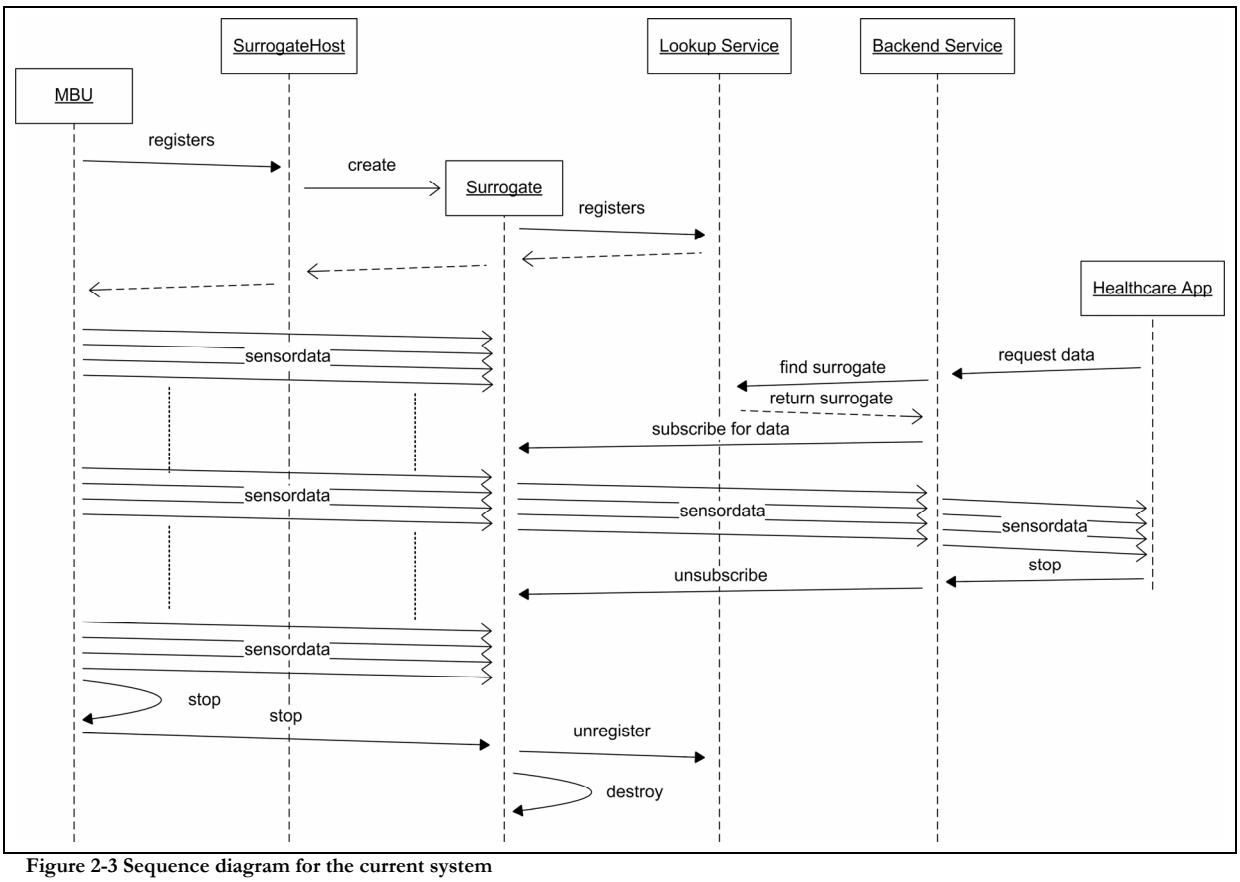


Figure 2-3 Sequence diagram for the current system

2.2 Connectivity

The main purpose of the system is to transfer sensor data from the patient to the healthcare professional, therefore connectivity is one of the most important tasks of the platform. Figure 2-2 showed a connectivity overview, this section will elaborate on the specific connections.

2.2.1 Sensor device to MBU

The system is flexible with regard to sensor devices, for each device-type a so called Device Implementation must be written that handles device communication. The Mobi sensor devices communicate to the MBU using Bluetooth, over which it transmits sensor data using the TMSi fibre protocol. A special MBU component called the BTStreamer implements the Bluetooth communication and the fibre protocol and offers the sensor data to the MBU in a stream of fixed-size data packets.

2.2.2 MBU to Surrogate

The communication between the MBU and surrogate is the most important communication part, since it uses unreliable networks (GPRS/UMTS) and has limited bandwidth available. This communication is clearly the bottleneck of the system. The University of Twente has designed a special protocol to facilitate this interconnection between the nomadic service and the surrogate host, which is called the HTTP-Interconnect protocol or MSP Interconnect protocol. This protocol is designed to connect mobile devices to JINI networks; it uses HTTP as an underlying architecture, because many telecom operators only allow HTTP traffic through their networks. More information on this Mobile Service Platform can be found in [ESU05] and [TOL05].

The interconnect protocol offers means to send a request/response and one-way messages from the mobile device to the surrogate and vice versa. It also supports a data stream from the mobile device to the surrogate. MobiHealth uses the messages for connection setup and it uses the data stream for actual sensor data.

The Mobile Service Platform provides some mechanisms to prevent connection loss; however, if the MSP cannot recover from connection loss, the session is stopped and not automatically restarted.

2.2.3 Surrogate to backend

The surrogate to backend communication incorporates usage of the JINI architecture. When a surrogate is initialized, it will export an object as a JINI Service. This object will be serialized and put into the LUS; where it will stay until renewed or removed. The object is a copy of the object that the surrogate had and contains an RMI reference to the actual surrogate.

A backend service locates the MBU in the LUS and retrieves the object that was placed in the LUS. It can now communicate with the surrogate using RMI by calling methods on the surrogate. The system only supports methods for connection set-up and shutdown. At connection set-up, the backend service sends an RMI reference of itself to the surrogate; this enables the surrogate to call methods at the backend service. This is being used for sending actual sensor data packets from the surrogate to the backend service.

The system is designed to be flexible by supporting multiple types of connection between the surrogate and the backend; however, the only method implemented is RMI.

2.2.4 Backend to healthcare application

Although the JINI architecture supports that a healthcare application connects directly to the surrogate, this is not used by PortiLab. As explained in section 2.1.3, PortiLab connects to the BSS using a socket for real-time monitoring and uses a file download for historic sensor data.

2.3 Implementation structure

We have thoroughly analysed the implementation of the system; in this implementation we distinguished managing components that dealt with initialization, connection handling et cetera and the components that deal with sending data. Although both types are important for system operation; the data sending components are most important for our research, therefore we will only elaborate on these components. The implementation of the remainder of the system is explained in [YUC05].

Figure 2-4 shows the data sending components and the data flow, based on the deployment configuration. These are discussed in detail in the following sections. One of the most important aspects of the system is the push mechanism that is used for all components: The device implementation reads data from the device and pushes it into the system. From there on, each component processes the data and then pushes it onto the next component. This leads to a chain of components that are all connected, from device until backend.

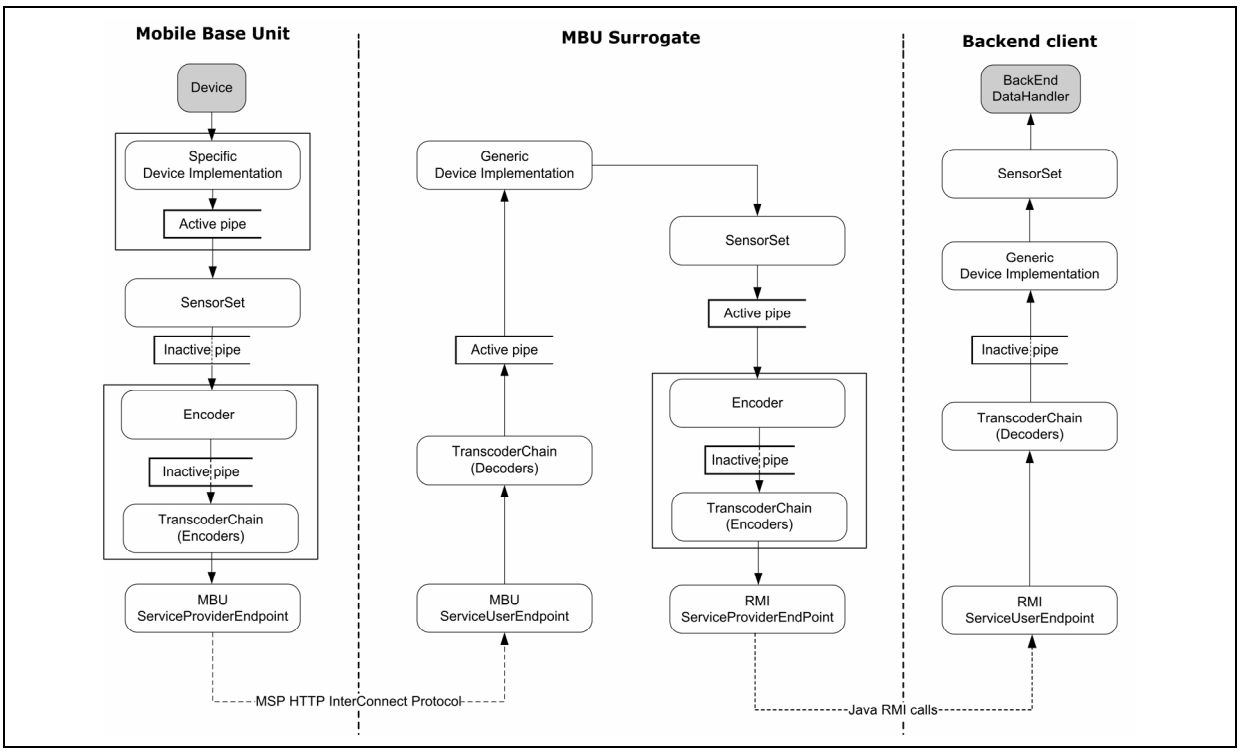


Figure 2-4 MobiHealth implementation structure

2.3.1 Sensor Set

The SensorSet is a digital representation of all the sensors connected to a single front-end device (like the Mobi). Subscribers to a SensorSet will receive the digitized sensor values as an array of bytes. This array can, for example, be used to plot a graph or to perform digital signal analysis. The SensorSet also contains a list of the

sensors to which can be subscribed individually, so that a subscriber only receives the data for that specific sensor. Unfortunately, this functionality is currently not working correctly.

The SensorSet is managed by a SensorSetManager and receives its data from a Device Implementation (see 2.3.3); it then forwards its data to all subscribers. These subscribers can subscribe at two points: the SensorSet itself or at its manager. The main difference is that the SensorSet sends data directly to the listeners, while the manager uses a pipe to buffer the information. The exact reason to have two points to subscribe to is unknown, but probably subscribing to the manager is for listeners that require a lot of processing time or have delays because of communication lines. If the data is processed instantly, the extra pipe does not seem necessary.

2.3.2 Transcoding chain

The transcoding chain is a chain of encoders that can alter the representation of data in the system; this is mostly used for saving bandwidth and CPU power. The transcoding chain is linked to the service provider endpoint that sends data to a peer. All encoders must have a corresponding decoder at the peer, if no decoding is required, the data can be passed through unaltered by the decoder.

The most important encoders currently implemented are:

- **Filter:** Remove the data of sensors that are not active according to the configuration, this reduces the amount of data sent over the line (e.g. disable a Plethysmogram for a cardio trial);
- **Packet:** Buffer a certain amount of data samples and forward them in one large packet, reducing the amount of packets sent over the line. The optimal buffer size depends on the communication network used [BUL05], [WAC05];
- **Deflate/Zip:** Two different implementations that compress the data by using a standardized method, this also reduces the amount of data sent over the line.

As shown in Figure 2-4, the transcoding chain is used twice: once for communication between MBU and surrogate and once for communication between surrogate and backend. If multiple backends are connected they will share the transcoding chain if the same transcoding elements is requested; however if they request a different chain, the surrogate will create a new transcoding chain. The surrogate can thus have multiple transcoding chains at the same time.

2.3.3 Device Implementation

At the MBU side, the device implementation is responsible for reading data from a device and relaying this to its SensorSet. This implementation is specific for the device, so the TMSi sensor box has an own implementation and additional devices, like a GPS receiver, will need their own implementation as well. At the MBU side, the purpose of this device implementation is clear, but it also exists at the surrogate and the backend; these locations use a generic implementation. This generic implementation receives the data not from a device, but from a pipe that receives data from the decoder-chain. It is unknown why a device implementation is necessary at the surrogate and back-end, since it is not used by any backend server. Most probably it exists to simulate the MBU as accurate as possible at those remote locations.

The device implementation has knowledge of the sensors that are attached to it, but this is limited to descriptive information only. The data is immediately forwarded to the SensorSet, where other components can subscribe.

2.3.4 Service endpoints

Service endpoints are the entities that provide connectivity from one location to the other and vice versa. Depending on configuration, these endpoints use a specific protocol to send data; in the current setup, the system uses the MSP Interconnect protocol between MBU and surrogate and uses Java RMI push calls between surrogate and backend clients. Other implementations are currently not in use.

Two types of endpoints can be identified:

- **ServiceProviderEndpoint:** This is connected to the encoding transcoding chain, so it receives data from the system and sends it over the network using the configured protocol.
- **ServiceUserEndpoint:** The side that receives data from the ServiceProviderEndpoint and forwards this up into the decoding transcoding chain, so it can be put into the sensorset.

2.3.5 Pipes

At several locations in the system pipes are used to buffer data; data is put in on one side and sent to all pipe-listeners on the other side. These pipes can be configured to be active or inactive.

- **Active:** These pipes have their own 'pump'-thread, this causes the source of the data not to block for a while if data is being sent to multiple receivers that possibly do some intensive computation or use blocking socket calls.
- **Inactive:** These pipes just relay the data to the listeners, running in the same thread as the source of the data.

It is important to have active pipes at strategic locations, such that important operations (like reading new data from the device) do not block or delay too much. However, too many active pipes will cause more threads running in the system; this causes extra overhead. The current settings for active/inactive pipes are based on experiments.

2.4 History and current work

This section describes the history of the MobiHealth application and the (con)current work on the application, which gives an insight into how the application evolved and what major changes can be expected.

2.4.1 History

The MobiHealth project started mid-2002 and resulted in a Mobile Base Unit (MBU) application, implemented in Java, which ran on an HP iPAQ PDA with Linux. This configuration was used for the various trials in the MobiHealth project and proved that it was possible to use 2.5/3G technologies for wireless sensor data communication and that a stable commercial product would be very useful [MH/D5.1]. However, the application was very complex in usage and appeared to be unstable. One of the conclusions in the trial evaluations was that a re-implementation of the system (to improve the software) was required to solve some of the problems experienced during the trial.

Yucat has ported a simplified version of the complete PDA application to Windows Mobile using C# .NET to prove that a Windows Mobile PDA was also suitable for running such an application. This feasibility study led to a port of MobiHealth to the Windows Mobile platform, in which the Java core remained intact, while the GUI was replaced with a C# .NET GUI, this is described in [THI05].

The last major change to the application was removing the BAN interconnect protocol into a separate project: the M-health Service Platform (MSP), which evolved into the Mobile Service Platform. The Windows Mobile application that uses the MSP is used for this research and is referred to as the MobiHealth application or the current application. It is also the application that is currently being used in the HS24 and Awareness projects.

Figure 2-5 MobiHealth trial equipment

Figure 2-5 shows the patient equipment for the MobiHealth trials. The device on the left is the HP iPAQ, the box on the upper-right is a TMSi Mobi4 device, which is connected to five ECG leads and an alarm button. Nowadays, the alarm button is on top of the Mobi device and the ECG leads are grouped to reduce the amount of connectors.

2.4.2 Current work

At the moment of writing, several projects are working with the MobiHealth application; the HS24 project aims to stabilize the system and adding some required functionality. The focus of this project is on the PDA application.

The application is also improved within the Awareness project; however, it focuses more on backend development (an integrated portal) and using the application with new technologies, like positioning and context-aware services.

Yucat is currently improving the Bluetooth capabilities for Java on the Windows Mobile platform by developing a new Bluetooth Framework.

2.5 Conclusions

Studying the system has revealed the complexity of the current implementation. This complexity mainly consists of the large amount of coupling of the code elements. The general design idea was to have code that was flexible with regard to different communication protocols and the same code could be reused at all three locations. This design idea also adds to the complexity; furthermore, some location specific (e.g. MBU) parts of the code are put in a general class that is available throughout the system (at MBU, surrogate and backend).

The current platform lacks documentation and code-comments; as a result much design knowledge has been lost, which makes it difficult to understand the system. This inevitably led to making assumptions about the design rationales at some points of this analysis.

The surrogate architecture with the MSP-interconnect protocol seems to be ideal for the mobile health application, since it supports public mobile networks and solves the standard problems of a nomadic mobile service in a JINI network; however the current implementation does not use all functionality offered by the MSP. The surrogate to back-end communication is based on Java-RMI, and while the connection setup is complex, the actual communication is quite straightforward.

All in all, the complexity causes that locating bugs and solving these bugs is a tedious task. Moreover, making small changes or additions is a time consuming job, since it is hard to predict what classes are affected by these changes. Nevertheless, several patterns and implementation structures are very usable for a stable yet flexible mobile e-Health platform.

For a platform that can be used for various scenarios and that offers a good framework for further development, we do not deem the current implementation sufficient. Therefore we must decide if we want to redesign from scratch, using the design and implementation knowledge gathered from this analysis; or if we want to change the current implementation step-by-step removing the coupling and working towards a platform that satisfies our needs. This decision will be taken based on our requirements analysis for the platform, which is presented in the next chapter.

3 Requirements for a mobile e-Health platform

This chapter gives an overview of the usage of, and requirements for the mobile e-health platform. These requirements form the base for the design of the mobile e-health platform and can be determined by identifying the needs of the end-users and the future developers of the system. We can distinguish two types of requirements: the *end-user* requirements and the *technical* requirements. The first can be derived from scenarios and end-user interviews, while the latter are enforced bottom-up by technological constraints.

One of the most important overall requirements is the flexibility of the platform: it should be possible to use it in a large diversity of scenarios that share the overall goal: monitor a patient and transfer sensor data using wireless technologies to a healthcare service provider and finally to a healthcare professional. Since too much flexibility can result in a complex application, we must find a balance between flexibility and complexity using frameworks and configurability and clearly defining the required flexibility aspects.

The two most important technological constraints are bandwidth and processing power; the used mobile communication technologies do not guarantee a fixed bandwidth and the available uplink is not fast enough to send a large amount of sensor data. Therefore bandwidth usage must be optimized by, for example, disabling low priority sensors. Mobile devices offer limited processing power and processor usage also relates to power consumption; therefore processor must also be optimized.

We have gathered the end-user requirements from existing requirement documentation, talking to (medical) experts within the Awareness project (Roessingh Research & Development [RRD] and Twente Medical Systems International [TMSI]) who have participated in previous trials and to application developers of the MobiHealth system. We have tried to identify those aspects of the system that are necessary to design a platform and tried to keep the list independent of a specific scenario; therefore the list may not seem to be complete for a system, but we think it is sufficient for the design that we consider in this thesis.

The first section describes the usage scenarios on which the system will be based; this also helps us in verifying our design and prototype, to see if it supports the scenario as it is stated here. The second section focuses on the generic use-cases that can be derived from these scenarios. Based on the scenarios and use-cases, we list the requirements for the system in section three. Section four and five elaborate on security and legal issues, which could not be dealt with completely within this thesis, but are very important for future reference and implementation. The last section concludes on the requirements and introduces our design focus for the rest of this thesis.

3.1 Usage scenarios

Usage scenarios define a situation for which the system can be used; they are often related to a medical condition of a patient, whose quality of life can be improved when he is connected to the mobile e-health platform. The first part of this section describes the leading scenarios for the Awareness project; the second part generalizes these and other scenarios into four scenario categories of which the specific characteristics are determined in the third section. The fourth section discusses the data types that can be distinguished within these scenarios and the fifth section discusses the required flexibility to be able to implement these scenarios using the platform.

3.1.1 Leading scenarios

There are two leading scenarios in the Awareness project [AW/D1.1]; the first is based on a patient who suffers from epilepsy. Epilepsy is a disorder in which nerve cells of the brain, from time to time, release abnormal electrical impulses; so-called seizures. The seizures in epilepsy may be related to a brain injury or a family tendency, but often the cause is completely unknown.

The patient is equipped with a 24-hour seizure-monitoring system that measures heart rate variability and physical activity. Based on these variables, the system should be able to predict future seizures, which can be communicated to relatives or healthcare professionals automatically, together with the location (acquired using for example GPS) of the patient. If possible, the patient himself is warned by the system that a seizure is imminent. The aim of using this system is to provide the patient with both higher levels of safety and independence in order that he may function more normal in society despite his seizures.

The second leading scenario is based on a patient with chronic musculoskeletal pain. The patient is equipped with a system that monitors the patient's health status and can be used to provide bi-directional feedback. The patient can indicate his current activity and tell the system that he is currently is experiencing pain, so a healthcare professional can tell a patient to change his behaviour. To be able to determine incorrect behaviour, it is important for the health care professional to have detailed vital signs and physical activity measurements

3.1.2 Generalization of scenarios

Although the two scenarios as described above are leading for Awareness, it is important that the platform we design is flexible such that it can be used in a variety of scenarios. Therefore we have analysed the trials and scenarios for the two related projects HealthService24 and MobiHealth and combined these into four generic scenario descriptions, which can be used for further requirement analysis.

Continuous patient monitoring

When using the system for home monitoring, a patient is equipped with a BAN and instructed on its usage. The patient wears the BAN as much as possible and data is sent to the hospital or a data centre, from where it can be analyzed by healthcare professionals (HCP) or specialized backend servers. This is used when the patient suffers some chronic disease or requires some extra monitoring after returning from the hospital.

Temporary patient monitoring

The patient uses the BAN on request of the HCP during specific activities. The HCP can monitor the patient's vital signs and other sensor data and send messages to the patient, asking him to, for example, walk faster or

slower. This method can also be used to monitor changes in the patient's health over a longer period of time, if the activity is repeated in the same setting.

Healthcare professional walk-in

The HCP visits a patient and connects a BAN to the patient; the patient does need to have knowledge of the system. The HCP can then either stay, perform some measurements that are sent into the patient's electronic dossier and leave with the BAN. Or the HCP can leave the BAN at the patient after ensuring a proper setup and retrieve the BAN a certain period later (24 to 48 hours). During that period, the BAN has sent data to the backend and a HCP can analyze this data and draw conclusions regarding hospitalization, medication etcetera.

Trauma monitoring

A trauma team is equipped with one or more BANs. Upon arrival at a trauma scene, they can connect the patient and send data to the hospital immediately. Hospital HCPs can then monitor the patient and respond accordingly, for example by preparing for surgery, reserving equipment. Moreover, they know the situation of the patient upon arrival. This trial introduces an extra concept to the system: a Vehicle Area Network, or VAN. When a VAN is used, the BAN does not connect directly to the hospital server, but connects to the ambulance. This ambulance contains a router that sends the BAN data to the hospital server, but it may also contain other functionality, like data storage. This VAN was introduced, because in some cases, the Trauma Team may be inside a building or tunnel, where 3G coverage is limited. The MBU sends the data using a medium distance technology, like WiFi to the ambulance and the ambulance has better 3G connectivity.

In this thesis, we will focus on the first three categories and will not look into a VAN; however trauma monitoring may be possible using GPRS/UMTS.

3.1.3 Scenario characteristics

Based on the trial descriptions for MobiHealth [MH/D1.3], HS24 [HS/D3.1] and Awareness [AW/D1.1], we have identified six characteristics that the platform must have to support the trial scenarios. Most possible scenarios for which the system will be used can be described in terms of these properties; therefore they form an ideal basis for defining the platforms requirements.

- **Session duration:** The duration of a session can range from a patient who needs to be monitored 24/7, for example to notify him of an imminent seizure to a patient who only requires monitoring for a short training exercise.
- **Data transmission mode:** The data transmission mode indicates whether data should be transmitted continuously, or only after a certain event (e.g. a seizure).
- **Data density:** The amount of data that needs to be sent; some scenarios suffice with a heart rate and a location, other may require a continuous 8 channel ECG to be transmitted. This can fluctuate within a session, e.g. after a specific event or on request.
- **Allowed data delay:** The delay is defined as the period between actual data measurement at the patient and the arrival of the data at the screen of the healthcare professional. For inter-active sports scenarios, this delay must be minimized.
- **Data types:** What kind of vital signs and other data needs to be transferred is discussed in the next section.
- **MBU operator:** In most scenarios, the patient is equipped with an MBU to monitor his personal data, but it is also likely that field nurses are equipped with an MBU and monitor multiple patients each day.

For some scenarios it is necessary to make a trade-off in these properties, for example, 24-hour monitoring with minimum delay and maximum data transmission is hard to achieve with current technologies; therefore it is important to prioritize the properties and define system behaviour for each scenario. Table 3-1 maps the characteristics on the generic scenario categories and the leading scenarios. The values are based on regular scenarios that match the generic category.

	Duration	**Data mode**	**Density**	**Delay**	**Operator**
Continuous	Continuous	Depends	Depends	Short if DSP used	Patient
Temporary	Max. 1 day	Continuous	Depends	Short if real-time feedback	Patient
Walk-in	Max. 2 days	Continuous	Depends	Depends	Healthcare prof
Trauma	Max. few hours	Continuous	High	Short	Healthcare prof
Epilepsy	Continuous	Event	High	Long if DSP at MBU	Patient
Musculoskeletal	Continuous	Continuous	High	Short if real-time feedback	Patient

Table 3-1 Scenario characteristics mapped onto scenario types

3.1.4 Characterisation of data

There are several types of data that the system will need to transmit; these can be categorized as follows:

- **Continuous vital signs:** These vital signs need to be represented as a waveform to be interpreted; it is sometimes also possible to derive other information from these vital signs, for example, it is possible to deduct a heart-rate from ECG measurements.
- **Discrete vital signs:** These vital signs can be represented as numbers, for example a heart rate.
- **Data blocks:** These are binary data objects, like images or short videos that are necessary for diagnosis and treatment.
- **Events:** Events are simple messages from one part of the system to another, for example information on the patient's activities, an alarm or the patient's device status.
- **Conversation & Questionnaires:** It is sometimes necessary for a healthcare professional to give real-time feedback to the patient, and the patient may also be asked to report activities or change in medical condition.

3.1.5 Flexibility

Since flexibility is one of the focus points for our design, we will discuss the two main flexibility axes for the mobile e-health platform. These flexibility axes contain several sub-requirements on flexibility, which we have deduced from the scenario properties and the possible data types that the system transmits. Based on these generic flexibility requirements, we can state the more concrete technical requirements for our platform in the following sections.

(1) **Implementation flexibility:** it is important that the implementation is flexible, allowing developers to implement new or improved functionality, the design must support at least:

a. *Defining internal component behaviour:* the components designed in thesis must be easily replaceable with an improved version, so functionality must be separated into components;
b. *Adding functionality to support new sensor devices:* different scenarios require different sensor devices, these may be of a different brand and use different connection protocols;
c. *Implementing a new graphical-user interface:* the user-interface is very scenario dependent, both the functionality and the graphical design, it must be easily interchangeable;
d. *Adding a new lower-level protocol:* although the structure of the system will be clearly defined, the protocols used between the distributed components may depend on the security required, the technology used and the efficiency used. Supporting other protocols ensures a clear decoupling of communication concerns from processing logic;
e. *Define interface to communicate with backend client applications:* the backend client application also depends on the scenario and implementing such an application requires extensive domain knowledge. Since this part of the system is out of the scope of our design, we must ensure that the developers of this application can define their own communication means with the platform. Moreover, it allows other kinds of services to be offered from the device;
f. *Add functionality to mobile device:* the mobile device supports the generic functionality. Often, extra functionality is required for a scenario, like signal processing, data analysis and et cetera.

(2) **Runtime flexibility:** the flexibility at runtime defines that the system, while running, must be able to change at least:
a. *The vital signs that are read from a device and broadcasted into the system:* since processing power is limited, processing unused data is a waste of resources and may increase power consumption;
b. *The vital signs that are being transmitted:* as with processing power, also bandwidth is limited and sensor data must only be transmitted if enough bandwidth is available and the vital signs is required at the back-end;
c. *The communication protocol in use:* if the system supports multiple communication technologies, the underlying protocol may be changing at run-time, to ensure uninterrupted sensor data delivery, the communication protocol must be changeable at run-time;
d. *Starting or stopping additional functionality, like digital signal processing:* if sufficient bandwidth is available, it is possible to send a large amount of vital signs. However, if limited bandwidth is available, the bandwidth usage may be optimized by pre-processing the vital signs by, for example, deriving a heart rate from an ECG. Since this requires processing power, it should be disabled when deriving is not required.

3.2 Use cases

Use-cases represent "*a series of interactions between an outside entity and the system, which ends by providing business value*" [KUL03]. Defining the use-cases for a system that should be as flexible as the mobile e-health platform is difficult, since the interaction largely depends on the scenario in which the platform is used. However, it does give insight in the actors, sub-systems and the main interaction points. Therefore we define very generic use-cases for our platform.

3.2.1 Actors

Within the mobile e-health platform, we have identified five actors that deal with the system regularly. The actors in this case are roles, so it does not need to be a specific person within an organization. The first four actors only deal with the system when it is deployed and will be using the system on a regular basis.

- **Patient:** The patient is the person that will be monitored using the system. It is assumed that this person has no experience in working with medical equipment, telecommunication equipment or PDA's;
- **Healthcare professional:** The healthcare professional (HCP) monitors the patient and is directly involved with the treatment of the patient;
- **Voluntary aid person:** In some scenarios, a voluntary aid person is contacted by the system in case of an emergency;
- **System administrator:** The administrator is responsible for managing the devices after system deployment. He is responsible for configuring devices for new patients and keeping track of all devices;
- **Application developer:** The application developer is responsible for implementing a specific scenario and making adjustments to the core system if this is required by the scenario. This also includes integration into the hospital infrastructure. The developer is a different kind of actor than the previous four, since it deals with the implementation of the system. However, since this research focuses on a platform that will require future work, it is an important actor for our requirements analysis.

When designing use-cases, external systems can also be defined as being actors, we have identified two potential external systems that interact with the platform:

- **External patient device:** An external patient device is the device that is carried by the patient and provides data to the system. This data can be of any type, for example vital sign data or GPS location;
- **Hospital patient management system:** Most hospitals have their own patient management system, where patient information is stored. It is very likely that the platform needs to be integrated with these systems and therefore interaction between the platform and this system is required.

3.2.2 Systems

The actors as described above can interaction with the system using the following systems:

- **Mobile Base Unit:** The MBU is the device that the patient carries; therefore it is a mobile device. It communicates with the backend system and can be connected to external devices. There are multiple MBUs active within the platform, where each patient has one MBU;
- **Backend system:** The backend system is the part of the system that receives the data from all MBUs and processes this. External hospital systems communicate with this system to retrieve the data they require for the patient files, healthcare professional and administrator applications also use backend functionality;
- **Healthcare professional application:** The healthcare professional uses this application to view patient sensor data and for communication with the patient. It is connects to the backend system, so it can also be seen as a part of the backend system;
- **Administrator application:** This application enables the system administrator to manage the devices, it is also linked up to the backend system;
- **Voluntary carer device:** A voluntary aid person carries this device to receive patient status information that allows him to respond in an emergency situation.

3.2.3 Use-case description

Since exact system interaction relies too heavily on the scenario properties, we will only provide a top level use-case, showing the most common interactions at a highly abstract level. This gives us an overview of the relations between the actors and the systems, as is shown in Figure 3-1. The relations between the healthcare professional and administrator applications and the backend system are not shown.

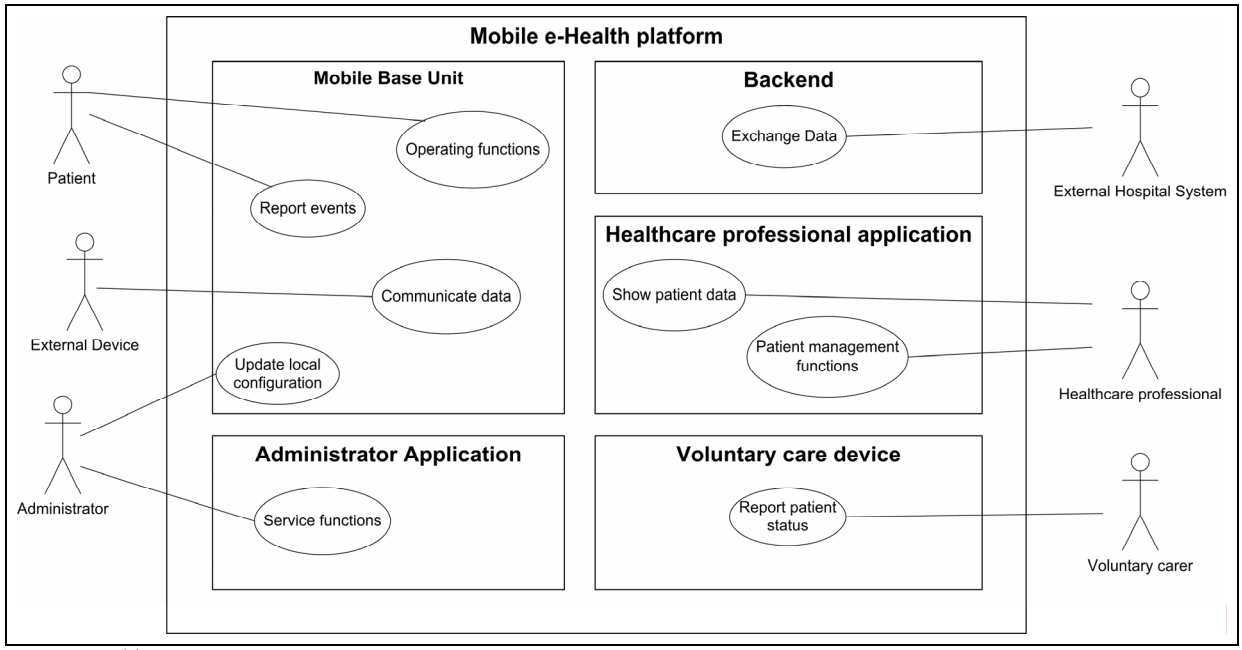


Figure 3-1 Top-level use case diagram

The table below gives a short description of the use-cases illustrated in Figure 3-1.

#	Use-case	Actor	Description
Mobile base unit			
(1)	Operating functions	Patient	Defines the operations that the patient can perform on the MBU, like starting a session and reporting activities.
(2)	Report events	Patient	Defines what happens when the system wants to report an event to the user, for example an alarm or incoming physician message.
(3)	Communicate Device Data	External device	Defines the interaction between the external device and the MBU. The exact interaction is usually defined by the device manufacturer.
(4)	Update local configuration	Administrator	The administrator can change the configuration of the MBU, this use-case defines how.
Administrator applications			
(5)	Service functions	Administrator	When the patient runs a session, the administrator can check the configuration and status of the device, and possibly make changes to the configuration.
Backend system			

(6)	Exchange data	External system	This use-case defines how a hospital information system interacts with the backend.
Healthcare professional application			
(7)	Show patient data	Healthcare professional	This use-case describes how a physician can retrieve patient sensor data and analyse it.
(8)	Patient mgmt functions	Healthcare professional	Defines the way a healthcare professional interacts with the patient or his device, for example giving instructions or changing the sensors that are being sent.
Voluntary carer device			
(9)	Report patient status	Voluntary carer	Voluntary carers have their own device and can receive patient status and location in case of an emergency.

Table 3-2 Top-level use-cases descriptions

This thesis focuses on use-cases 1, 2, 3, 7 and 8 and therefore links the requirements to these use-cases; we have not found requirements that can be mapped on the other use-cases and are not covered by the requirements identified for our focus use-cases.

3.3 Requirements

This section describes the technical requirements that can be derived from the scenario descriptions, flexibility requirements and use-cases. The end-user requirements that are necessary to define MBU interaction and healthcare professional and administrator application design will not be discussed, since this is out of the scope of this research. This restriction directly relates to the implementation flexibility requirements #1c and #1e as presented in section 3.1.5.

Within this section we use the following definitions:

- **Mobile e-Health System:** The combination of one patient with an MBU, sensor device, its surrogate and associated back-end applications;
- **Mobile e-Health Platform:** all systems, the backend network and all related applications.

In the remainder of this section we will use the short term System and Platform to denote the Mobile e-Health System and Mobile e-Health Platform respectively.

Furthermore, we use the following definitions for identifying the importance of a requirement:

- **must:** this requirement is strictly necessary, either by the end-user or for the system to operate correctly;
- **should:** this requirement is important, but not strictly necessary;
- **may:** this requirement may safely be ignored; however, it would be nice if it was implemented.

The related column in the table shows where the requirement was derived from or related to; **U#** indicates a Use-Case and its number, **F#** indicates a flexibility requirement and **R#** is used to refer to a requirement. Technical constraint related requirements are indicated with a **T**.

The first section describes the generic requirements and overall system behavioural requirements. The second section focuses on MBU specific requirements, which are related to the generic requirements, but deal with MBU specific functionality. The third section describes the requirements that we have identified for the backend network and the fourth section focuses on some non-functional requirements. The last section describes the requirements that arise from an application developer's perspective, what is required to reach a scenario specific implementation of the system. Security requirements are discussed in section 3.4.

3.3.1 Generic requirements

The generic requirements describe what functionality the platform must support for its basic behaviour.

#	Description	Related
1.1	The system must be able to transfer real-time sensor data from a mobile patient to a healthcare professional or an automated backend application.	U7
1.2	The system must support messages and notifications from patient to healthcare professional and vice versa, the types of messages and notifications can be defined per scenario.	U1,U2,U8
1.3	The system must support sending system generated messages and notifications to both patient and healthcare professional.	U1,U2, U8,U9
1.4	The system should be able to prioritize messages and notification, for example to ensure an alarm is transmitted as soon as possible.	R1.3
1.5	The system should support bi-directional transmission of binary objects (like images, videos).	U5, U8
1.6	The platform must be designed and implemented in such a way that it can be used for a variety of scenarios.	F1
1.7	To support #1.6, the system should be designed in a modular way; thus defining the system in term components and their behaviour, to allow easy implementation for new scenarios.	R1.6, F1
1.8	The system should be able to deal with temporary loss of connectivity between the mobile components and the fixed network, without losing sensor data.	F2c
1.9	The real-time sensor data must be flexible; it must be possible to change the configuration of sensors that are being transmitted from the MBU to the backend.	F2b

Table 3-3 Generic requirements for the mobile e-health platform

Although viewing historical sensor data is often also mentioned as a requirement for the platform, it is not included in this list; because we see this as a backend application and is left for future development.

3.3.2 MBU requirements

The MBU requirements are listed below; we stress that only the requirements for core functionality and required flexibility are listed here.

#	Description	Related
2.1	The MBU must minimize bandwidth usage and should not send data that is not required at the backend application. Selecting the required data is scenario dependent.	T
2.2	The design must support flexible user-interfaces with different levels of functionality and appearance.	F1c
2.3	The MBU must be able to show status information regarding the session; the detail depends on the scenario.	U2

2.4	The MBU must support store-and-forward: a certain amount of sensor data is buffered and can be sent to the backend on request or after a certain event. This reduces bandwidth usage.	T
2.5	The MBU must be able to support multiple communication technologies and protocols.	F1d
2.6	The MBU should support seamless (for backend) switching between these technologies and protocols.	F2c
2.7	The MBU must support different device types to be connected for sensor data.	F1b, U3
2.8	The MBU should be able to report status information to the backend, like battery status.	U5, U8

Table 3-4 MBU specific requirements

Requirement #2.1 is introduced because bandwidth is often limited at a mobile device and because using 2.5/3G networks is often related to high costs; moreover, if bandwidth usage is optimized, more sensors can be relayed; #2.4 is necessary for the scenarios with the data transmission mode in which an event triggers sending sensor data.

3.3.3 Backend requirements

Developing a backend application is beyond the scope of this thesis. However, we must keep in mind that data should eventually be transmitted to the backend. The following requirements will help us designing the structure of the backend network, which allows backend developers to write their applications more easily. Requirements for a specific backend application are not included (like storing historical sensor data).

#	Description	Related
3.1	The backend should be able to retrieve an estimated delay of the real-time sensor data	U7, U8
3.2	The backend must have access to the sensor configuration of the MBU	R2.1
3.3	The backend must be able to subscribe to sensor data	U7
3.4	The backend must be able to receive and send notifications to the MBU	U5, U8
3.5	The backend must support multiple sensor data viewers (healthcare professionals and/or other backend servers) for a single patient	U7

Table 3-5 Backend related requirements

Requirement #3.1 is important for a physician to link sensor data to activities; #3.6 5 depends on the scenario, but ensures the system is not designed to function on a limited network only and #3.7 6 ensures the platform can be used to store sensor data and also support real-time viewing of sensor data at a different location.

3.3.4 Non-functional requirements

The non-functional requirements state explicit requirements that are related to the technical requirements as defined in the introduction of this chapter. Security is also a non-functional requirement, which we discuss in section 3.4.

#	Description	Related
4.1	According to the challenges in the first chapter, the light-weight platform must consume less than 60 percent CPU power for other processes and to allow additional functionality.	T
4.2	Depending on underlying communication technology and scenario, latency must be minimized; therefore any buffering within the system must be configurable to control system-added latency.	T
4.3	The system must be scalable in such a way that it is possible to add many MBUs without changing the implementation, but by only adding extra servers and possibly changing configuration.	T

4.4	Manageability of devices is also a requirement for a scalable system; the design must allow remote management of the MBUs.	U5

Table 3-6 Non-functional requirements

Requirement #4.2 is important for the allowed data delay scenario characteristic and #4.3 is required for commercial deployment of the platform.

3.3.5 Application development requirements

Because the design presented in this thesis provides a platform for multiple scenarios, it is inevitable that several scenarios require additional functionality in the system. We have listed the following requirements, to simplify the task of the developers who implement this functionality.

#	Description	Related
5.1	The system must be designed in a modular way, allowing application developers to use and improve components, without affecting other parts of the system.	F1a
5.2	Adding new devices (inputs) should be possible with minimal effort (i.e. only writing a 'driver')	F1b
5.3	The platform must provide a communication API for application developers that allow them to write distributed components within the platform.	F1

Table 3-7 Application development requirements

3.4 Security

There are four major security areas for computer and network security: *authenticity*, *confidentiality*, *integrity* and *availability*. For each of these, all possible threats must be analyzed and precautionary measures must be taken. Since the platform deals with medical records and patient information, especially confidentiality is an important issue.

However, analysing security schemes and designing how to secure the platform is out of the scope of this thesis. We do list certain points of interest where security is considered to be important and throughout the design we will point out where and how security may be implemented. [MAR03] describes a thorough security analysis for the MobiHealth system.

3.4.1 MBU

For the MBU, the following security issues have to be considered:

- **Physical protection of the devices:** the MBU and medical sensor device are usually small and expensive devices; theft must be discouraged by, for example, securing the device such that it can only be used as an MBU and when a device is lost, it may start to show contact information on how to return the device or possibly send out its (GPS) location.
- **Access & session start protection:** no other person than the patient or the healthcare professional must be able to use the device; to protect the integrity of the sensor data and the privacy of the patient. The device can use a password protection for the mobile e-health application or, for example, fingerprint identification.
- **Protection of data on the device:** medical data and patient information on the device must be protected such that malicious users cannot retrieve this data from the device. Data encryption is a possible solution.

- **Protection of the mobile e-health application:** the application must be protected such that malicious users cannot modify the application allowing them to gain unauthorized access to the backend network.

The first three can be covered at device-level and will not be dealt with in the mobile e-health application. The fourth issue should be covered in the backend network protection.

3.4.2 Backend network

The backend network is connected to the MBUs at one side and the healthcare professional application and administrator application at the other side. It has an important role in relaying data between both sides, while keeping integrity of the data and restricting access to this data.

- **Protection against unknown devices:** the backend network must not allow unknown devices to use its facilities;
- **Protection against unknown clients:** the backend network must deny access by unknown clients, to protect patient information and medical data;
- **Limit access to patient data:** patient information and medical data should only be available for their own healthcare professionals. The backend network must ensure that the data is protected and only allow access for authenticated and authorized users.

3.4.3 Communication

The main purpose of the system is to transfer medical data from one side to the other; therefore communication is very important. If the system uses open networks, like the Internet, security is extremely important; the four security areas, as mentioned earlier in this chapter, especially apply here. Many threats can be neutralized by a standardized cryptographic protocol, like Secure Sockets Layer (SSL) and Secure HTTP (HTTPS). It is also recommended to communicate only a patient number instead of his full details, to ensure his privacy even more.

The implementation flexibility requires that the lower level communication protocol should be interchangeable; if communication security is required, it can be included in the lower level protocol.

3.5 Legal requirements, certification & standards

This section gives a guideline for the legal requirements and certifications that may apply for the mobile e-health platform. It is likely that different legal requirements or certifications depend on the specific scenario for which the system is built; therefore it is necessary to check any additional requirements for all scenarios. Although analysing these requirements thoroughly is not feasible within the scope of this thesis, we give a short introduction on the subject to give a starting point for further analysis.

3.5.1 Legislation

For both Europe and the United States (we have not looked into any other countries), legislation is very stringent for medical devices. The mobile e-health platform should be considered as a medical device, since it deals with the transport of medical device data. Not only medical device legislation is important, but since the data is transferred over the Internet, also privacy regulations are very important.

For the Netherlands, privacy legislation is defined in the 'Wet op Bescherming Persoonsgegevens' or 'Personal Data Protection Act' [SAU02]. This legislation defines that personal data should be protected as much as possible when public networks (like the Internet) are used [BOR01].

The legislation for medical devices mainly states that medical devices require certain certifications, which means they must adhere to several standards. The legislation also require a well-defined 'intended use' specification, this specifies what the device should be used for, what it is not capable of and under what conditions it should be used. Furthermore it must specify possible failures and hazardous situations that may arise from those failures.

3.5.2 Certification & standards

The European Union requires medical devices to be certified by an appointed body. Certification can be obtained when adhering to certain standards. This section shows the applicable standards for the mobile e-health platform, however, this list may not be conclusive.

Medical Device Directive 42/93/EEC

This directive describes the requirements for an obligatory CE certification [MDD03]; it contains a list of essential device requirements, a device classification and requirements on the quality assurance process. Most of the essential device requirements focus on patient safety and sensor connectivity; because the platform uses external devices that need their own certification, many of these requirements are not applicable for the system.

The device classification defines certain classes for medical devices; however it is difficult to classify the mobile e-health platform, since Annex IX of the MDD states:

"Software, which drives a device or influences the use of a device, falls automatically in the same class."

So the class depends on the device that is connected and whether or not the system drives or influences this device.

Further, the MDD states that the manufacturer of a medical device must ensure using an approved quality system and that a so-called notified body must audit this quality system.

CENELEC – EN 60601-1 standard

The European Committee for Electrotechnical Standardization (CENELEC) has published several standards that deal with medical electrical equipment under the EN 60601 standard. Part 1 of this standard (EN 60601-1), describes the "General requirements for basic safety and essential performance", which partly applies to the mobile e-health platform. Especially section 1-4 is important: "Collateral standard: Programmable electrical medical systems", which describes covers requirement specification, architecture, detailed design and implementation software development, modification, verification and validation, marking and accompanying documents [CENELEC].

The EN 60601 standard is based on the international IEC 601 standard, of which part 601-1-4 is discussed in [HER95].

3.5.3 Quality assurance

System certification is mainly based on the quality of the system. If the system is of bad quality, it is unlikely to be certified and thus cannot be sold. Quality assurance is an umbrella activity that, if done correctly, ensures a good quality of the system [PRE00].

Both the MDD 42/93/EEC as well as EN 60601-1-4 indicate that having a well-defined development process is important for quality assurance. There are many standards on quality assurance; the most common are the ISO 9000, ISO 9001 and ISO 9004 standards, which are general quality management standards and can also be used for software quality. The ISO/IEC 90003:2004 standard provides guidelines on how to apply the ISO 9001:2000 standard to computer software and the IEC/ISO 12207 standard defines software life cycle processes. The Carnegie Mellon University's software engineering institute has defined the Capability Maturity Model (CMM) that identifies best practises to increase process maturity.

Quality assurance is an important activity and the approach should be verified by the certification organization before development of a commercial system can start.

3.5.4 Further reading

There are many more standards and documents on quality assurance and medical device development. These are listed below for further reading and should be studied carefully before trying to bring the system on the market.

- Handbook of Medical Device Design [FRI01];
- ANSI/AAMI standards; especially SW68 (Medical device software-Software life cycle processes) [AAMIWEB];
- FDA Guidance document #337: Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices [FDA05];
- FDA Guidance document #558: Off-The-Shelf Software Use in Medical Devices [FDA99];
- IEC 60601-2-27 Standard: Specifies requirements regarding electrocardiographic monitoring equipment. This standard is interesting because ECG monitoring is the base for some trial-scenarios

3.6 Conclusions

This chapter presents an overview of the requirements for the platform. Because we are designing a platform, it is difficult to gather the end-user requirements, since many possible scenarios must be supported. Therefore we have listed those requirements that are necessary in a framework that allows implementation of a variety of scenarios.

3.6.1 Base platform

In the previous chapter we left the decision on continuing work with on MobiHealth system or starting a new design trajectory, using MobiHealth knowledge. The MobiHealth system only supports real-time sensor data; of which the configuration is not flexible. When looking at the requirements, we see that much functionality is not yet included in the system; as concluded in the previous chapter, adding functionality to the current

implementation is a time-consuming task. Future developers might also have difficulties in studying the system, due to the complexity and lack of design documentation.

Therefore we think, on the long-term, it is better to do a complete redesign, using the design-knowledge gathered from studying the MobiHealth system, where this thesis will form a base design document. Although this might require some extra work at this moment, it is likely to save time in the near future and improve the overall quality of the system.

Proper design documentation is also required for certification of the platform, which is an essential step when working towards a commercial stability; this thesis will not be sufficient for certification, but provides an important step in the right direction, especially considering the lack of design documentation for the current implementation.

3.6.2 Design focus

A complete redesign of the platform is a large project, so it is impossible to give a complete redesign and implementation in this thesis; therefore we will only focus on MBU and corresponding surrogate design, for which we will define the main components and their relations. This will give a base that can be used for further design and implementation of the platform. We will design our components in such a way that they can be reused at the healthcare professional applications, to reduce ambiguous design effort.

We cannot design every component in detail; therefore we will define component behaviour and leave detailed design for future work. Although we will define a generic API for these components; component designers are encouraged to add functionality, while using our generic API as a base for compatibility purposes.

We stated that platform flexibility is very important, because it must be suitable for a variety of scenarios. The flexibility of the design relies on a well defined method to transfer sensor data and the interaction possibilities between the components internally and the external entities (the user, backend application and/or healthcare professional). Therefore we will also focus on how data can be represented and communicated using a generic approach.

4 Platform design

This chapter gives an insight into our new platform design; we will use a combined top-down and bottom-up design process, in which focus lies on splitting the system up into components. Figure 4-1 illustrates this process; in step (a), the system is shown as a whole, revealing its interaction points with the environment. Step (b) shows that the system is built up from several components, and step (c) indicates that these components contain sub-components. Depending on the complexity of the component, additional decomposition steps can be taken [VIS02].

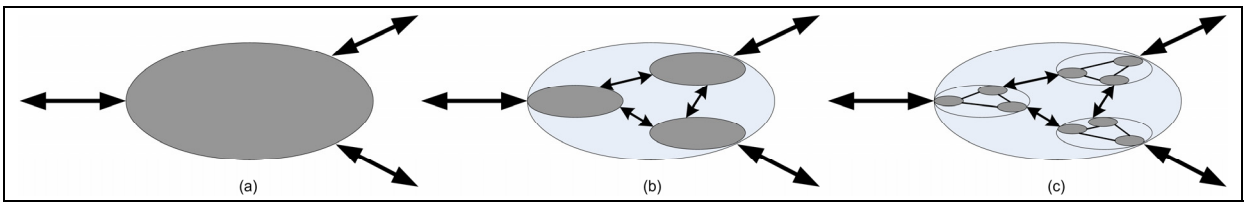


Figure 4-1 Generic component decomposition [VIS02]

Section 4.1 describes the top-level design for the platform, where we illustrate the distribution of the components and their interaction points. Section 4.2 elaborates on the BAN model, which forms our basis for sensor configuration representation. In section 4.3 we distinguish the different types of data that appear in the system and how this data can be represented in the system. Section 4.4 elaborates on the core components that can be used at the different distributed sites. Section 4.5 defines a generic communication framework and what kind of data can be exchanged between the sites. Section 4.6 provides some preliminary conclusions.

4.1 Top-level design

There is not a single top-level model of the system, because it is quite complex. This section will give some viewpoints on the system and its main components. These components will be explained in the following chapters.

4.1.1 System overview

Figure 4-2 shows an overview of the system. It shows the main components of the system and the communication paths between them. There are three sites or domains in the system:

- The **body area network (BAN)**, which contains the MBU, the sensor box and possibly other devices, like GPS modules. The main goal is to gather sensor data and provide communication means for the patient to his healthcare professional.

- The **backend network**, the part of the system that receives the sensor data and makes it available for the components in the healthcare domain, it can also provide data processing and data storage. It also provides functionality for healthcare domain applications to communicate with the MBU.
- The **healthcare domain**, this part of the system covers all applications and systems that need the sensor data for reviewing, monitoring, storage, and etcetera.
-

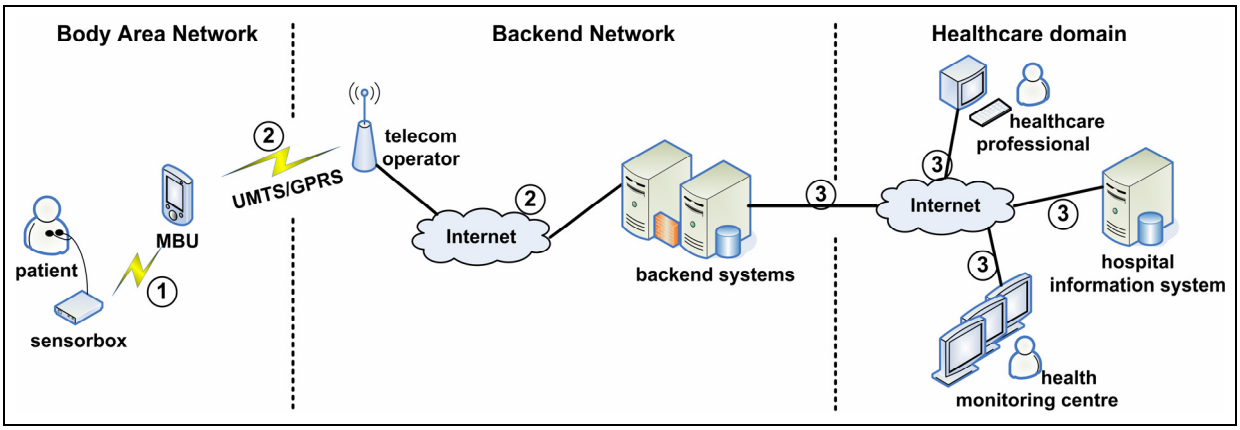


Figure 4-2 Overview for the mobile e-health platform

There are three communication links that are of importance for the system, indicated by the encircled numbers in the figure.

1. **Device/MBU communication**; this communication link is device dependent and is usually a short-range wireless technology, like Bluetooth;
2. **MBU/backend communication**; in the figure above, we have indicated this communication link as being GPRS/UMTS, because this is the narrowest link in the system, it is the bottleneck for sending the sensor data. Data is sent from the MBU via GPRS or UMTS into the telecom operator network, from where it is sent over the Internet to its destination (the backend network). It is also possible that the UMTS/GPRS link is replaced with other wireless technologies, like WiFi, for which a telecom operator is not always necessary;
3. **Backend/Healthcare application communication**; we have indicated that this communication runs over the Internet, but it is also possible that the backend itself is located within the Local Area Network of the hospital. It should be possible that multiple (types of) users (as indicated in the figure) can access the sensor data.

4.1.2 System architecture

The system architecture will be based on the JINI surrogate architecture [SUN03], using the University of Twente's Mobile Service Platform HTTP-interconnect protocol as described in [TOL05] and [ESU05]. We have chosen this architecture, because it proved to be of use in the MobiHealth system and it offers a solution to several problems in mobile communication [DOK03], as explained in the MobiHealth analysis chapter. The Mobile Service Platform uses HTTP to prevent firewalls from blocking traffic; furthermore, it allows the MBU to join a JINI network as a service. By exposing the MBU as a service in the JINI network, healthcare domain applications can be written as JINI clients that can discover online MBUs and we do not need to implement a registry ourselves. We define the following platform components in terms of this JINI surrogate architecture and the generic JINI architecture [SUN99].

Component name	JINI name	Site	Description
Mobile Base Unit (MBU)	Device	BAN	A mobile service that offers sensor data and other

			functionality to the surrogate, to which it connects using different (wireless) technologies.
Lookup service (LUS)	Lookup service	Backend	Used to register & lookup e-Health Services.
MBU Surrogate	Surrogate	Backend	Object in surrogate host that communicates with the MBU on one side and exposes a (sensor) service at the other. This service is registered in the JINI network.
e-Health Service	JINI Service	Backend	The service that the surrogate exposes; it is registered in the LUS by exporting the e-Health proxy. This proxy communicates with the service using a predefined interface (red in the figure).
e-Health Proxy	Service Object	Backend & Healthcare	The e-health service exports the proxy to allow clients to communicate with the service. The proxy offers the service interface (blue in the figure) to the client and communicates to the surrogate via the red interface.
Surrogate host	Surrogate host	Backend	The surrogate host provides an environment to run the MBU surrogates, allowing them to join a JINI network.
Healthcare application	-	Healthcare	The healthcare application provides the end-user functionality, while using the e-health client to indirectly communicate with the MBU.
e-Health Client	JINI Client	Healthcare	The e-Health client is the part of the healthcare application that can communicate to the e-Health service. It retrieves the proxy from the LUS and uses its (blue) interface for service interaction.

Table 4-1 Main platform components overview

Figure 4-3 shows how these components are distributed and interconnected; the Mobile Base Unit is located in the body area network and is connected to the Internet over a wireless connection using a telecom operator. The MBU Surrogate runs in the surrogate host and contains an e-Health service; this surrogate host is connected to the backend network, which is optionally protected by a firewall. The MBU Surrogate registers its e-Health service in the JINI Lookup service, which is also located in the backend network. Finally, there is an e-Health client, which is the healthcare application; this can be located anywhere on the Internet, depending on the firewall settings of the backend network and other security mechanisms used.

Several remarks with regard to this figure:

- Multiple MBUs can be online at the same moment, this results in an extra MBU Surrogate for each MBU and extra e-Health proxies in the LUS;
- We use the term healthcare application for any application that connects to an e-Health service, this can also be an automated system;
- It is possible that these automated systems, like a data repository, are located in the backend network.

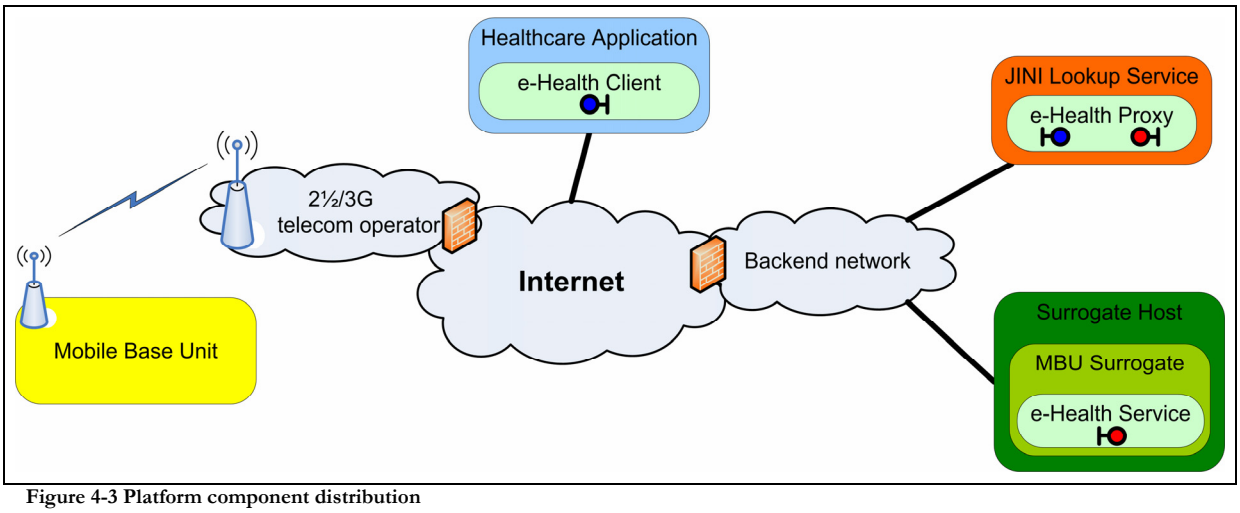


Figure 4-3 Platform component distribution

4.1.3 Component interaction

We have shown a global overview in section 4.1.1 and have defined the main components of the system in section 4.1.2; now we take a look at how these main components interact. Figure 4-4 shows the interaction structure for the system. The numbers indicate the order in which the events may occur.

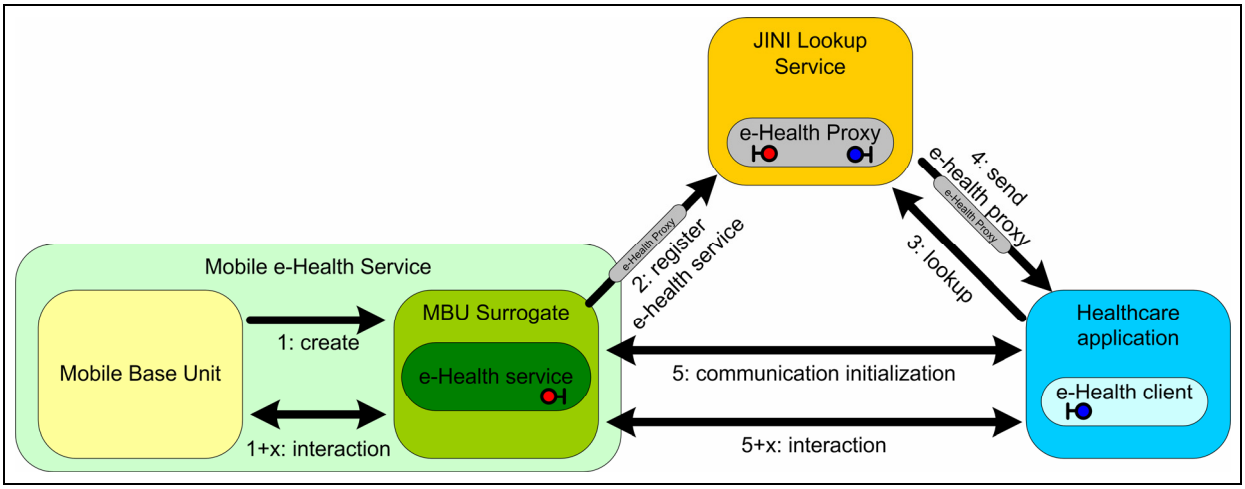


Figure 4-4 Component interaction

1) The MBU is started and connects to the surrogate host, which creates and runs an MBU Surrogate;
1x) After surrogate creation, the MBU can interact with the MBU Surrogate, for example sending sensor data;
2) The MBU Surrogate creates an e-Health service and registers this e-Health service in the Lookup Service by sending the e-Health proxy;
3) An e-Health client requests the e-Health service from the Lookup Service. This can be either user initiated, for example a healthcare professional who wants to see real-time sensor data, but this can also be an automated process, for example a data repository that connects to all e-Health services;
4) The lookup service sends the e-Health proxy to the e-Health client;
5) Using the obtained e-Health proxy, the e-Health client can start communicating to the e-Health service in the MBU surrogate;
5x) After communication initialization (5), the e-Health client can now interact with the MBU Surrogate, and indirectly with the MBU itself.

We call the combination of MBU and MBU Surrogate the Mobile e-Health service, because the actual service logic is located within the MBU, but the service access point is the e-Health service object in the surrogate. These services are sometimes also referred to as *nomadic* mobile services [TOL05], because a mobile service is often associated with a service that is accessed by a mobile client, whereas with a nomadic mobile service the service itself is mobile.

4.1.4 System decomposition

In the previous sections we have shown the main components and the way they interact; this section will show a global decomposition of the components, so we can see what logic is used and what parts within these components we need to design and implement. Figure 4-5 shows this decomposition.

The MBU (yellow) consists of a core, a part that is responsible for the backend communication, the device drivers, MBU plug-ins (optional) and graphical user interface. This thesis focuses on the design of the core and the backend communication; for the other parts we define the structure. The design of the MBU is described in Chapter 5.

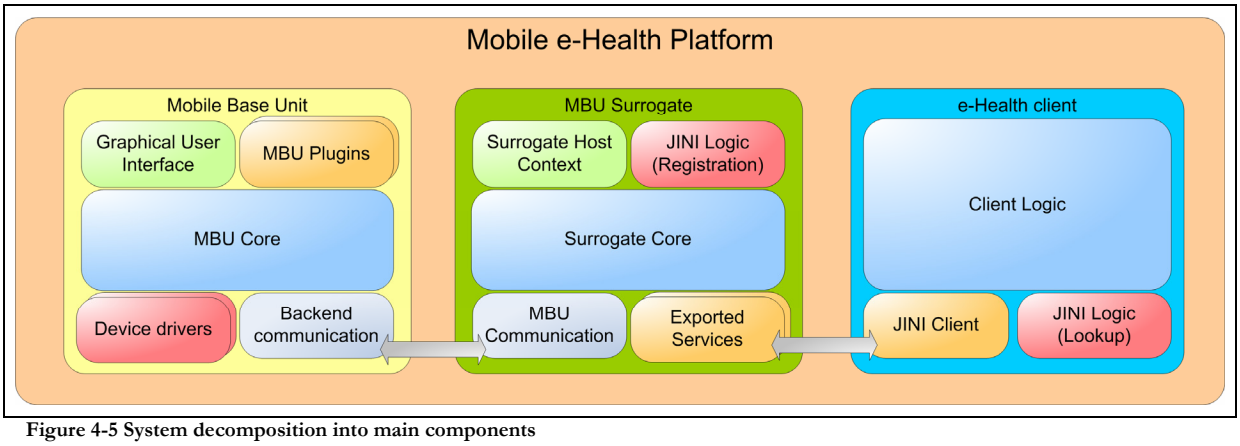


Figure 4-5 System decomposition into main components

The MBU surrogate (green) consists of a core, the MBU communication part and the exported service(s). It requires the already existing JINI logic for service registration and a surrogate host context for information on the connection to the MBU. Our design focuses on the surrogate core and MBU communication. We will also provide a framework for the exported services. The surrogate design is discussed in Chapter 6.

The last part is the e-Health client (blue), which will not be thoroughly designed within this thesis. Therefore we suffice by indicating it requires JINI logic for service lookup and a JINI client to communicate with the surrogate. The e-Health client logic can range from being very simple to extremely complex. We will touch upon this subject shortly in Chapter 6 and our prototype implementation

We will try to design as much functionality as possible in sub-components, which is required for implementation flexibility (see section 3.1.5). Several of these sub-components can be reused at the different main components; this limits the amount of functionality that we actually need to design and implement; and furthermore it results in a more straight forward design, which is helps future developers to understand the system quicker.

4.2 Body Area Network model

There are several definitions on what a Body Area Network (BAN) is; to start with, we use the definition of the Awareness project [AW/D4.7]:

> *"A BAN is a computer network which is worn on the body and which moves around with the person (that is, the wearer is the unit of roaming). It incorporates a set of devices which perform some specific functions and which also perform communication, perhaps via a central controlling device."*

In the mobile e-Health platform, the BAN consists of the MBU as the central controlling device and a set of medical devices and possibly some other devices, like a GPS module. The Awareness definition is too broad for our mobile e-health platform, since the responsibilities of the MBU can be distributed, therefore we rewrite the definition for the mobile e-Health BAN as follows:

> *"The BAN is a computer network that is worn on the body of a patient and which moves around with the patient. It consists of a set of devices, which provide sensor data and communicate with the MBU"*

The BAN can be modelled in an UML diagram as shown in the next sections. The main purpose for modelling the BAN in MobiHealth was to have an MBU object that could be exported as a JINI service. By traversing the BAN model, the MobiHealth clients could request any sensor they are interested in.

The model is also useful for specifying the BAN configuration: a simple UML model can be mapped onto an XML document easily. This XML document can be used to initialize the system and to communicate to the backend what sensors are connected.

4.2.1 MobiHealth BAN model

The body area network model, or BAN model, shows how a BAN is structured. The MobiHealth system design used the BAN model as shown in Figure 4-6.

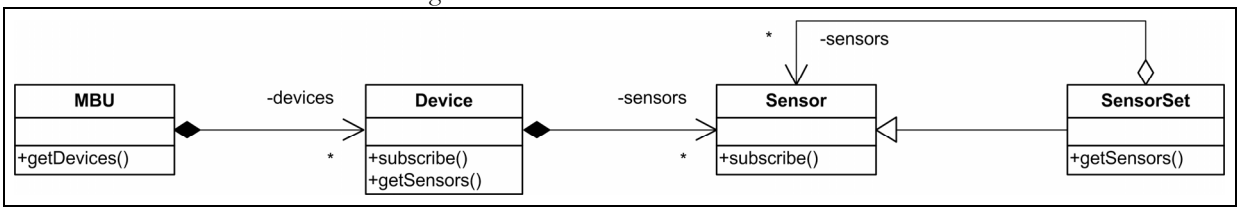


Figure 4-6 Current BAN model

The design is simple: the BAN contains an MBU, the Mobile Base Unit. This MBU is connected to several devices, like a sensor box or GPS module. These devices have a certain amount of sensors that supply the sensor data. These sensors can be grouped in a sensorset, which itself is also a sensor. A component interested in sensor data can subscribe for the data either per sensor or per sensorset, as explained in Chapter 2. The model contains a partial composite pattern [GAM94]: a sensorset is a sensor itself, and contains a certain amount of sensors; it can also contain other sensorsets.

In the MobiHealth implementation, this model was not implemented completely. The main cause for this was that the system implementation focused on the TMSi Mobi sensor boxes and the Portilab application. The Mobi sensor boxes send their sensor data at one specific frequency for all connected sensors; the Portilab application

requires all incoming sensor data to be at one frequency and synchronized, so it can actually handle only one sensorset at a time. Therefore, some workarounds were introduced in the system: An MBU contains one or more sensorsets that consist of one device with a certain amount of sensors and a special composed-device type was introduced to support multiple devices. The functionality to subscribe for one sensor is also not implemented; it is only possible to subscribe for a sensorset.

Recent research in the AWARENESS project [AW/D4.7] aims at a system where the BAN consists of a combination of end-devices that represent sensors and intermediates that are linked to end-devices or other intermediates. The platform we present in this thesis should be seen as one large intermediate, which can also be split up in several smaller intermediates.

4.2.2 Using the BAN model in the mobile e-health platform

As said in the introduction of this section, in MobiHealth the BAN model is used for exporting an object that provides a description of the configuration and can provide access to the sensor data. Using the JINI surrogate architecture, the designers of the system envisioned that an MBU object, which was generated at the MBU, would be available to any backend party interested in the sensor data. By traversing this object, they could subscribe for sensorsets or sensors individually, resulting in many points of subscription.

We think the general idea of having a MBU object is interesting; however, it is not suitable as the actual service that should be offered to a backend, because it increases the complexity of the application. All sensors must maintain a list of subscribers and there is no central point in the system where subscribers can be found. It also increases complexity on dealing with relaying sensor data from one site to the other; because, sensor data should not be transferred if there are no listeners at the backend (requirement #2.1). In our design, we still use the concept of a BAN model, but primarily to describe configurations, on how the sensors are connected to the system and what sensors are available at a specific site.

4.2.3 Proposed BAN model

The MobiHealth BAN model uses sensor sets to group sensors for the healthcare professional application; the client application shows the sensor data for a specific sensor set. These sensor sets are predefined in the configuration. Since healthcare professionals are not interested how the data is transmitted, nor how it is gathered; we think the grouping of sensors into a sensor set for viewing, is a responsibility of the healthcare professional application, not of the platform. Because designing and developing this application is not in the scope of this research, we will focus on how to use the BAN model for data transmission and data collecting, which allows us to redefine the BAN model.

When considering devices like the TMSi Mobi or a GPS module, they do not classify sensors in specific sensor sets and they offer the data either as one block, or on a per-sensor basis, not per sensor set. Therefore, we can model a concrete BAN as in Figure 4-7; this simple model shows the MBU that is connected to some devices and sensors are connected to these devices.

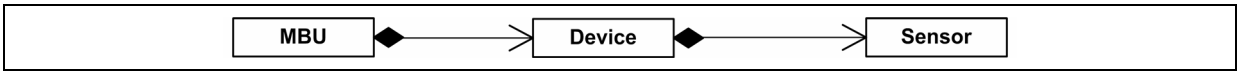


Figure 4-7 Concrete BAN model

This concrete BAN model can be used to give a configuration of the MBU in terms of which sensors are connected. However, to process the data from the sensors within the system, we suggest grouping similar and synchronized sensors that run at a certain frequency to allow more efficient internal processing and transmission of the sensors. This also allows us to remove a group of sensors from the system and to choose whether or not to relay the group, which is necessary for the runtime flexibility (see section 3.1.5). We call a group of sensors *channel*, which can be compared to a SensorSet in the current implementation. It is possible to have multiple channels at the same frequency for one device.

The new BAN model as shown in Figure 4-8 shows the main goal of having a BAN model in our design: it is used to describe a configuration of sensors and because it is very generic, we can use it for different types of configurations. The main configuration we work with is the regular *SensorConfiguration*, which contains a certain amount of *ChannelDescriptions*. Each channel runs at a certain frequency, and produces samples of *Bytesize* bytes and can be identified by a unique (for this MBU) *ChannelID*. The channel consists of *SensorDescriptions* that describe the actual sensors using various properties, which we have derived from the current implementation. The *Bytesize* and *Offset* are used to describe what part of a channel-sample contains the sensor-sample data. We have explicitly not defined a *setSensor* operation in the *ChannelDescription*, because the channel-data content should not change while the channel is in use. The channel description within the sensor description should always be set, because a sensor description alone can not be used to obtain its sample data.

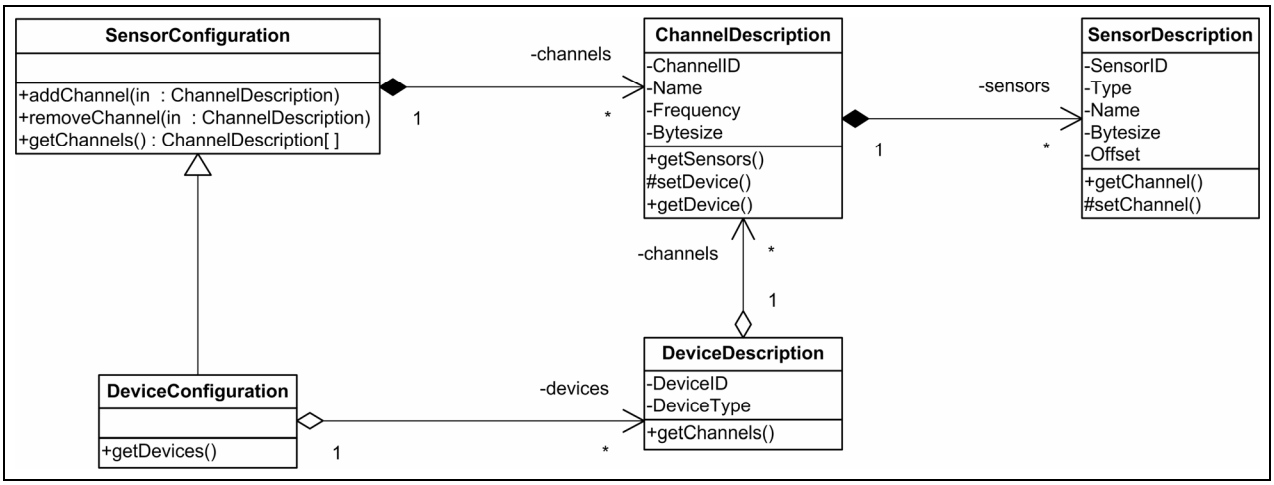


Figure 4-8 BAN Model for sensor configuration

We have moved the devices to a special type of configuration, the *DeviceConfiguration*, because the concept of a device is often not required by internal system components. However, it is likely that the healthcare professional or an administrator needs to know what devices are connected to the MBU; so this configuration must be available for the backend. When a DeviceConfiguration is used, the device description at the channel description must be set when the device description is created.

4.2.4 Sensor configurations

It is possible that the sensor configuration of an MBU changes, for example when the patient is equipped with an extra sensor device or an improved device. Especially for the walk-in scenarios it is important that the sensor configuration does not need to be pre-defined at the backend. Therefore we will define flexible sensor configurations that originate at the MBU, because the sensor data originates at the MBU.

Furthermore, because we want to support enabling devices and sensors at the MBU and change whether or not sensor data will be relayed to the backend, we can identify different sensor configurations. For the MBU, we have identified the following configurations:

- **Device configuration:** The collection of sensors that is available from the devices in the BAN, in other words, the concrete BAN, with the sensors grouped into channels; for this the special *DeviceConfiguration* type as described above is used;
- **Derived configuration:** The collection of derived sensors that can be generated within the system;
- **Sensor bus configuration:** The collection of sensors that is available in the sensor bus; this is a subset of the device configuration combined with a subset of the derived sensors;
- **Relay configuration:** The collection of sensors that is being relayed to the surrogate, this is a subset of the sensor bus configuration.

Initial configurations can be communicated to the surrogate using data blocks, and if there is any change to a configuration, this can be sent as an event. The surrogate can receive the sensor data from the MBU and therefore has its own sensor configurations:

- **Relay configuration**: The collection of sensors that are received from the MBU, which is the same as the relay configuration at the MBU;
- **Sensor bus configuration:** The collection of sensors available in the surrogate sensor bus; usually, this is the same as the incoming configuration, but it is possible that additional signal processing is included in the surrogate and that the sensor bus contains more sensors (signal processing at the surrogate is not considered in our design);
- **Backend configuration:** There is a backend configuration for each connected backend, which consists of the sensors that are being relayed to that back-end and is a subset of the surrogate sensor bus configuration.

4.3 Internal data representation and propagation

This section describes how we will represent sensor data internally and what kind of information can be exchanged between components. First, we will define three types that can be used for all data that needs to be transmitted and the subsequent sections will describe these types in detail.

4.3.1 Internal data characterization

We can derive three types of data representation from the data characterization in section 3.1.4; by defining these more abstract types, we support future types of data and reduce the complexity of the application, since we can define generic methods for data transmission. These data types will be exchanged between the internal components as well as the different locations (MBU, surrogate and healthcare application).

1. **Real-time sensor data**: this is the most important type of data for the system, since this is what the system is used for. The data is continuous at a frequency so we can use this data type for continuous sensor data. If discrete sensor data can be published at a fixed frequency, we can also use it for discrete sensor data.
2. **Events**: events represent all types of discrete data that components can exchange. This includes discrete sensor data, internal component events, patient to healthcare professional communication (and vice versa) and alarm events. The events have a certain priority, a destination (specific component, or broadcast) and can have a small payload.

3. **Data blocks**: data blocks are large blocks of data of various types. It can be used to transfer files (like photos, videos), historic sensor data or component configurations.

4.3.2 Real-time sensor data

In the MobiHealth implementation, real-time sensor data is represented as a small data block containing one sample of sensor data for each sensor. This is sent through the system by using a push mechanism, combined with a subscriber/listener architecture: Data originates at a data producer, for example a device driver and consumers can subscribe to this data. The data is pushed into the system by the producer, running in a thread (p1); it is then sent to all subscribers in a sequential order; first the data is pushed to the first consumer, which processes the data, then to the second consumer etcetera. All consumers use the thread (p1) for their data processing. This is the inactive piping mechanism as explained in section 2.3.5 and is illustrated in Figure 4-9.

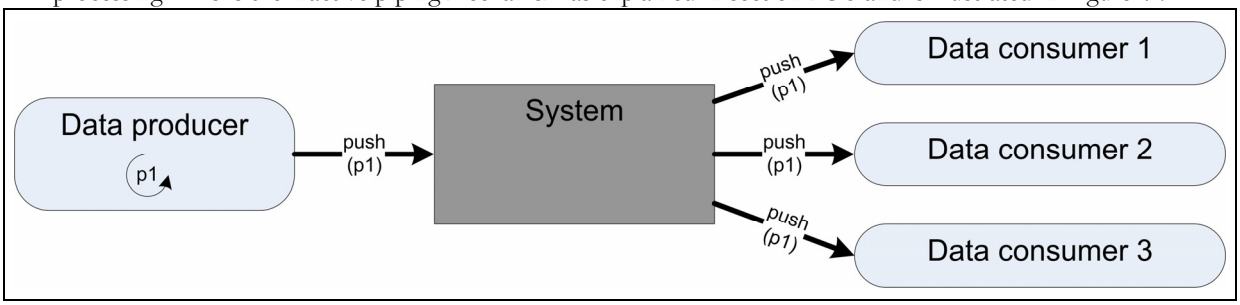


Figure 4-9 MobiHealth push mechanism

Often, a consumer may spend some time on data processing, which blocks the producer because his thread is being used. To solve this, an active pipe was added to the system: the producer sends data into a pipe in the system and within this pipe a thread (p2) ensures the data delivery to all consumers and decouples the producer and consumers. This is the active piping mechanism as discussed in section 2.3.5 and is illustrated in Figure 4-10.

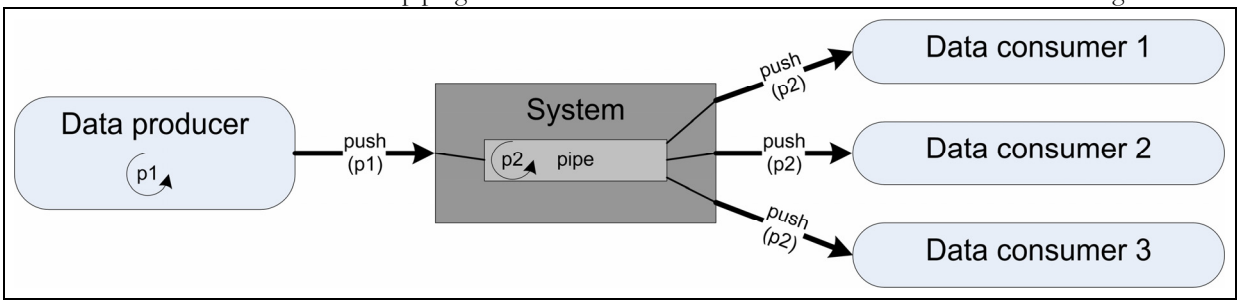


Figure 4-10 MobiHealth push mechanism, with active pipe

Although the pipe decouples the producer and consumers, one consumer can still delay the delivery to the other consumers. A good example of this problem is a consumer that first gathers a large amount of samples and then processes them. The first few pushes will be handled very fast, since the consumer will only buffer these samples, but when it processes, it requires some extra processing power. This can be solved by adding an extra active pipe in each consumer, which adds extra latency and complexity to the system.

In the MobiHealth system, some components act as a consumer and producer in a chain-like manner; they change or process the data, and then push it onto the next component and while some of these components only allow one subscriber, there are also components that allow multiple subscribers (e.g. both the SensorSet and the SensorSetManager). The problem in this approach is that there are different components to subscribe to, for the same sensor data and there is no central location for subscribing to sensor data.

Another problem in this approach is that a very high frequency results in many push-calls (one for each clock-tick), which can cause a high system load if there are many consumers. In many cases, consumers do not require the data to be delivered per sample, but can also deal with blocks of samples. The increased system load may be unnecessary in some cases.

To solve these problems, we introduce a bus architecture where sensor data producers can publish their data and where the sensor data consumers can request the data (subscribe for data). This bus is accessible at a well-known location in the application and any producer component can publish its data and all consumers can subscribe for data. In this architecture, we will use sensor data streams instead of push calls. Data is written as a byte-stream and can be read as a byte-stream as well. If the producer receives large blocks from its source, it can simply write these large blocks into the sensor bus; if it receives data per sample, it can write per sample. Consumers can also read per sample or per large block, whatever they require. The consumers and producers are decoupled, because a producer uses its own thread to publish the data and the consumers also runs in its own thread. We have illustrated this approach in Figure 4-11. Section 0 discusses the design of such a sensor bus.

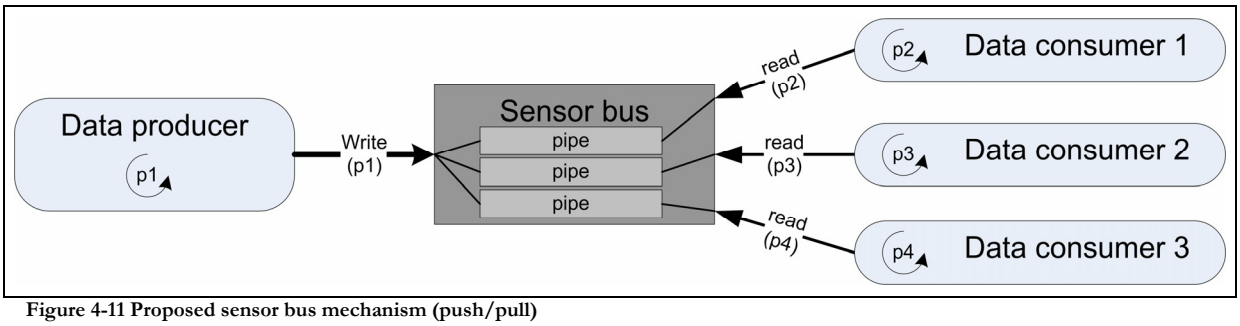


Figure 4-11 Proposed sensor bus mechanism (push/pull)

Since the system deals with real-time data that may need to be synchronized at the healthcare application, we introduce marks. The data publisher must mark his data at a certain interval with a timestamp, such that the data can be synchronized by comparing the timestamp for two channels. This mark can also be used as a reference to a certain data session. We have considered three possible methods for sending this mark:

- **In-band:** The mark is sent into the sensor data stream. To identify this mark, it must be sent at fixed intervals (defined by amount of sensor bytes sent). This has as advantages that it is easy to identify the mark at the receiving side (i) and it does not require a lot more computing power and complex mechanisms (ii). The disadvantages are that it is important to verify the amount of sensor bytes written into the stream to prevent streams corruption (i), and each receiving component must support reading these marks (ii);
- **Out-band, no stop:** The mark is supplied by the sensor data sender for a specific sample (by giving a 'mark' command after sending that sample) and the mark can be retrieved by the receiving side (it must request the 'last mark' and receives that mark and an offset). If the mark is not retrieved, it will be overridden by the next mark. The advantage of this system is that data users not interested in marks can simply read the sensor data; the main disadvantage is that out-band marks will require some kind of synchronization in the stream to link the mark to the right sample.
- **Out-band, with stop:** Sending a mark is like with 'out-band, no stop', but now the receiving side will not receive any more sensor data, unless the mark is retrieved. This ensures accurate synchronization and flexible marking, because the mark-interval is not fixed. However, it requires synchronization between the in- and output of the stream.

We will not choose for a specific solution for synchronization and marking, because it is possible to implement all three, or possibly some other method. If a channel runs at a fixed frequency, it can be synchronized with another fixed frequency channel by keeping track of the amount of samples processed, once the channels are synchronized. For example, a channel A at 128 Hz and a channel B at 512 Hz can be kept in synchronization when the data consumer reads 4 samples of B for each sample of A.

An important property of real-time sensor data is that listeners can subscribe whilst data is already being streamed. The listener does not require the data that was sent before he subscribed. Because of this property, extensive buffering or synchronized start sequences are not required. Since the sensor bus knows the byte size for each sample (from the configuration), it can ensure that samples are delivered correctly when a listener subscribes after the streaming has started.

4.3.3 Events

Since the system will deal with several tasks and is even distributed among different sites; we will use events to let these tasks and sites communicate. We must emphasize that we are not performing true event-driven programming, which focuses on using a single thread and an event queue to process its instructions. Our events are used for all kinds of discrete messaging between internal components. We distinguish two main types of events:

- **Broadcast events**: events that are broadcasted into the system and can be subscribed to by any component. It is possible to keep the event at one site, or relay it to all sites.
- **Unicast events**: events that are sent to a specific component; these can also be classified as **messages**. To be able to send the message to the correct component, addressing of components is required (see section 4.4.1).

Besides this classification, all events have a specific event type, like 'medical alarm', 'buffer overflow' or 'new sensor available'. The events should also support some kind of prioritization, which is mainly necessary for communicating events to other sites and therefore must be transmitted over a network. A small optional byte-payload can also be added to the event, of which the format is event-specific; since we expect many events to be generated and communicated, we must encourage to use a simple format for this payload and not to transfer large amounts of data via events, data blocks can be used for this purpose (see below). The event structure is shown in Figure 4-12.

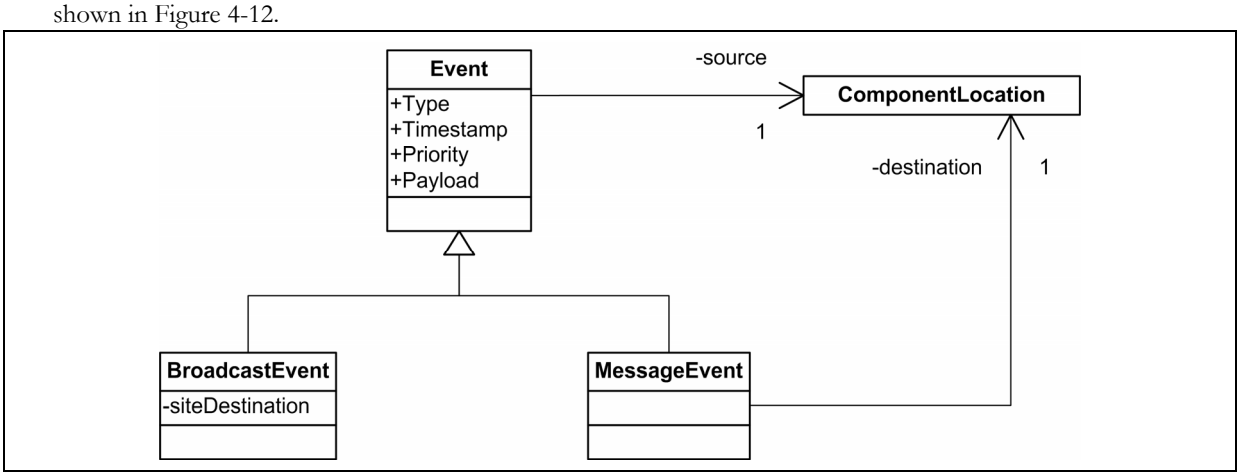


Figure 4-12 Event structure

To support event-distribution, the system will contain an **event bus** where all components can publish their events and can subscribe for other events. Section 4.4.4 explains the event bus in more detail.

4.3.4 Data blocks

It is likely that components sometimes have to send a large block of data to a different site; some examples are files (like photos and videos) and historic sensor data (for store and forward trials). The important difference with real-time sensor data is that the start and end of the stream are clearly delimited and usually, the delay is of less importance.

To deal with this data type, we introduce data blocks; for each type of data (file, sensor data, etc), a specific block must be implemented that follows a standard interface. To be able to transfer these data blocks independently from the lower communication layer; the data must be serialized (represented as a byte-array). After transmission, the data can be deserialized into a data block. We will not define an exact strategy for (de)serialization; however, for Java a standard ObjectOutputStream is an appropriate candidate.

4.4 Core components

To support the data representation as described above and to support system flexibility, we introduce several components that can be reused at the different sites. They are always present in the MBU and the surrogate, and can also be used at a backend implementation (depending on the complexity of the backend implementation). Figure 4-13 shows these core components and how they are related; the following sections explain these components.

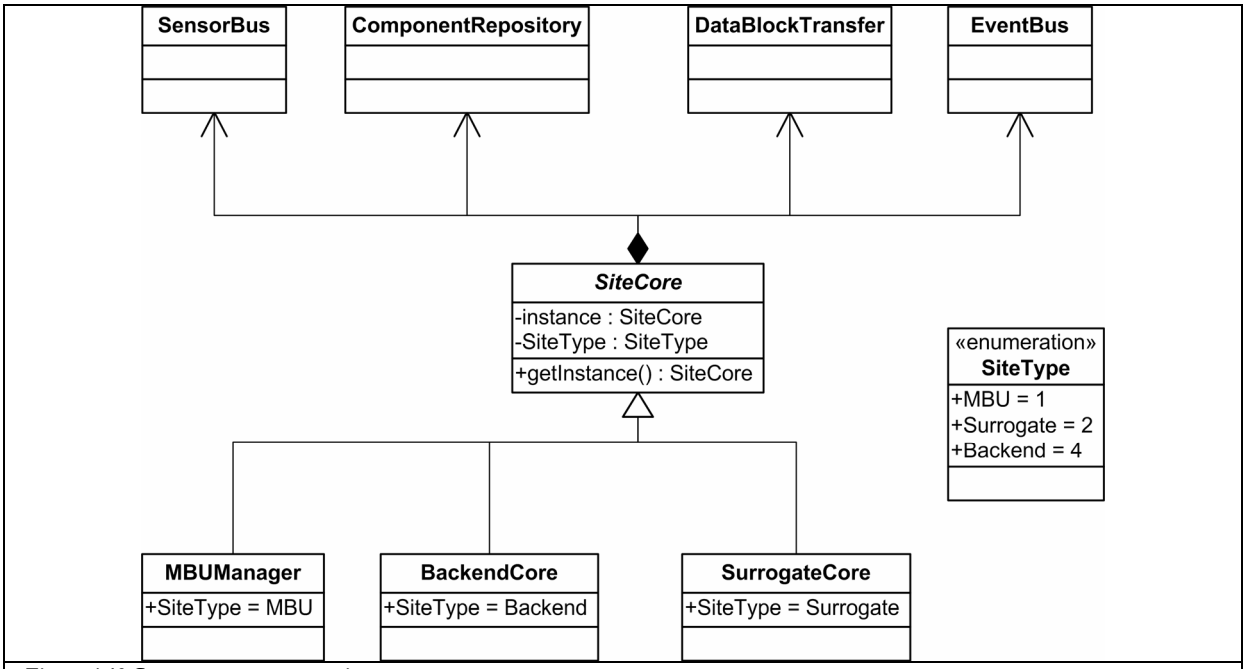


Figure 4-13 Core components overview

4.4.1 Site core

The SiteCore is the main component for each site; its main responsibilities are initializing the other core components, maintaining a reference to these components, loading configuration and possibly defining site behaviour. The SiteCore bootstraps a site.

We have defined this component to provide a central point for other (core and non-core) components to get a reference to the core components. This way it is possible to design a component that can be used at multiple sites in the system, without site specific dependencies. To achieve this single point of access, we define it as being an abstract singleton class that initializes the core components in its constructor [GAM94]. Each site must implement its own site core; we define the site core abstract to enforce this.

4.4.2 Component repository

As stated in the introduction of this chapter, we will use a component based design approach; within the system, we define several components, all with their own tasks. For component interaction and manageability, we have defined a generic component-interface that all components must implement. All these components must register themselves at a component repository, to allow other components to locate them.

An address consists of two parts, the site identifier and a component identifier, where the first identifies what site the component is located (MBU, surrogate, backend) and the second a unique component-id within that site. We have specified this in a *ComponentLocation*; so if an alternative way of addressing is implemented, this is the only entity that needs replacement. To ensure a unique address, addresses must be assigned by the repository.

We have illustrated our component design in Figure 4-14. The *Component*-interface shows the methods that each component must support; indirectly, it indicates that a component is of a specific *ComponentType* and has a component location. A component (C-A) that depends on another component (C-B) can use the repository's *get*- and *findComponent*-operations to obtain a reference to C-B, this allows direct component interaction between multiple non-core components using method calls instead of the event bus.

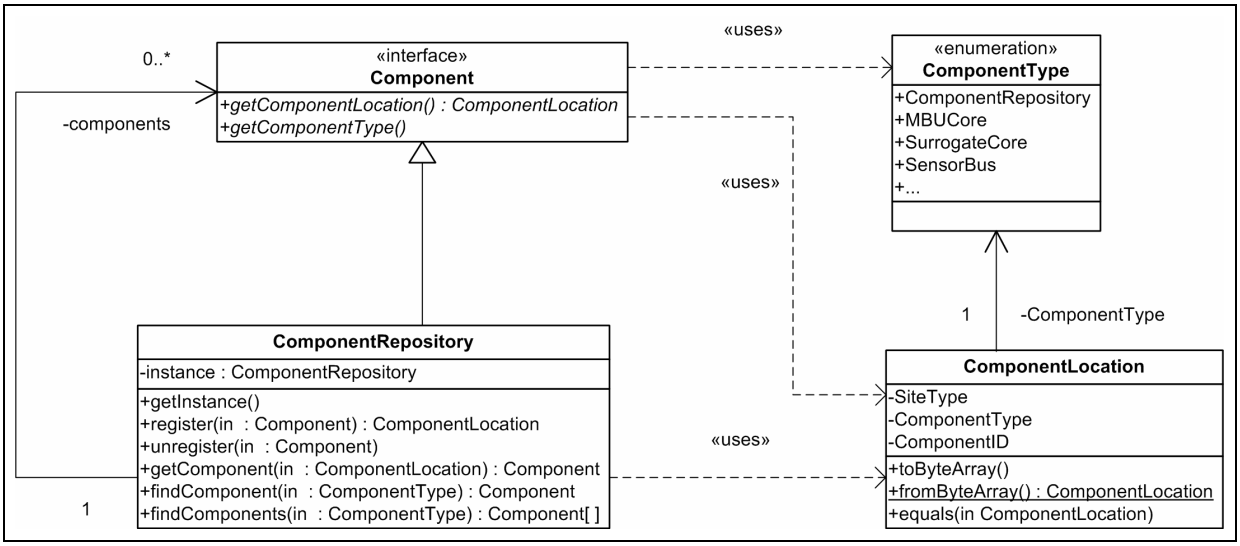


Figure 4-14 Component Repository design

If remote component interaction is required between non-core components, for example for adding chat functionality, remote discovery is also required. The repository may listen to a discovery broadcast event for a specific component type, check if it contains a reference to such a component and then send a message to the component that initiated the discovery, supplying the ComponentLocation of the discovered component.

4.4.3 Sensor bus

The sensor bus is a central location where all data producers can publish sensor data. The device drivers are the most likely components that produce sensor data, but also components as a signal processor can publish (derived) sensor data. We have explained in the section on the BAN model that we will process sensor data per channel; where a channel consists of several sensors that run at the same frequency. The sensor bus provides the functionality to transfer the data, as explained in section 4.3.2.

Figure 4-15 shows the structure for the sensor bus; the *SensorBus* is the central point of access for sensor-data and allows registration of a channel and subscribing for a channel. The bus keeps track of the registered channels in a *SensorConfiguration* object. For each channel that a data producer registers, the bus creates a *ChannelOutputStream* into which the producer can push the channel data. This output-stream creates a *ChannelInputStream* for each data consumer that subscribe at the sensor bus, the consumer uses this input-stream to pull channel data for the channel. The bus uses the *ChannelDescriptions* to identify the different channels and their corresponding out- and inputstreams.

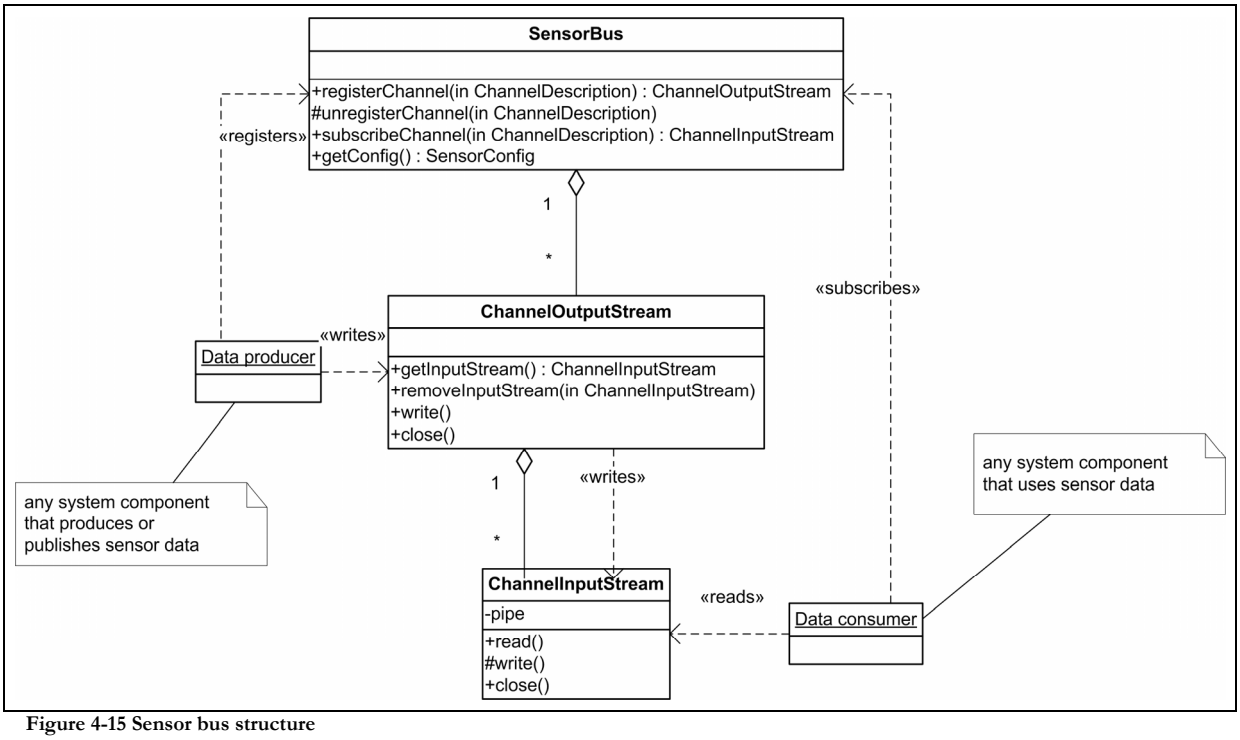


Figure 4-15 Sensor bus structure

The generic operation of the sensor bus is illustrated in a sequence diagram in Figure 4-16; a data producer registers a channel at the sensor bus by calling the *registerChannel* operation with the channel description, the bus then creates a *ChannelOutputStream* and adds the description to its configuration, the output stream is returned to the data producer. The data producer can now start writing data into this stream, which, at first, will be discarded, because there are no listeners. A data consumer can now subscribe for data at the sensor bus, but first it requests

the configuration and determines for which channel he wants to subscribe. Then he calls the *subscribeChannel* operation, with the channel description; the sensor bus will ask the *ChannelOutputStream* to create a new *ChannelInputStream* that can be used by the consumer. If the producer now *write*s sensor data into the outputstream, it will *write* this data into the inputstream. The inputstream puts the data into its internal pipe (not shown), until a data consumer reads this data by calling the *read* operation.

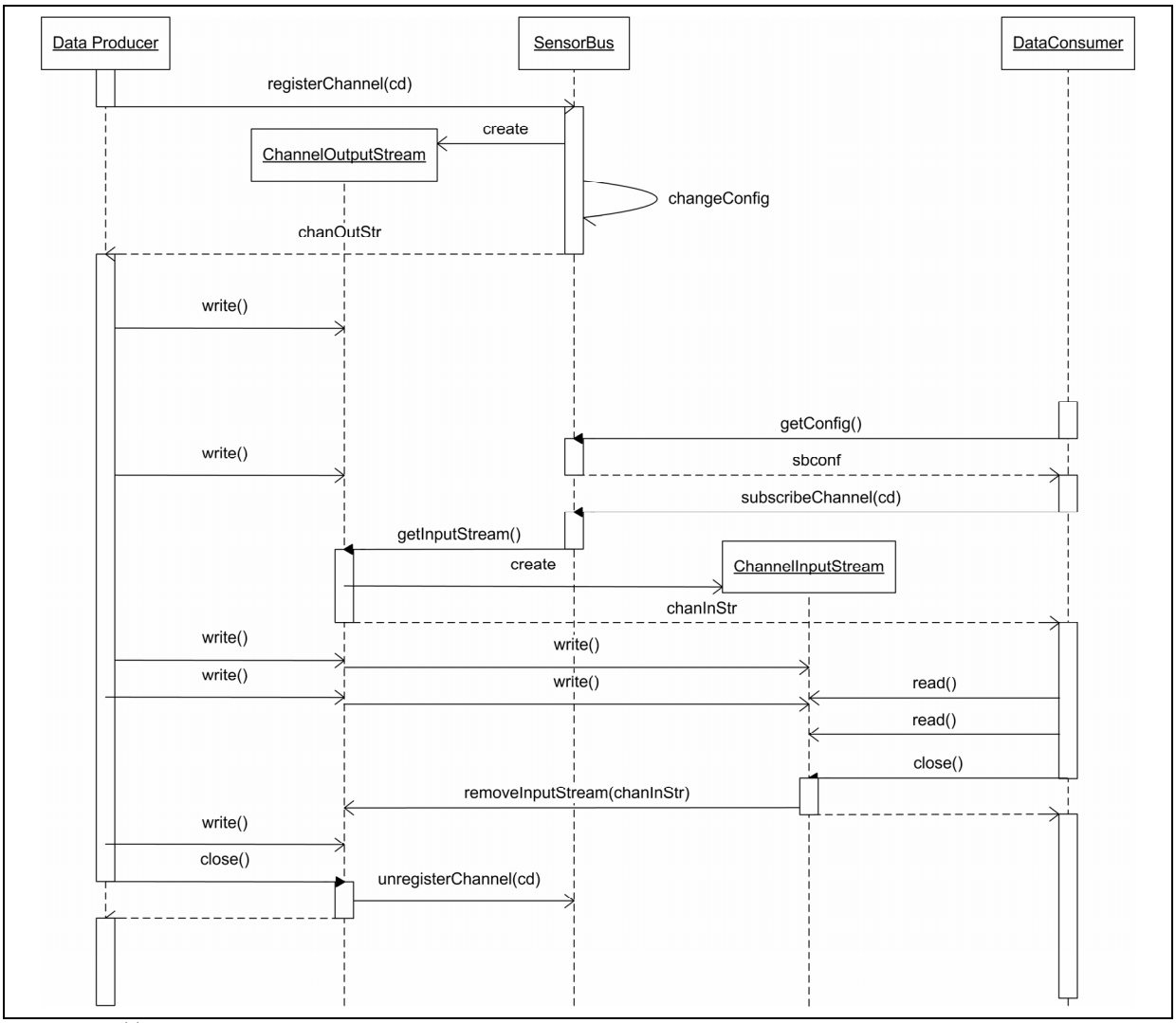


Figure 4-16 Time sequence diagram for sensor bus operation

The sensor bus uses the event bus to broadcast events when new channels are added, if channels are closed or if an exception occurs.

There are several issues that must be considered when implementing the sensor bus:

- **Full pipes:** The pipes are used to decouple the data producer from the data consumer; if the consumer does not process the data fast enough it is possible its pipe is full. If a pipe is full, it can either block or throw an exception and close the pipe. The producer cannot block, because it might lock up the entire data supply and an exception results in loss of data at consumer side. Both are undesirable; however, because blocking behaviour is hard to detect, we choose for a non-blocking pipe. This adds a responsibility to the data consumer that he has to make sure he to read input fast enough, if data loss is not tolerated. The sensor bus may publish an event indicating that a pipe has filled.

- **Pipe implementation:** Since pipes deal with two threads, some kind of synchronization must be implemented. There are several strategies for implementing this synchronization, which can affect both speed and system resources. We will leave examining these strategies for implementation, since this is platform and programming language dependent.
- **Discarded data:** Data that is sent into the sensor bus before any listener is connected will be discarded; because of the continuous nature of the data in the sensor bus, this is not considered a problem.
- **Sample-start synchronization:** Producers can write data per byte, while a sensor sample can consist of multiple bytes; the bus must ensure that the first byte being read by a consumer is the start of a sample.

4.4.4 Event bus

A common approach for event distribution is to create a class for each event possible and a listener for that specific event; however, listener methods must be implemented for each type of event a component listens to and components have to subscribe directly at the publishing component or a mediator is used to couple the event producer and the listener. For our platform, we use a slightly different approach, where we have a generic listener type and a generic event, because this allows us to add new event types more easily and event listeners will not need to implement a method for each possible event they will receive. The type field will indicate what the event actually is; therefore we need to define a list of unique event-types, which should at a shared location such that all components in the system can access it. We use a mediator, which is called the event bus that sends all events to the subscribed listeners. There can be only one event bus for each site (MBU, surrogate, backend). Figure 4-17 shows the UML diagram for the event bus.

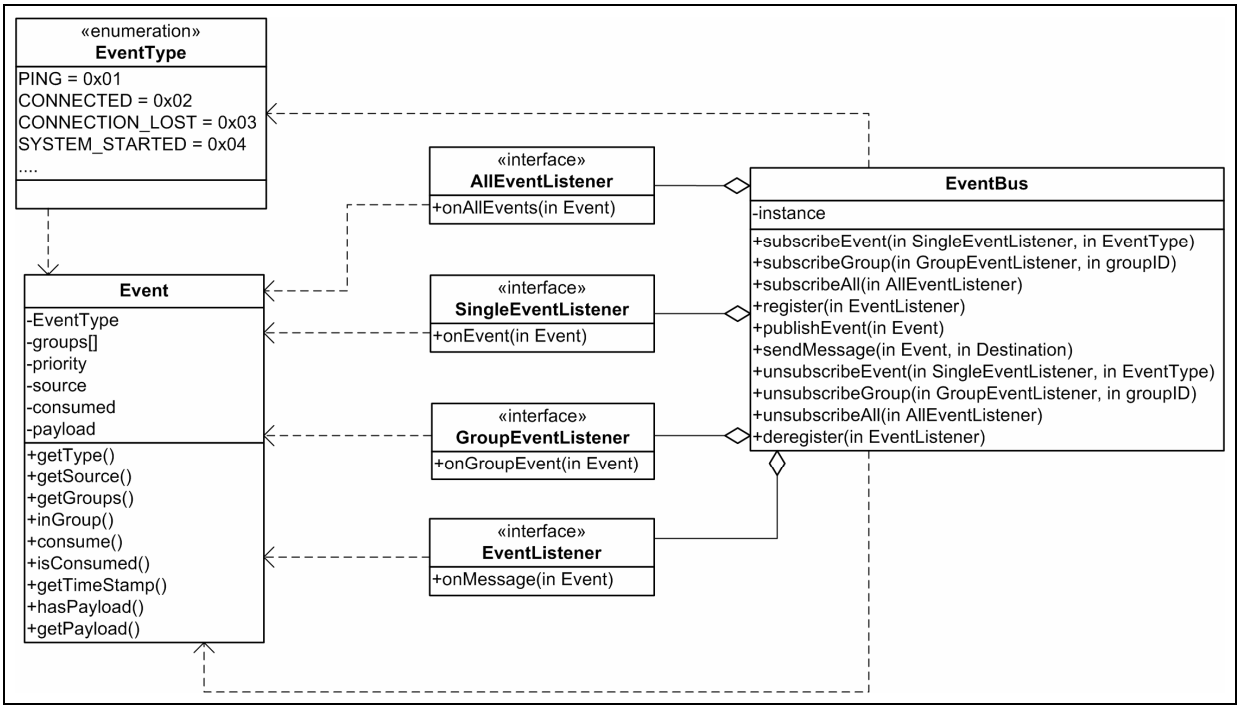


Figure 4-17 Event bus structure

The diagram reveals that we also implement methods to publish a unicast event, which requires a destination in addition to the event. Depending on what kind of events they subscribe for, components must implement the Listener-interfaces; for example, if they want to receive messages, they must implement *MessageListener*.

The event bus supports subscribing for single events (*subscribeEvent*) and for all events (*subscribeAll*),; components that want to receive private messages must register themselves at the event bus (*register*). Some components may register for all private messages (*registerAll*), which should be solely used for log or relay purposes, because messages are not meant to be used as a broadcast mechanism.

Figure 4-18 shows a sequence diagram for a simple ping event. Component A registers itself at the event bus for ping-events by issuing a *subscribeEvent* call. When a ping-event occurs at Component B; it will publish this at the event-bus using *publishEvent*. The bus will then relay this event to all listeners, currently only Component A, by issuing the *onEvent* call. When all listeners have processed the call, the bus returns control to Component B.

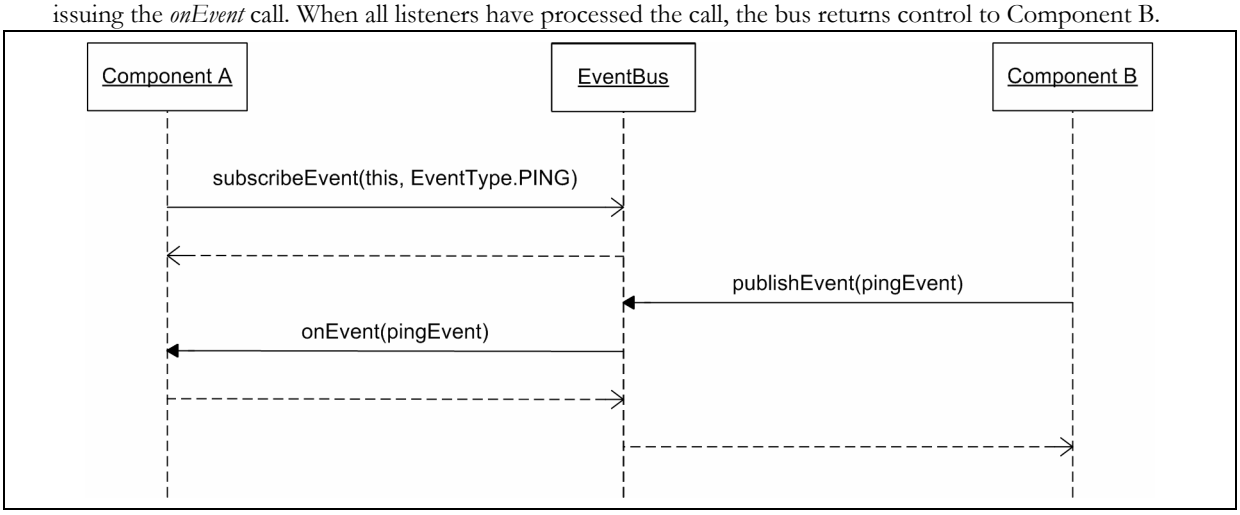


Figure 4-18 Sequence diagram for event subscription and publication

This sequence diagram shows that a *publishEvent* call will block for as long as the event is not processed; this could be including the relaying the event to another site. Therefore we state that event listeners should put the event on an internal stack if the event processing can take too long; however, we can not enforce this behaviour, possibly blocking the event publisher for too long. We have not identified any situation where this could pose a serious problem for the system and have not specifically designed a solution for this problem. If it proves to be a problem, we suggest adding an event-processor to the bus, which publishes events to the listeners in a separate working thread (or thread-pool). With this approach, Component B's *publishEvent* call will immediately return, and after that, Component A will receive its *onEvent* call.

4.4.5 Data block manager

The data block manager is responsible for encoding and decoding data blocks into the format that the lower level communication layer (see Section 4.5) uses and receiving data from the communication layer and relaying these to the correct component. If the communication layer does not support large blocks, the data block manager is responsible for splitting up a block into multiple parts and reconstructing incoming parts into a data block. We will not elaborate on this, but leave this as future design work.

When a data block is reconstructed from the lower layer, the manager uses the component repository to locate the destination component and then pushes the data block to this component; if the component is not found, a message will be sent to the sending component, indicating the failure; same goes for a component that is not capable of processing data blocks.

Figure 4-19 shows our design for a data block manager; the *DataBlockManager* is a component that supports receiving blocks from local components, and receiving data PDUs (see section 4.5.2) from the lower level. Components that can process data blocks must implement the DataBlockReceiver interface and the DataBlockManager will push the blocks using the *pushDataBlock* operation. Furthermore, the DataBlockManager depends on the component repository and the communicator.

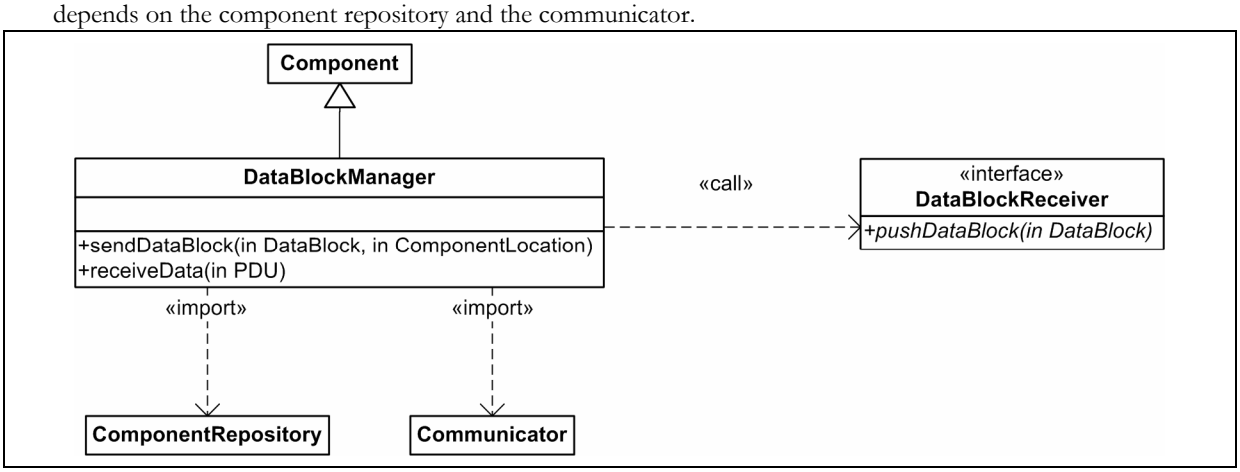


Figure 4-19 Data block manager design

4.5 Communication

This section describes a generic communication approach that simplifies data transmission and allows easier integration with new technologies. We start with the framework description and structure and then discuss how this framework can be used for transferring our data types and what must be implemented to support relaying of event and sensor data.

4.5.1 Connection framework

There are at least two interaction points between the e-Health platform and a lower level service provider (LLS); the first being between the MBU and the surrogate and the second being between the surrogate and the e-Health client. Moreover, it is likely that the lower level service provider changes, either because new technologies are available or because the connection method changes. Therefore we introduce a connection framework that offers a standard API for connectivity to a lower level service provider, which is shown in Figure 4-20.

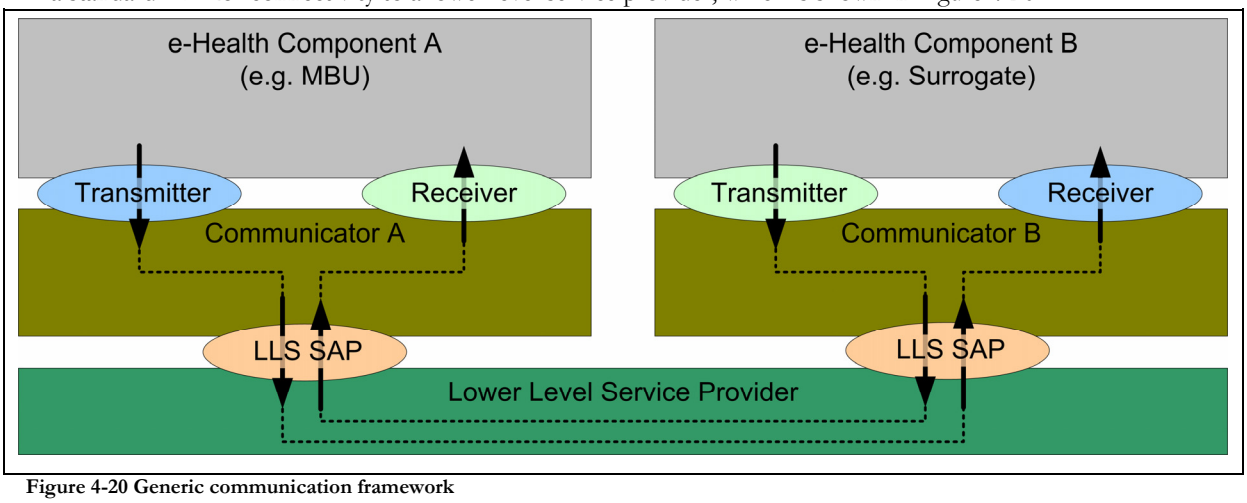


Figure 4-20 Generic communication framework

Each LLS requires implementation of a communicator (i), which is responsible for connection management; a transmitter (ii), which translates outgoing data onto the LLS and a receiving entity (iii) that can deliver the data to a (generic) receiver in the e-Health component. When a new LLS must be supported, it is only necessary to write the above three components, without requiring any change within the e-Health platform component at either side of the LLS.

4.5.2 Framework design

Figure 4-21 gives the UML diagram for the communication framework and shows that a communicator consists of a transmitter and a receiver. Important to mention is that the implementation of the receiver is generic and can be re-used for each communicator. The communicator, however, should implement functionality to receive calls from the LLS and forward these to the receiver using the calls shown in the diagram.

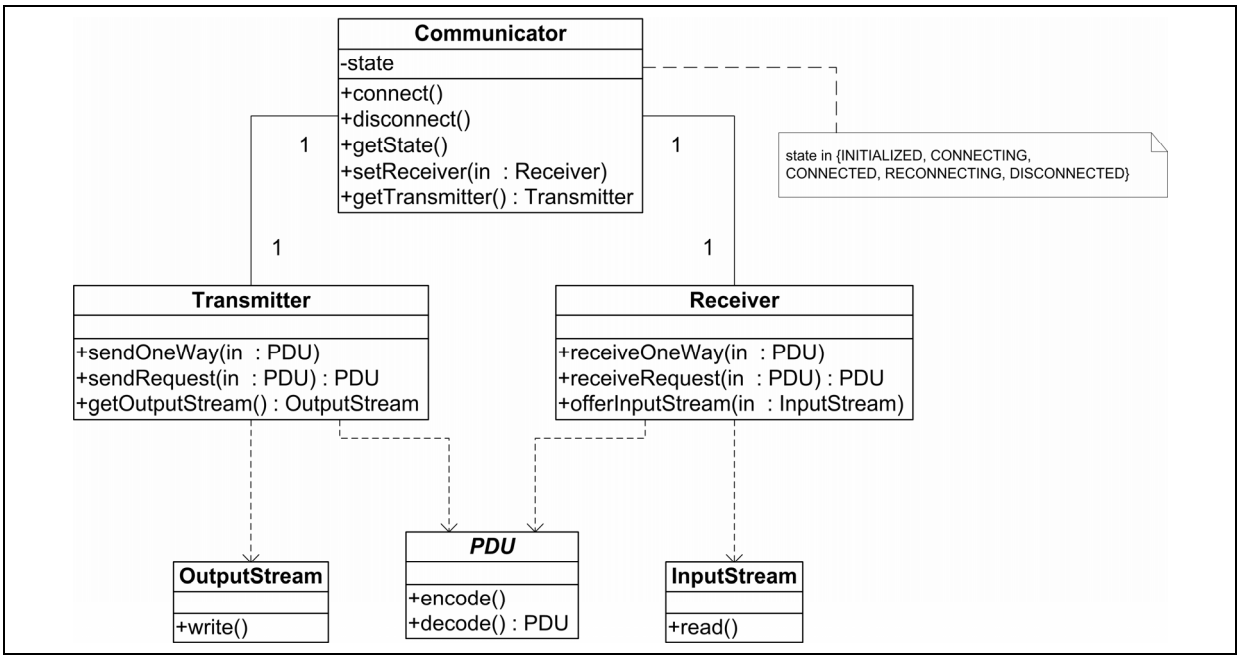


Figure 4-21 UML diagram for the communication framework

The figure shows the following components:

- **Communicator:** The communicator is the main component within the framework; it takes care of connecting to the LLS (not shown in the diagram) and keeping the state of this connection. It consists of a transmitter and a receiver. The communicator is responsible for receiving calls from the LLS and then forwarding these to the receiver;
- **Transmitter:** The transmitter is the bridge between the e-Health component and the LLS, it translates calls from the component onto the LLS API, possibly performing additional steps like encoding, compressing and multiplexing the data;
- **Receiver:** The receiver is a generic component that is available in the e-Health component; it is responsible for handling the incoming data, for example by decoding the PDU (see below) or relaying the stream to the responsible component;
- **PDU:** A PDU (Protocol Data Unit) is the type of data that can be exchanged between components. Different PDUs can be defined for the system, but they must all be encoding using a generic encoding scheme, such that the communicator can transmit them properly over the LLS, without the need to rewrite

these PDUs for each LLS. An example encoding scheme that can be used is XDR encoding [RFC1014]. The communication manager should be able to determine a maximum PDU size.

- **OutputStream:** An outputstream is a serial byte-stream that can be used for many purposes, all data written into this outputstream, will arrive at the other side's inputstream in the same order. Data can be written per byte, or per block of bytes, using a write-operation;
- **InputStream:** The inputstream is the peer of the outputstream, bytes arrive in the inputstream as they were written at the other side and can be processed by the receiver by using the read-operation.

All calls at a transmitter at one site will result in the corresponding receiver call at the other site.

4.5.3 Event communication

Since our event bus only distributes the events and messages locally, the system requires a component that relays the events using the communication framework. To save bandwidth, it is important that not all events are relayed, but only the events that are flagged to be relayed. Therefore we need some flagging method allowing components to indicate remote broadcasting of an event. We could opt for adding this flag in the method call *publishEvent* at the event bus, or we can add it as an attribute to the event. Because the destination might be necessary at the other site, we have chosen to add this as an attribute (*siteDestination*) to the broadcast-event. To simplify setting the broadcast-range of an event, we will add three operations to the event-type, namely *toMBU*, *toSurrogate* and *toBackend*.

The relaying component (*EventRelayService*) subscribes for all events and all messages and analyses the destination for these events; for generic events, it check the site destination, for messages it check the site-part of the component location. If the event or message is not local; the system will encode the event onto a PDU and then transmits this PDU using the transmitter of the framework. Since events are always unidirectional, we can use the sendOneWay operation of the transmitter.

4.5.4 Sensor data communication

Sensor data communication is the most important form of communication for the system, especially the communication between the MBU and the surrogate, because this is the bottleneck communication line for our system. We must try to optimize bandwidth usage to reduce cost and to allow sending all required vital signs (Requirement #2.1). Besides optimization of the transmission protocol using compression and packetizing [BUL05], [WAC05], one of the best ways to save bandwidth is to disable transmission of sensor data that is not required at the backend, or in other words, only enable those sensors that are required at the backend. Section 5.6.3 describes the *SensorRelayService* that implements this behaviour. This service uses the outputstream in the transmitter to send the sensor data to the surrogate and defines its own protocol that efficiently transmits the sensor data.

Surrogate to backend communication should be implemented in the exported e-health service and depends on the backend application requirements. Often, there are no bandwidth constraints for this communication, so optimization is not always necessary. Section 6.4 elaborates on this subject and describes several possible approaches.

4.6 Conclusions

This chapter described the top-level design of the mobile e-health platform using the Mobile Service Platform and JINI architecture as a basis; the platform consists of several sites: the MBU, the surrogate and the backend clients, which all are decomposed in their main components. We have identified several common components, which are defined in terms of their overall structure and their interface; internal component design is left for future work. The chapter also discussed the use of the BAN model, and some approaches to internal data representation. The last part of this chapter described a generic approach for communication that can also be reused at the different sites.

The presented design does not incorporate much of the MobiHealth implementation, except for the usage of the Mobile Service Platform and the JINI architecture. Most of the useful MobiHealth components deal with the sensor data, which is propagated internally using a (buffered) push-mechanism. This push-mechanism allowed components to delay each other, which is undesirable; therefore we changed this into a push/pull mechanism that resulted in our sensor bus design and sensor channel approach. Changing this approach had one major disadvantage: not much of the MobiHealth design could be re-used.

We have tried to focus on platform flexibility: main functionality is split up into components and the sensor bus, event bus and communications framework provide a central location to these components for interacting with other components at their own site, as well as at other sites. The redefined BAN model provides a conceptual understanding of how sensors are internally structured and transmitted and provides a configuration notation.

5 Mobile Base Unit

The Mobile Base Unit (MBU) is the central processing unit within the Body Area Network (BAN); its main responsibility is to gather sensor data from the sensor devices and relay this to the back-end of the system where the data can be processed or sent to the screen of the healthcare professional. Due to the limited amount of bandwidth available, this relaying must be as efficient as possible. The MBU consists of a Personal Digital Assistant (PDA) that is equipped with means to communicate with sensor devices and is capable of using 2.5/3G connection to the Internet and a software package that implements MBU behaviour and uses these capabilities to acquire and relay sensor data. For the remainder of this thesis, we will use the term MBU to refer to this software package, in the MobiHealth implementation, this part is also known as the *banware*.

Section 4.1.4 shows a decomposition of the MBU into its main components; this chapter zooms in on all those components and their sub-components and explains their purpose and design, taking the requirements presented in chapter 3 into account. The design leaves some gaps for future work, especially on some more complex subjects as connection management; however, using the design presented, it is possible to implement a simple version of the MBU.

The first part of this chapter describes the core components in the MBU and how they interact; the second part shows how device connectivity is handled and the third part discusses a framework for plug-ins that provide additional functionality. The fourth section provides information on the design of a graphical user interface for the MBU and the fifth section discusses MBU behaviour and operation. The sixth section elaborates on the communication between the MBU and the surrogate and the chapter ends with some preliminary conclusions.

5.1 MBU Core

The MBU core consists of the components that are necessary for the core functionality of the system and these are therefore required for almost every scenario. This section discusses these main components and their responsibilities. Figure 5-1 shows how these components are related; the arrows indicate the event flow and sensor data flow and also the way components are instantiated and managed. The sensor bus and event bus both have a prominent position in the core, since they provide a backbone for component and sensor data communication; the component repository is not shown, although it is available. The figure illustrates that the Graphical User Interface, the Plug-ins, the Device Drivers and the Backend Communication are all separated from the core, but can all interact with the event bus and sensor bus; giving them access to all relevant system information. These parts are the flexible components of the MBU and therefore core components do not have knowledge of their internals and must use the event and sensor bus to communicate with these components.

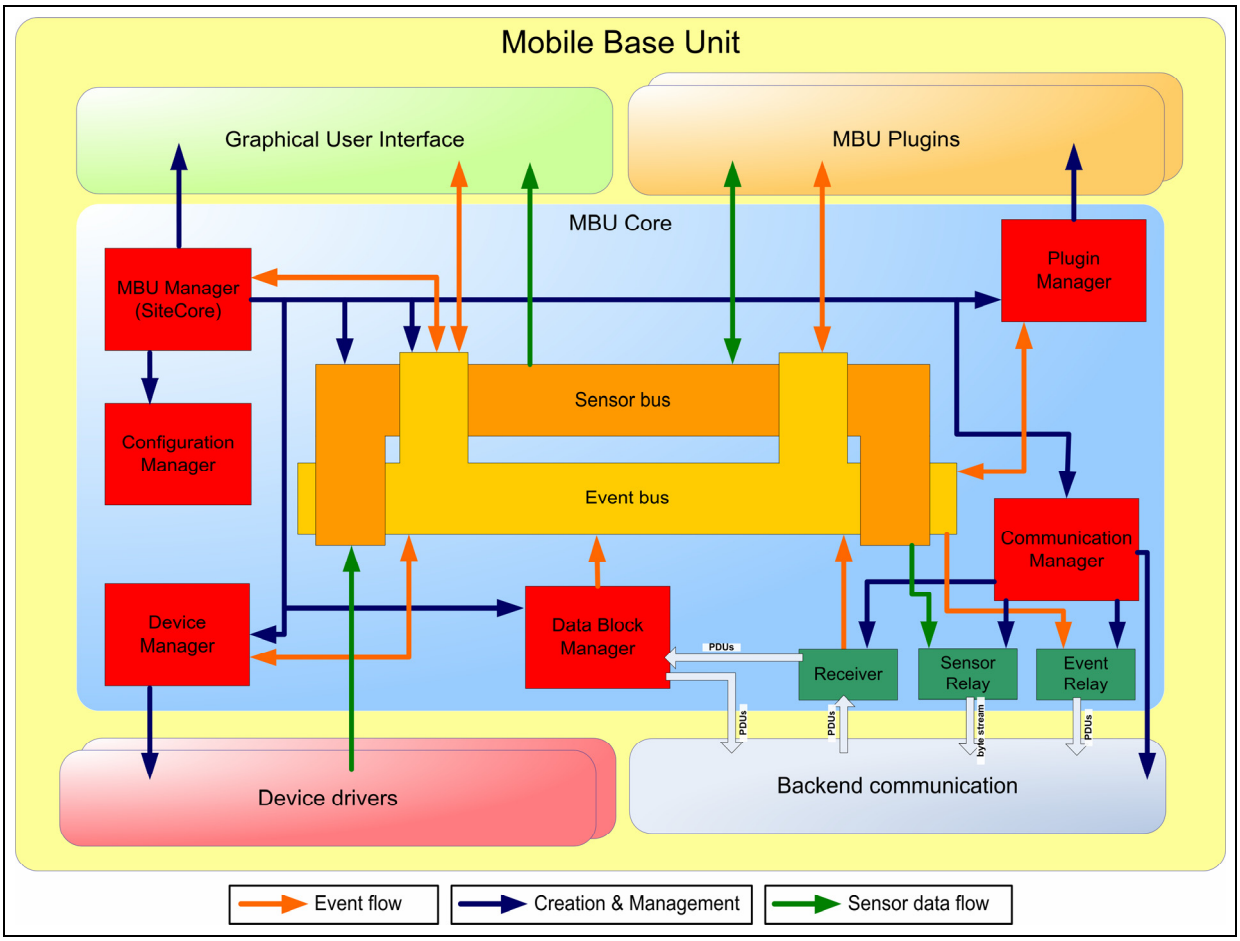


Figure 5-1 MBU components and interaction

5.1.1 MBU Manager

The MBU Manager is the SiteCore for the MBU; therefore it provides the functionality as defined in section 4.4.1 and contains a reference to all the core components. In addition, it is responsible for creating and managing the MBU specific core components, which are the *DeviceManager*, the *PluginManager*, the *ConfigurationManager* and the *CommunicationManager*..

Besides creating and managing other components, MBU behaviour should be defined in the MBU manager. While other components listen to events that are related to their functionality; the MBU manager listens to all events and generates new events or calls methods based on the combination of those events. Behaviour after an event may depend on previous events; therefore we suggest implementing a Finite State Machine (FSM) using the State pattern in combination with the Flyweight pattern as described in [GAM94] to define system behaviour in the system. Section 5.5.2 describes basic system behaviour in terms of states and events. More complex scenarios may require more logic in the MBU manager, like a context-aware reasoning machine and a more simple approach to defining system behaviour, but this is left for future research.

Based on incoming events, the MBU manager will send commands to other components; for MBU components there are two ways:

- **Method calls:** Since the MBU manager contains a direct reference to all core components, it can directly call all public methods in these components.

- **Events:** The MBU manager can send a pre-defined event (broadcast or directed) to components that define a specific component action.

Both methods have their own pro's and cons; using method calls, the MBU manager can receive a return value and use that to determine further action and it produces less overhead. Using events, the MBU manager can broadcast its command, allowing possible plug-in components that deal with the same kind of functionality to also process the event. However, possible return values must be returned by a unicast event, which is not very efficient. We suggest using a combination of both ways, depending on the type of command, its importance and reliance on return value. For remote components only events can be used.

5.1.2 Configuration manager

The configuration manager is a central location where components can request specific parts of the system configuration; the manager is responsible for loading the configuration from a file, offering a central repository for configuration objects and writing the configuration to a file. Based on experiences with the MobiHealth system, XML seems to be ideal for defining the configuration. XML configuration-nodes can be parsed either by the configuration manager or by the component that requires that configuration if its format is unknown. For our design, we will limit ourselves to a configuration manager that will parse a configuration file and offers an interface for components to get their own configuration element that they should parse.

A snapshot of a possible configuration file is shown in Table 5-1. For example, the patient information is a core configuration node; therefore the configuration manager supports parsing this node into a patient-object (see Figure 5-2). The communication node is also in a known format (it consists of communicators); however, the communicator nodes only have certain known attributes; their child-nodes depend on the related communicator, and thus must be parsed by this communicator. Note that the 'raw' communicator is just added for illustration, we do not foresee designing or implementing a raw socket communicator in the scope of this thesis.

```
<!-- Patient information -->
<Patient>
  <ID>MST-10-1000</ID>
  <Name>John Doe</Name>
  <Phone>06-12345678</Phone>

  <!-- Patient's physician -->
  <Physician>
    <Name>Dr. James Johnson</Name>
    <Hospital>Medical Center Townsville</Hospital>
    <License>1234</License>
  </Physician>
</Patient>

<!-- Communication configuration -->
<Communication>
  <Communicator name="default" type="msp" priority="1">
     <JiniSH>jinish://10.0.0.52</JiniSH>
     <SurrogateJar>http://10.0.0.52:8081/MBUsurrogate.jar</SurrogateJar>
  </Communicator>

  <Communicator name="backup" type="raw" priority="5">
     <ServerIP>10.1.1.10</ServerIP>
     <Port>1234</Port>
  </Communicator>
</Communication>
```

Table 5-1 XML configuration snapshot

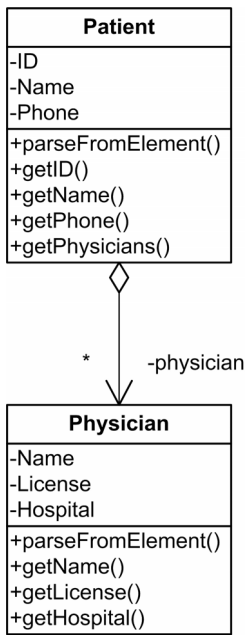


Figure 5-2 Patient UML

The exact notation for the configuration will be defined in the implementation phase; preferably by defining an XML-schema for each configuration element. The notation for each specific communicator, plug-in and device driver should be defined by the implementer of these components.

Since the surrogate is a transient object, the surrogate configuration should be stored at the MBU. The communication manager is responsible for reading the surrogate configuration and allowing the communicator to retrieve and transmit this to the surrogate.

A complete example configuration is shown in Appendix A.

5.1.3 Communication Manager

The communication manager is responsible for establishing a connection to the surrogate and reconnecting if the connection is lost. Using the generic communication framework that was discussed in the previous chapter, the communication manager loads a communicator that is responsible for communication to the surrogate. The type of the communicator and its related settings are defined in the configuration. The communication manager is also responsible for setting the receiver for the communicator, allowing incoming data to be processed.

For the basic platform, only a single communicator type is supported. The communication manager can also be used to define a context-aware mobility manager that detects available networks and automatically selects the best network for the data. This is a research area that is being working on within the Awareness project and is left open for future work on the platform [PED05]. The generic framework presented in the previous chapter simplifies implementation and allows adaptive networking.

Section 5.6 elaborates on the design of a communicator based on the Mobile Service Platform and how the generic framework is used in the MBU.

5.2 Device drivers

Device drivers are responsible for gathering sensor data from external sources; most likely these are medical devices, but possibly also devices that collect context information, like a GPS receiver for location services. The driver pushes this sensor data into the system, using the sensor bus. This section describes the device manager and the generic design for these device drivers.

5.2.1 Device manager

The device manager loads and manages the device drivers; its main task is loading the device drivers after reading the configuration and managing them by giving the appropriate commands after a certain event. The complexity of the scenario defines how and when these commands are given; for example, for a basic system, the device drivers are told to connect immediately at start-up, while more complex scenarios require more control over this behaviour.

The device manager also maintains a sensor configuration for the devices connected; this configuration defines what sensors are available at the device using the device configuration as defined in section 4.2.4. The channels in

this configuration do need not to be active in the sensor bus; but can be used for reference at the backend side. In the future, this reference may be used to remotely enable/disable sensors at the devices; for now, we suffice with simply not using sensor data of those 'disabled' sensors; however, if energy consumption of a device may benefit from this or if putting unused sensors in the system is a burden to system performance, it could be implemented.

5.2.2 Generic device driver design

For system flexibility, all device drivers must follow a generic design and follow a few behavioural rules. The design is defined by the DeviceDriver interface in Figure 5-3 and it is necessary to write a device driver for all device types that will be connected to the platform. The interface extends the Component interface and adds seven operations:

- **connect:** The driver connects to the external device and keep this connection alive; if any sensor data is sent to the device, it should be discarded. In most cases, device drivers will run in their own thread to read the sensor data, this thread may be started here to allow processing of keep-alive messages;
- **disconnect:** Requests the driver to disconnect from the device and return to its initial state;
- **registerSensors:** The device driver registers the channels that are enabled at the sensors bus, opening the sensor channels for data output. Having this operation allows other components to subscribe to the sensors before the stream is actually started, which is not strictly necessary.
- **startStream:** The driver starts relaying data from the device and puts this on the sensor channels that were obtained in registerSensors;
- **stopStream:** The driver stops relaying the data into the sensor bus and unregisters its sensors at the sensor bus, while the connection to the device is kept alive;
- **getDescription:** Get the DeviceDescription for this DeviceDriver; the DeviceDescription contains ChannelsDescriptions and these contain SensorDescriptions, as explained in the section on the BAN model. This description must be generated at device driver initialization;
- **getState:** Get the state for this driver (see below).

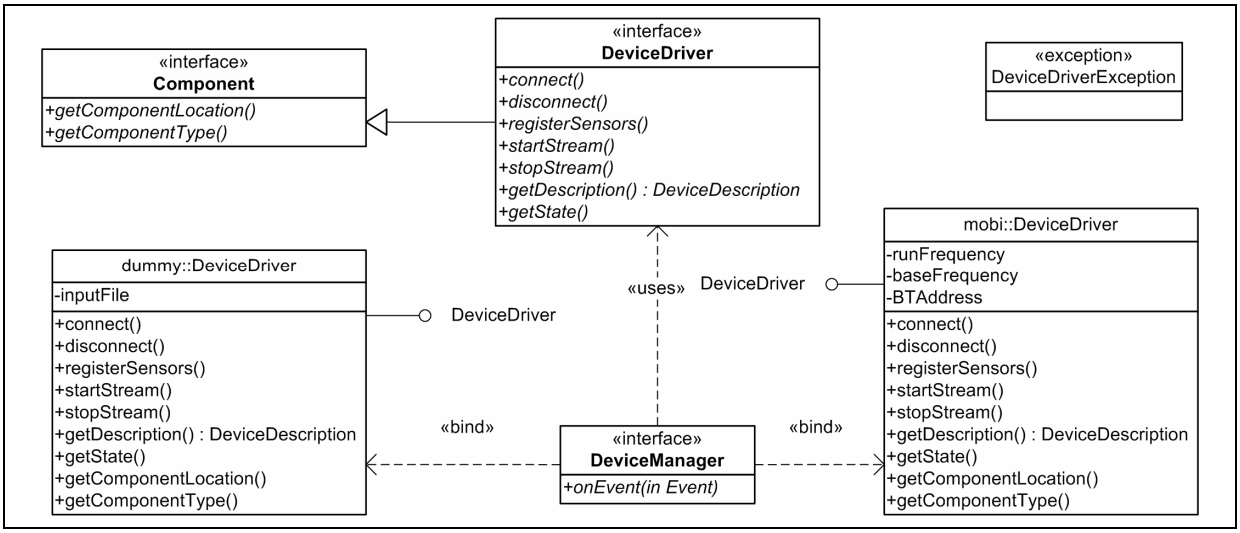


Figure 5-3 Device driver structure

The device manager supplies the device's XML-configuration to the device driver upon loading; the device driver itself is responsible for parsing this configuration and interpreting the data. Since the device driver implements the Component interface, it must register itself at the component registry and can use the same core functionality as any component. If it implements an event-listener, it can also subscribe for events or messaging.

Figure 5-4 shows the four states of the device driver and what operations are possible within those states. All device drivers must implement at least this behaviour and throw a *DeviceDriverException* if an operation is called that is not supported in that state. The *getState* operation is provided to prevent exceptions from being thrown. The driver will never return to a registered state from a streaming state, because subscribers (via the sensor bus) may then experience an inconsistency in the data stream.

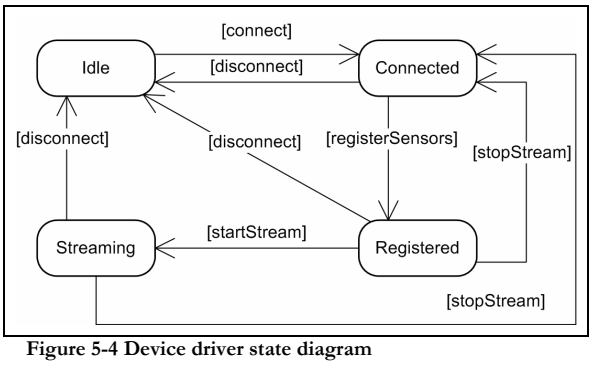


Figure 5-4 Device driver state diagram

5.2.3 Device specific design

The TMSi Mobi device is the only device that is currently being used in the Awareness project; the current system supports this device as well as a dummy device driver that reads stored MobiHealth sensor data from a file, which is mainly for testing purposes. This section points out some common issues in device driver design using the Mobi as a reference.

Most medical devices will be connected using wireless technologies, like Bluetooth. Unfortunately, using Bluetooth requires a platform dependent functionality and can not be design platform independent. The JSR-82 specification aims at a defining a standard Java-API for Bluetooth drivers [JSR82]; however, these are not yet commonly available for Windows Mobile PDAs. Yucat is currently developing a JSR-82 compatible Bluetooth library and has already developed a Bluetooth library with a proprietary API for communicating with the Mobi, which is called the BTStreamer. The current BTStreamer does not support device discovery and offers only limited functionality; the new JSR-81 compliant library will support it.

A device driver for the Mobi using the BTStreamer can connect to a predefined device address and will automatically deal with the TMSi fibre protocol; this results in a stream of sensor data that can be sent into the sensor bus channels directly. More advanced device drivers may add possible device discovery, use the capabilities of the Mobi to automatically detect the sensors that are connected and/or offer dynamic channel configuration.

5.3 Plug-in components

Plug-ins are components that are not required for the core system, but provide additional functionality, which is required for many complex scenarios. Plug-ins can be used for digital signal processing, sensor data analysis or other sensor data related operations, but they can also be used for providing file transfer capabilities, patient diaries and remote administration. It is also possible that plug-ins generate additional sensor data, a digital signal

processing component may derive a heart rate from ECG and publish that heart rate as an additional channel; these sensors are called the derived sensors.

5.3.1 Plug-in framework

The plug-in framework is inspired by the Component Configurator pattern [SSR00] and aims at flexible loading and unloading of components. There are many patterns for a plug-in framework, some allowing even a 'hot-replace', where the component is updated at runtime [KON04], but this functionality is not (yet) required for the mobile e-health platform. The platform uses a simplified version of the pattern, where we have integrated the component configurator and the component repository into the plug-in manager. The plug-in manager keeps track of the plug-ins and loads and unloads them.

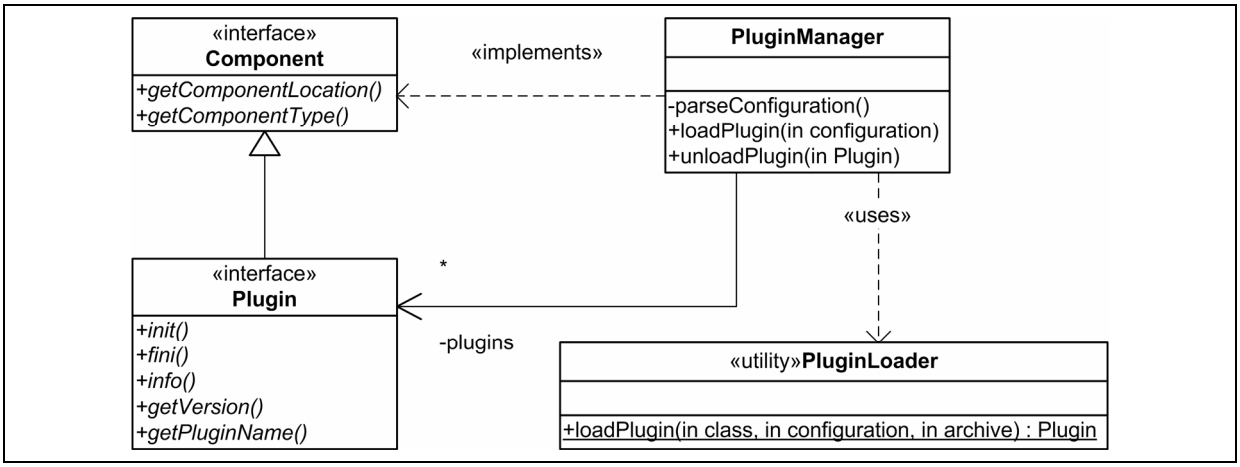


Figure 5-5 Plug-in framework

Figure 5-5 shows the plug-in framework; consisting of the following components:

- **Component:** All plug-ins are a component and thus can use all other core components and core functionality as defined in the previous sections; plug-ins are also of a component type and have a location;
- **Plugin:** This is the interface that all plug-ins should implement; it contains the *init* and *fini* operations that are used to respectively start and stop the plug-in. Suspending and resuming is not yet supported; but might be added in the future;
- **PluginManager:** The plug-in manager is the component repository and component configurator in terms of the Component Configuration pattern and is explained in further detail below;
- **PluginLoader:** The plug-in loader contains a method to dynamically load a plug-in object; either from the archive supplied or from the MBU-archives.

The plug-in framework is very generic and can be reused at the surrogate and backend application; however, there is currently no need for plug-ins at the surrogate and backend design is left for future implementation. If additional functionality is required at the surrogate, this should be implemented in the corresponding JINI service that is implemented.

Since all plug-ins are also components, they must register themselves at the component repository at start-up; if there are any inter-plug-in dependencies, it is desirable that a plug-in announces its initialization by broadcasting a certain event; allowing the other plug-in to retrieve its reference from the component repository using the event-source.

5.3.2 Plug-in manager

The plug-in manager is responsible for loading plug-ins and enabling and disabling them. At start-up, it will load all plug-ins that are specified to be active in the configuration; other plug-ins can be loaded later, for example when the manager receives a specific event; or by an explicit 'load plug-in' call. To increase runtime flexibility, we define that plug-ins can be loaded from an external archive file, opening a door for remote plug-in installation. These parameters should be defined in the configuration object that is given as an argument to the *loadPlugin*-method.

Since the plug-in manager is one of the core-components in the MBU any other component, including the plug-ins can access the plug-in manager and ask to load or unload a plug-in. It might be necessary to introduce some security features in the plug-in manager to prevent malicious components from loading even more harmful plug-ins.

5.3.3 Example plug-ins

This section describes several plug-ins that are likely to be implemented for various scenarios; these will not be implemented in the scope of this thesis, but illustrate the diversity of the plug-ins.

- **Local data storage:** The local data storage plug-in writes sensor data to local storage, allowing this data to be sent to the backend later as a data block. This is an important plug-in for research trials where no data may be lost at all;
- **Sensor data buffer:** The sensor data buffer stores the last few minutes of sensor data, allowing this to be sent to the backend on request or after a specific event;
- **Signal processing:** A signal processing plug-in analyses sensor data and produces derived sensors from this data, for example deriving a heart rate from an ECG signal. This plug-in is very important for the Awareness epileptic scenario;
- **Data analysing:** The data analyser reads sensor data and produces events based on this sensor data, for example, sound an alarm when heart rate drops below 40; it is a simplified version of a signal processor, because it does not produce derived sensors itself;
- **MBU Administration:** An MBU administration plug-in can be used to export local MBU functionality to the backend, allowing the MBU to be administered remotely.

Many others plug-ins are conceivable and the possibilities with plug-ins are virtually unlimited, but restricted by the available resources at the MBU, therefore plug-ins must be designed to be resource-efficient.

Some of these plug-ins need to have a peer at either the backend or the surrogate to interpret the data and make the plug-in useful; for example, remote MBU administration is best used with a corresponding MBU Administration JINI Service (see section 6.2.3).

5.4 Graphical User Interface

For the end-user of the MBU, the Graphical User Interface (GUI) is the most visible and thus important part of the MBU. It is out of the scope of this thesis to define what the GUI should look like and what actions are possible; however, it is important to define how a GUI can be implemented and designed, and what interaction with the system is possible.

5.4.1 Available technologies

The MobiHealth system uses a C# GUI that interacts using XML-RPC calls with the java core; system calls to the operating system are better defined in C#, since it uses the .NET Compact Framework, which is available at all Windows Mobile PDAs (as being used in the MobiHealth system). These system calls are necessary for, for example, retrieving information on battery status and signal strength. GUI design for C# is also quite simple using an IDE like Microsoft Visual Studio.

The big disadvantage of creating a GUI in C# is that the communication between the java core and the GUI must be defined. A Java GUI can directly use all functionality of the system, which enables the GUI developer to quickly use all functionality of the system. There are several APIs for designing a GUI in Java: AWT and Swing are both SUN defined toolkits; and SWT (Standard Widget Toolkit), which is developed by IBM within the Eclipse project. Swing is a large GUI framework that draws components using graphic primitives (e.g. lines and text) and is completely platform independent; SWT is an API that uses native widgets for drawing its components and relies on a platform-dependent SWT-library. SWT is slightly faster when than Swing, because it uses native code, however, Swing offers more functionality and is used more widely.

The main advantages of C# are better integration with the operating system (Windows Mobile), allowing more advanced features to be implemented and C# is a lot easier in use for designing a user-interface. Furthermore, if the user interface is explicitly decoupled from the Java MBU core by using a middleware, it also allows other programming languages to be used and even use different user-interfaces with the same core. Within the HS24 project some research to defining a clear middleware structure is being conducted.

5.4.2 Design

The design of the GUI depends largely on the requirements for the specific scenario. If the GUI is only used for starting/stopping a session, showing status information and occasionally an alarm, it does not require an extensive design. However, if interaction between the patient and the healthcare professional is required; it is necessary to design this interaction using the components the platform provides (plug-ins, event-sending, etc).

Very complex GUIs that also need to tap into the sensor data (for example to show snapshots), can use any functionality that the platform offers; a GUI can tap into the sensor bus to read sensor data, listen and respond to for any event, et cetera. However, we suggest keeping the GUI simple if it is meant for a patient, this will prevent the patient from affecting the system operation.

The GUI can be integrated in the MBU either as an MBU component or using the plug-in framework. If the GUI should never change, it is best to include it as a MBU component; however, if it is likely that the GUI must change during a treatment, for example if a patient enters a new phase of treatment, it can be included as a plug-in, which allows updating.

5.5 Operation

MBU operation strongly depends on the scenario in which the system is used; this section describes the main interaction between the components and how MBU operation can be defined. We have stated that the MBU

manager is responsible for the operation and define three main phases of the MBU manager, the start-up phase, the running phase and the shutdown phase; these are illustrated in Figure 5-6 and discussed below.

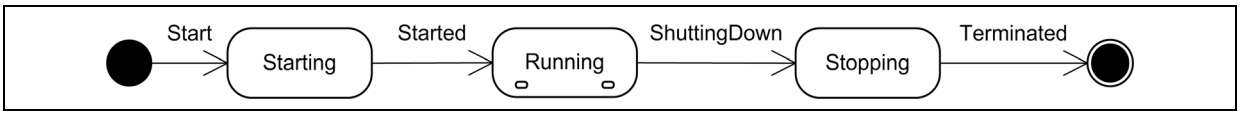


Figure 5-6 Main MBU state diagram

5.5.1 Start-up phase

In the start-up phase the system will read the configuration and initialize all components. This initialization is performed in several steps, which we will define below. The start-up phase will only occur once when the MBU application is started.

Step	Name	Description
1	Read properties	System wide properties are read and parsed; these properties define properties that are necessary for system start-up, for example the location of the configuration file.
2	Load core	The MBUManager (SiteCore) is initialized.
3	Core components	The generic core components are initialized by the site-core: a. Configuration manager b. Event bus c. Component repository d. Sensor bus e. Data block transfer The configuration manager initialization should be first, to allow other components to obtain their own configuration properties; since all other components may depend on the Event bus, it is initialized second.
4	MBU specific core	MBU specific core components are loaded: a. Device manager b. Communication manager c. Plug-in manager The initialization of these managers and the sub-components they generate may run in the main thread of execution, but they start also start own thread for further operation; therefore it can not be guaranteed that a connection is available when a plug-in is loaded.
4a	Devices	The device configuration is parsed and for each active device, the device driver will be loaded with his own configuration object; all devices will be registered within the manager. The device drivers should not connect to their device; this should be done after start-up.
4b	Communicators	The communication manager reads the communication configuration and loads the default communicator and issues the connect command.
4c	Plug-ins	The plug-in manager reads the configuration and loads all plug-ins that are flagged to be loaded and started.
5	Running phase	The system now moves to the running phase; it is possible that devices,

communication or plug-ins are not yet completely initialized. It broadcasts an event that indicates that the system is initialized and running.

Table 5-2 MBU startup

5.5.2 Running

The behaviour of the MBU at runtime depends mostly on the scenario in which it is used; therefore we will focus on a very generic behaviour in which the MBU will stream sensor data as soon as both a connection to the surrogate is available and sensor devices are connected and reading sensor data; because other MBU may depend on either the connection to the devices or the connection to the surrogate, there are four different states within the running state, as shown in Figure 5-7. As explained earlier in this chapter, this can be implemented using a Finite State Machine (FSM) that will redirect events to the active state. This active state implements state specific behaviour per event and generic behaviour is implemented in a default handler. MBU behaviour can be implemented in these states.

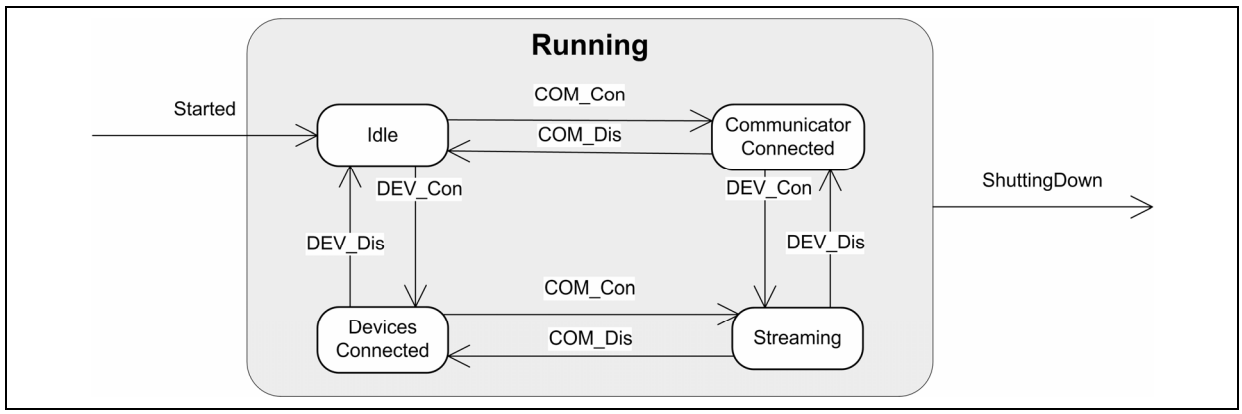


Figure 5-7 MBU Running composite state diagram

The running state consists of the following sub-states:

- **Idle:** The communicator is not connected and no devices are connected, this is also the initial state;
- **Communication Connected (ComCon):** The communicator is connected to the backend, but there are no device connected to the MBU;
- **Devices Connected (DevCon):** The MBU has a connection to the sensor devices and can receive sensor data, but is not connected to the surrogate;
- **Streaming:** The MBU is both connected to the surrogate and connected to the devices and is relaying the requested sensors to the surrogate.

The system changes state after each of the following events; depending on its current state.

- **COM_Con:** The communicator is connected to the surrogate;
- **COM_Dis:** The communicator has lost its connection to the surrogate;
- **DEV_Con:** The sensor devices are connected;
- **DEV_Dis:** The sensor devices are disconnected.

Figure 5-8 shows how the Finite State Machine can be implemented; the MBUManager initializes all four states (Idle, CommunicatorConnected, DevicesConnected, Streaming) at start-up and sets the current state to be Idle. The event bus sends events to the MBUManager using the *onEvent* method that is defined in the

AllEventListener; the MBUManager puts these events on a First-in, First-out or FIFO-buffer (possibly a priority enabled FIFO-buffer). The FSM method runs in a separate thread and pops events from the FIFO-buffer and delegates handling of these events to the current state. If there are no events in the buffer, the pop operation blocks. This approach decouples event handling from event generation. If an event causes a state-change; the handle returns the new state. Each state implements the *handle* method and defines state-specific handling for the received event; if the state does not define a specific handling it must delegate the event handling to the *handleDefault* method.

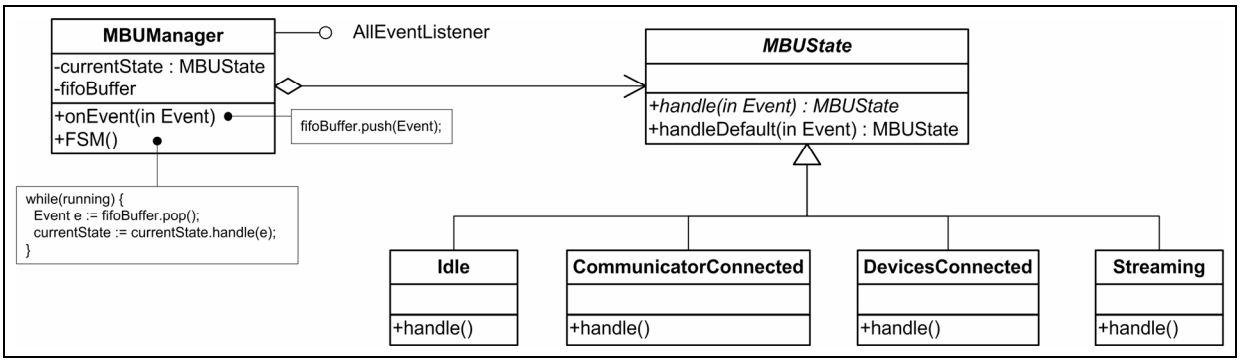


Figure 5-8 MBU Finite State Machine design

5.5.3 Shutdown

The MBU can move into the shutdown phase from any of the running phases. In most cases the GUI initiates the shutdown, but also an internal system event may cause shutdown, for example a low battery level. The most important step in MBU shutdown is broadcasting a Shutdown-Event; this allows all components to gracefully shutdown. The MBU suspends the termination phase for a certain period to allow the components to shutdown; after which it will force all existing threads to terminate. In the last step; the configuration manager will be issued the command to write the current configuration to file. This file can be used for the next time the MBU is started.

5.6 MBU/Surrogate communication

The MBU to Surrogate communication is the bottleneck of the system; therefore data transmission must be optimized. This section describes how we use the generic framework as presented in the previous chapter with the Mobile Service Platform. We need to map the concepts of the MSP onto the Communicator framework and specify an encoding of the PDUs and streams onto the MSP API.

5.6.1 MSP Communicator

The MSP communicator uses the IO API that is supplied with the Mobile Service Platform to communicate to the Surrogate. The IO API is very straightforward and defines two connections; the first is the *SurrogateHost Connection*, which is a connection to the surrogate host server. The MBU uses this connection to load its surrogate and receives a Surrogate Connection. This Surrogate Connection is the direct connection to the surrogate object and is used for sensor data and PDU communication. At the surrogate side, the MSP Communicator is associated with the surrogate object; which implements the Communicator side for the surrogate.

The communication framework is based on the MSP and therefore the one-way and request-reply PDUs map on the respective messages in the MSP and the sensor data stream maps on the MSP output stream. Therefore implementing the MSP transmitter, which is responsible for this mapping, does not require an extensive encoding scheme.

5.6.2 Receiver

The communicator sends incoming PDUs to the MBU Receiver; because sensor data is only sent from MBU to the surrogate, it does not deal with incoming data streams. The main task of the receiver is analysing the contents of the PDU and sending this PDU to the correct component. The basic platform only supports EventPDUs, MessagePDUs and DataBlockPDUs; therefore the receiver can decode the event, message or data block from the PDU and send it to the event bus as a broadcast event, a private event and to the data block manager respectively. For more complex situations; the receiver may be extended to register a destination for proprietary PDU formats; however, it should be possible to communicate most types of data using the events and data blocks.

5.6.3 Sensor data transmission

The sensor data relayer, or *SensorRelayService,* is responsible for transmitting the sensor data that is requested by the surrogate and backends. Since the CommunicationManager has context information on the available bandwidth, it is responsible for enabling and disabling the channels that are relayed. Sensors can only be disabled and enabled when its corresponding channel is already being relayed. To optimize bandwidth, the relayer must also support disabling sensors within a channel; however, since components at the surrogate or backend may not support changes in channel configuration, disabling a sensor must cause the surrogate to produce an overflow value for the disabled sensor. The operations that the SensorRelayService must support are shown in Figure 5-9.

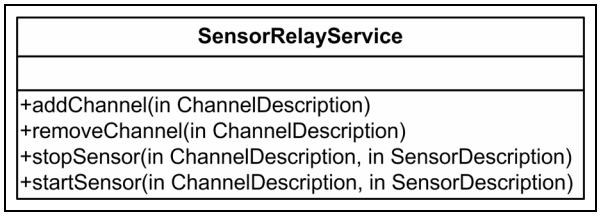


Figure 5-9 SensorRelayService operations

The output stream of the connection framework (and thus the MSP), is used to transmit the real-time sensor data; the output stream produces less overhead than is induced by the messages. The relay service must multiplex multiple sensor channels streams onto a single output stream because only one is available in the MSP (this may change in future versions). This approach requires a de-multiplex service (the sensor data receiver) at the surrogate that can convert the stream into the separate channels.

To be able to de-multiplex the data and to distinguish between the channels; the services could either exchange session management messages (using the MSP-messages) or the relay service can use in-band messaging. Since the MSP-messages can suffer from latency, it is not very suitable for real-time sensor data; therefore we will use in-band messaging. The sensor relay protocol defines the in-band messaging and is explained in the next section.

Figure 5-10 gives an overview of how sensor data propagates through the system; it originates from the sensor data producers who sent it into the sensor bus. The sensor data relayer subscribes to the channel it must relay and translates the sensor data into relay protocol data units (PDUs), which are encoded as a byte stream and sent

to the transmitter of the communicator. The communicator may perform transcoding by adding stream filters; an example of a transcoding filter is a deflate-filter, which compresses the data stream. The communicator's output stream is linked to the MSP, which converts the stream into HTTP chunks that are transmitted to the surrogate. At the surrogate, the MSP translates these HTTP chunks into an input stream that is pushed into the MBU communicator-receiver. The communicator unfilters the data (e.g. inflating) and sends the resulting stream to the sensor data receiver; the sensor data receiver reads the relay PDUs and registers all sensor channels at the (surrogate) sensor bus; for filtered sensors, the receiver inserts an overflow value. Services that connect to backend clients can subscribe for the sensor data at the sensor bus and communicate this to the backend client using some e-Health service protocol.

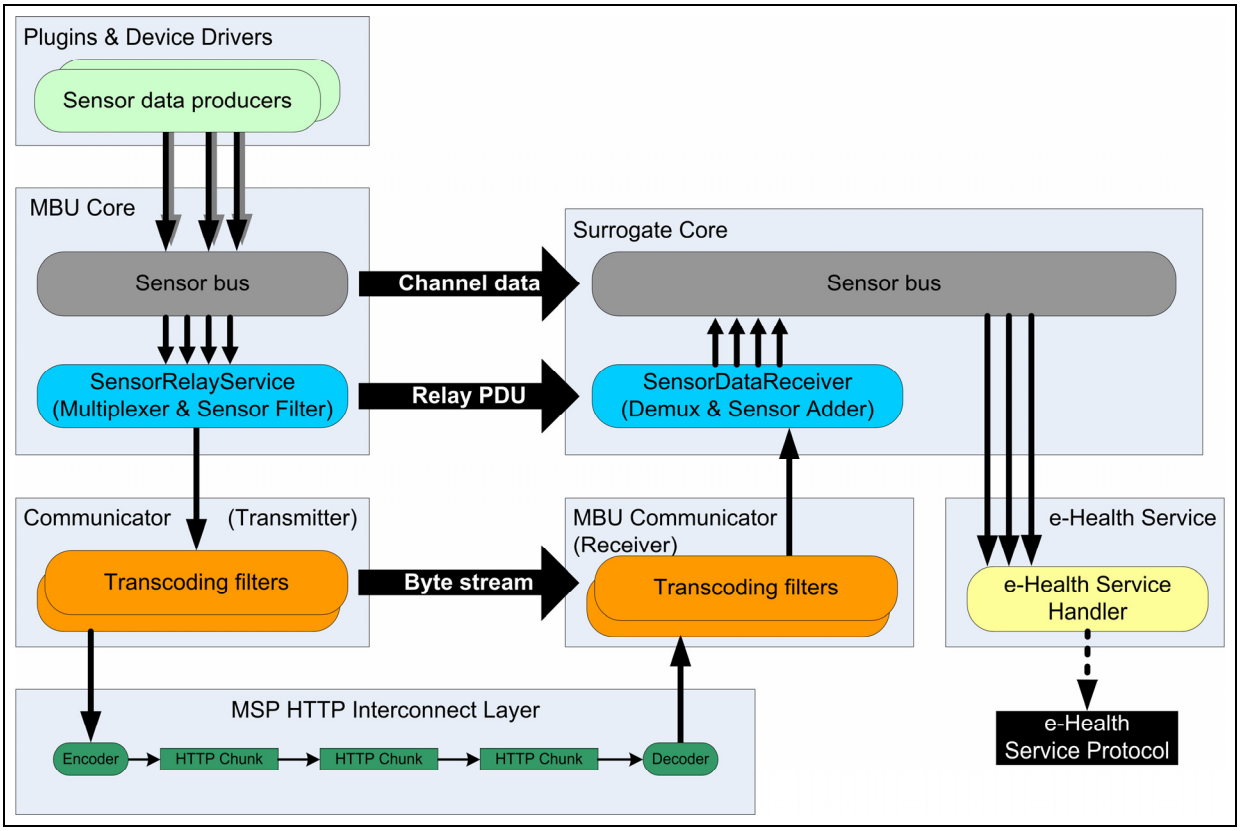


Figure 5-10 Sensor data communication overview

The figure illustrates what data is communicated between the various peer components at MBU and surrogate; the MBU sensor bus is connected to the surrogate sensor bus and (a subset of) the channel data is transmitted. At a lower level, the relay service sends relay PDUs to the sensor data receiver, and the transcoding filters exchange byte streams. At the, for our platform, lowest communication level, the MSP exchanges HTTP chunks.

5.6.4 Sensor relay protocol

The sensor relay protocol defines what PDUs that are sent from the MBU's sensor relay service to the surrogate's sensor data receiver. Since the communicator offers an output stream in which no data is lost and data is delivered in order, the protocol does not require extensive checksums nor sequence numbers. Because the decision on disabling/enabling channels and sensors is not the responsibility of the sensor relay service, only the results of these decisions need to be communicated, which simplifies the protocol and allows the protocol to be unidirectional. If the surrogate requires changes in the relay configuration, it can request this using the events.

The protocol supports the following PDUs:

Type	Payload	Description
Add Channel	ChannelID	A notification that the channel with the specific ChannelID will be relayed and thus the receiver can expect channel data for that channel; all sensors in this channel are enabled by default.
Remove Channel	ChannelID	A notification that the channel is removed, and thus that no more sensor data will be sent for this channel.
Stop Sensor	ChannelID, SensorID	A notification that the specified sensor will be removed from the sensor data for its channel; the receiver must write overflow values for that sensor to the sensor bus.
Resume Sensor	ChannelID, SensorID	A notification that the specified sensor will be included in the following data packets; overflow values are not necessary anymore.
Channel Data	ChannelID, Sensor Data	A sensor data packet for the specified channel.
Channel Mark	ChannelID, Mark	A mark/timestamp for the specified channel.
Ping	-	If no channels are relayed; the ping packet may be sent to prevent the output stream from a timeout, the receiver can discard it.

Table 5-3 Relay protocol data unit list

The receiver must deal with the following issues:

- **Missing sensor configuration:** The add channel operation may supply a ChannelID that is unknown in the sensor configuration the surrogate has; if this is the case, the receiver must request a new sensor configuration from the MBU and discard any incoming packets until the configuration is updated;
- **All sensors disabled:** If all sensors are disabled within a channel, the channel must be removed as a whole. If, for backend compatibility, it is required that overflow values are being continued, it is possible to add a new PDU type that indicates how many samples must be reproduced;
- **Synchronization:** When the relayer is gathering channel data from the sensor bus to create a channel data PDU; the system may receive a request for enabling/disabling a sensor within that channel. It is important that the channel data PDU is sent before the stop/resume sensor PDU, to prevent the receiver from running out of sync.

5.7 Conclusions

This chapter presented a completely new design for the Mobile Base Unit in comparison to the MobiHealth system. The device driver architecture is based on MobiHealth, but adds various states. Especially the plug-in mechanism and the communication framework provide significant improvements and increase implementation flexibility, to which also the Finite State Machine for MBU behaviour and the discussed approaches for user-interface design contribute. The communication manager and relay components see to it that bandwidth is used efficiently and offer the required run-time flexibility. The sensor bus and event bus play an important role in the MBU and are responsible for internal data distribution.

Although some gaps are left for future developers, the clear component structure and implementation flexibility provide a major advantage compared to MobiHealth.

6 Surrogate and Backend

This chapter describes the design of the MBU surrogate and discusses some issues in backend design. These components reside on the fixed network and are not limited by the constraints of a mobile environment. Bandwidth, processing power and other resources are not as scarce as they are on the MBU. Nevertheless, the design of the surrogate and backend is just as important as the design of the MBU, because they deal with the same, important, patient data and vital signs.

Surrogate design is closely related to MBU design, since the surrogate represents the MBU in the fixed network and provides the interface for interaction with the client applications that the healthcare professionals will use. The design of these client applications (the backend clients) is not within the scope of our research and also the communication between these client applications and the surrogate is left open. This chapter does provide insights on the design of both the backend client as well as the communication to these clients.

The first part of this chapter elaborates on the design of the MBU surrogate and its main components; the second section focuses on the JINI services that the surrogate can export; the third section gives some information on backend design

6.1 Surrogate

The surrogate is an intermediate object between a nomadic mobile service (NMS) and its clients; it is an essential part of the JINI surrogate architecture [SUN03] and was introduced to allow the NMS to join a JINI network. A JINI service must meet several critical requirements and an NMS often can not meet these requirements due to limited connectivity; therefore the surrogate acts on behalf of the NMS as the JINI service. This surrogate communicates with the NMS to offer this service. Surrogates are a well-known design-pattern in software design and the surrogate within the JINI surrogate architecture can be seen as a complex remote proxy [GAM94]. The MBU surrogate is the surrogate implementation for the MBU and is referred to as surrogate for the remainder of this chapter.

The surrogate is a transient component that is linked to a single MBU; meaning it will be loaded when the MBU is online and will be removed when the MBU goes offline. Besides allowing the MBU to join a JINI network, the surrogate also reduces the amount of data that the MBU sends over the network if multiple clients are connected.

Figure 6-1 shows the main components of the surrogate; the surrogate core defines the main functionality and contains the required SiteCore and all other core components. The surrogate host context and JINI Logic are required to support the surrogate architecture and JINI service exporting respectively; the surrogate host context

instantiates the surrogate object and provides some information on the connection to the MBU. The JINI logic enables the surrogate to export services into the JINI network, allowing JINI clients to connect. The MBU communication, or Interconnect Context, is the peer entity of the backend communication part in the MBU and is implemented as a communicator from the communication framework. The exported services block represents the services that are exported into the JINI network and that expose functionality to the backend clients. These exported services can be configured from within the MBU, which is explained in section 6.2

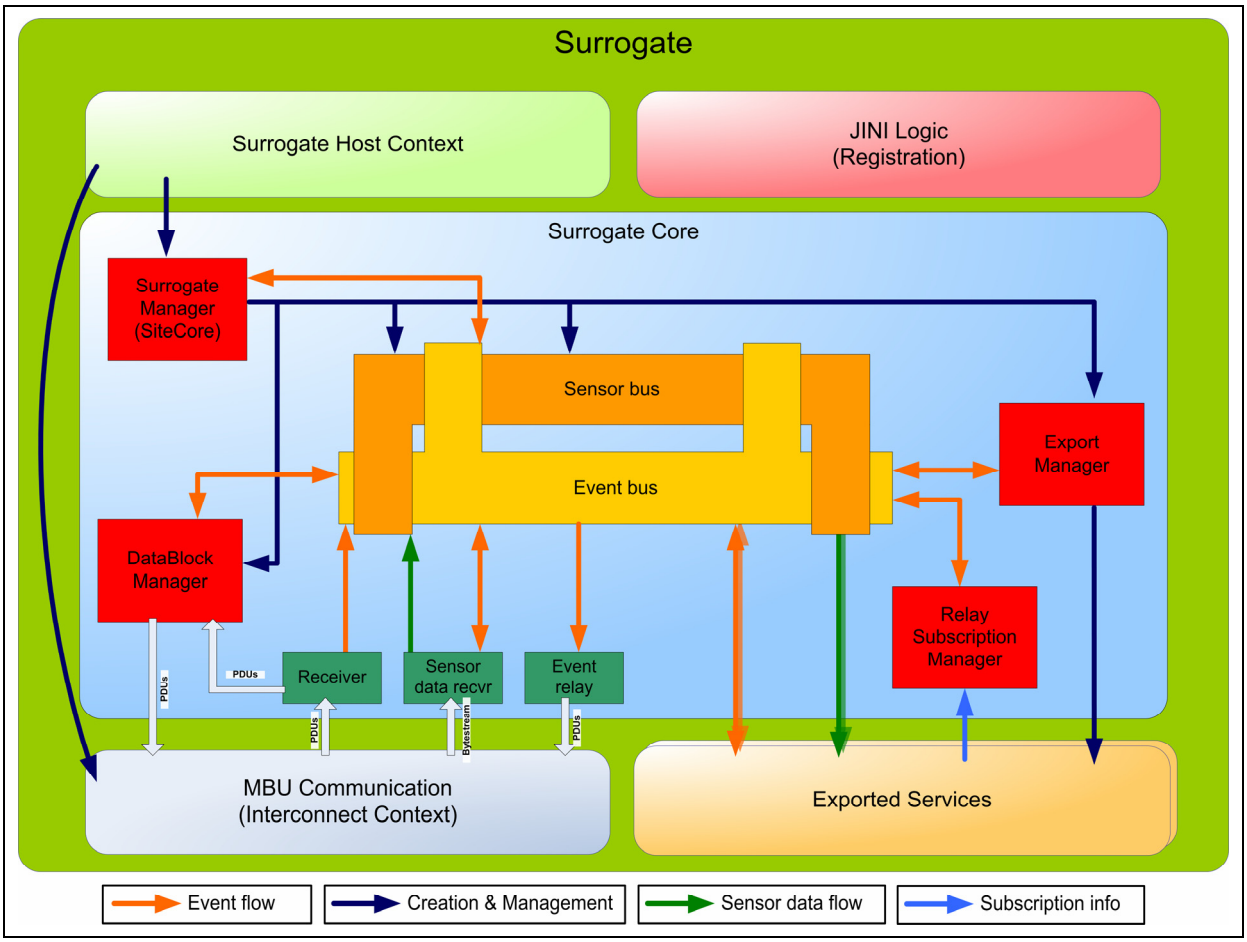


Figure 6-1 Surrogate structure

6.1.1 Surrogate core

The surrogate core is very similar to the MBU core and the same core components are present; the figure shows the sensor bus, event bus and data block manager and indicate that the *SurrogateManager* (or SurrogateCore) is the SiteCore for the surrogate. There are two surrogate specific components: the *RelaySubscriptionManager* and the *ExportManager*, the first keeps track of what sensor channels are requested by the backend clients and responds to that by requesting these channels at the MBU. The latter is responsible for loading and managing the exported services that are exposed for backend clients.

The SurrogateManager has the same responsibilities as the MBUManager: it loads and manages the core components and defines surrogate behaviour. A basic surrogate does not need an FSM, because surrogate functionality is quite limited; therefore the basic surrogate manager only requires an event stack and an event processor to decouple event processing from event generation.

In addition to this functionality; the surrogate manager also acts as a proxy for the MBU's device sensor configuration and sensor bus configuration; it will request them once and keep track of updates in these configuration (by processing update events). If a component within the surrogate requires one of these configurations, it requests them at the surrogate manager; this limits the amount of data that is transferred between MBU and surrogate. The surrogate manager also contains the surrogate configuration, which is provided by the MBU at initialization and is used to configure the exported services and possibly some overall surrogate settings.

6.1.2 MBU Communication

The MBU communication block in Figure 6-1 deals with communication to the MBU and offers the communicator API. The three green blocks deal with the data internally: the receiver (1) receives the incoming PDUs and either relays them to the data block manager or the event bus, the event relay service (2) is responsible for sending events to the MBU (events to backend clients must be handled by the exported services) and the sensor data receiver (3) handles the incoming sensor data byte stream.

The sensor data receiver (SDR) registers the incoming channels at the surrogate sensor bus and demultiplexes the incoming byte stream into sensor data, adding overflow values for each disabled sensor. This sensor data is published onto the sensor bus. The SDR requests the MBU sensor bus configuration from the surrogate manager and uses it to for the channel- and sensor-descriptions that are registered at the surrogate sensor bus.

Section 5.6 discusses the MBU/Surrogate communication in more detail.

6.1.3 Relay subscription manager

The relay subscription manager (RSM) keeps track of the channels that the backend clients request; this component is especially important if multiple backend clients are in use simultaneously. The exported services must subscribe at the RSM for each channel their clients have requested; the RSM requests these channels at the MBU based on these subscriptions. If all clients log off, their associated exported service *must* unsubscribe. If the RSM has no more subscriptions for a specific channel; it will notify the MBU, allowing the MBU to stop relaying that channel. Especially for scenarios where the backend clients often request different channels; this component will save a lot of bandwidth.

Once a channel is relayed, it is available in the sensor bus and any exported service can subscribe to it without subscribing at the RSM; however, it is possible that the channel will be closed suddenly, because the RSM had no more subscriptions.

6.2 Exported services

The exported services define the functionality that is exposed to the backend clients using JINI, for example sending sensor data or alarm events; the exact functionality depends on the scenario in which the platform is used, but a part is also common for many scenarios. To maintain implementation flexibility, it must be possible to add additional exported services without changing the surrogate implementation itself. This allows a developer

to design and implement the functionality he requires, using the components and data that is available in the surrogate, without changing the surrogate core functionality. This section discusses a framework that allows this implementation flexibility.

It is possible to either implement one large service that contains all necessary functionality (1), or it is possible to implement several small services that offer the functionality together (2). The first simplifies the development of an integrated end-user application that uses all functionality, but enforces the application developer to change his entire service for each scenario. The second allows access restriction per service and allows a variety of client applications that serve a different purpose, but the implementation of an integrated end-user application is more complicated. Examples of these services are provided in section 6.2.3.

6.2.1 JINI Services

Chapter 2 shortly explained JINI services; this section will touch upon the subject in more detail to give insight in the structure of the exported JINI health services. A JINI service exports a service proxy object into the lookup service, which can be retrieved by a JINI client. This proxy object can either be a simple stub class that only accesses remote methods; or it can be a smart proxy that contains both remote and local methods and allows data caching. Since a smart proxy can easily behave as a stub, we will only use smart proxies.

Figure 6-2 shows a class-diagram for a generic JINI health service; application developers should follow this structure if they implement a new exported service. There are two main interfaces: the *HealthService* and the *RemoteHealthService*; the first defines the actual service methods that are offered to a client and the second defines the (private) protocol between the client and the server. The *HealthServiceProxy* is the smart proxy object that is exported into the lookup server and downloaded by the client; it communicates with the *RemoteHealthServiceImpl* (the server), via the *RemoteHealthServiceImplStub*. The server is responsible for exporting the proxy and it uses *Entry* objects to specify service properties and uses a universally unique identifier (UUID) to ensure that possible duplicate entries can be distinguished. Entries are used to pass additional information about services that clients can use to decide if a particular service is what it wants.

The RemoteHealthServiceImplStub is transferred within the proxy to the client and is responsible for invoking methods on the RemoteHealthServiceImpl by serializing and sending the request, and receiving, deserializing and returning the response. Stubs are a common principle in distributed applications and can be generated either at compile time or at run-time, depending on the export mechanism used.

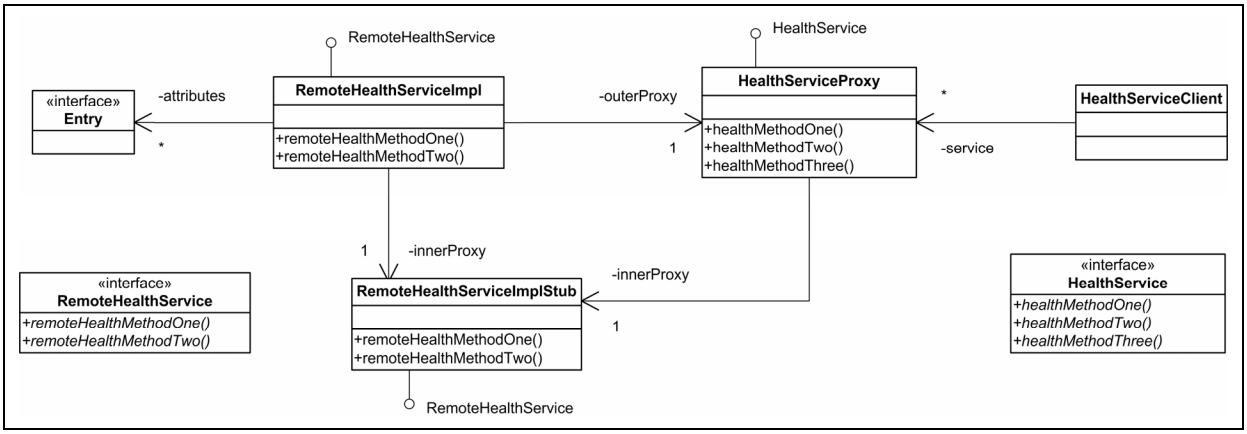


Figure 6-2 JINI Health Service classes

The application developer that designs this remote service should also pay attention to security; elaborating on all JINI security mechanisms and approaches is out of the scope of this thesis, more information on this subject can be found in [MAR03], [SCH00] and [SOM03].

6.2.2 Service export framework

To achieve the implementation flexibility for the exported services, we present a simple service export framework (see Figure 6-3) that allows a service developer to focus on the exported service itself, instead of loading and exporting the service. The most important component in this framework is the *ExportManager*, which is responsible for loading the exported services at start-up, and possibly also on request. It parses the export configuration that was sent from the MBU to the surrogate and then initializes all *ExportedService*-components that are defined in this configuration. The configuration is used for determining the class-file of the exported service and defines service specific attributes. It also defines an optional JAR file or a codebase URL that contain the class-files for this service, it is important to note that this functionality may cause security issues; therefore, if security can not be guaranteed, specifying a JAR file or codebase can be ignored. The ExportManager also defines a list of *Entries* that define MBU specific service properties; these may be used by the export service, but the service can also define its own entries.

The *ExportHelper* is a utility class that allows dynamic loading of the service, based on the class-file and its configuration. Furthermore it contains standard methods for creating a new unique identifier and creating (run-time) stub-files for the *RemoteHealthServiceImpl*. It also contains a standard method for exporting the service into the lookup service. These methods are available to simplify development, but if more complex methods are required, they may safely be ignored.

Both the exported service and the RemoteHealthServiceImpl can use the event bus and sensor bus to communicate with the system. The exported service should subscribe for the sensor channels if it deals with sensor data; furthermore, it should subscribe to the event bus to intercept messages for remote peer components.

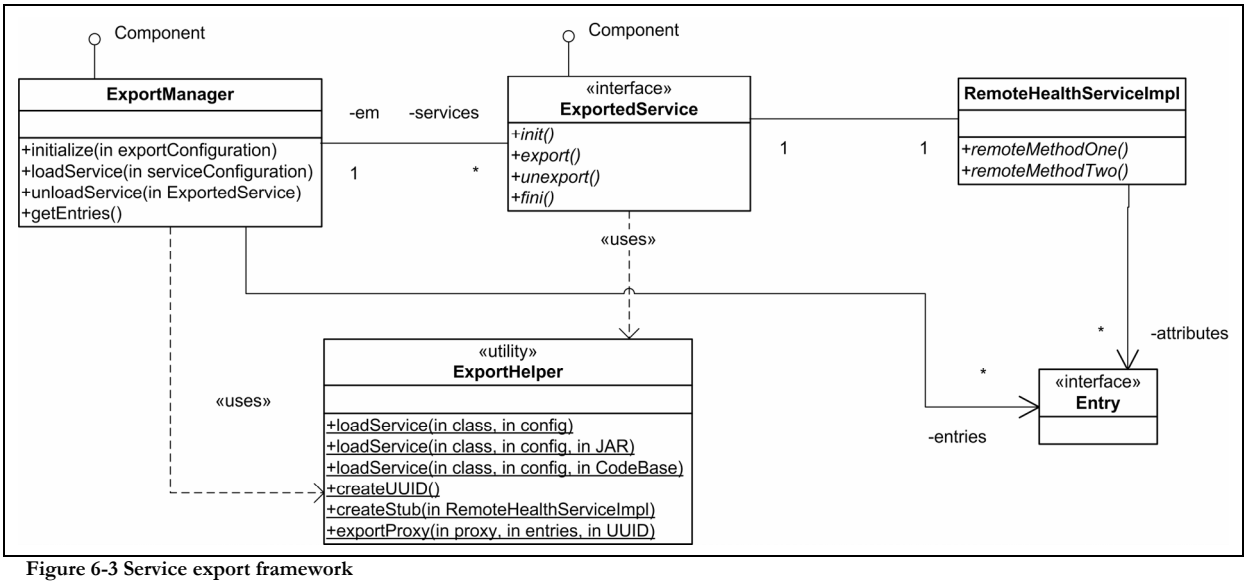


Figure 6-3 Service export framework

6.2.3 Recommended services

This section gives some services that we recommend for design and implementation; using these services it is possible to select the services required and reducing the implementation effort for a scenario.

- **MBU Medical Service:** The MBU medical service is the most important service the surrogate offers; it offers the sensor data, related system events and possible patient activities all in one. This service is quite complex and should be designed together with the backend client(s) that will use it. Possible backend clients are sensor data viewers and sensor data storage applications. This service should only be accessibly to the patient's health care professionals and possible the patient himself.
- **MBU Administration Service:** The MBU administration service provides MBU administration functionality; like updating MBU configuration, reviewing log-files and possibly component updating. This service should only be accessible by a system administrator, since it may require some training to ensure MBU security and operation.
- **MBU Chat Service:** This service allows mutual conversation between a health care professional and the patient.
- **MBU Status Service:** This very simple service gives an overview of the MBU status, including the loaded plug-ins and other exported service that are associated to this MBU. The system administrator application may use this service to provide an overview of all running MBUs to the administrator.
- **MBU Location Service:** The location service provides (GPS) location information on the patient; which can be used to dispatch emergency services or voluntary carers to the patient if an emergency occurs. [ESU05] describes the basis of such a service. Patient privacy should be carefully considered for this service.
- **MBU File Service:** The file service allows health care professionals or administrators to access files on the MBU; this may be pictures, videos or questionnaires that the patient created, or log-files or system files. Security is very important in this service, since it may expose patient data.

In our prototype implementation, we will implement a simple MBU Medical Service; however, since backend design is out of our design scope, this is just a prototype for testing purposes and should be redesigned in coherence with the backend application.

6.3 Backend clients

Backend clients are the clients that connect to the exported services and use these services to either provide an interface to the healthcare professional or other users of the platform, or they are automated applications that store, analyze or process the data. As stated before, we will not focus on backend client design and therefore we will suffice with discussing the two types of backend clients that we have identified. The communication between backend and exported service can be standardized, therefore we have provided some surrogate-to-backend communication mechanisms in section 6.4.

6.3.1 Persistent backend client

A persistent backend client is a client that is always available and is used for automated behaviour and storing and logging of MBU data and events. For example a sensor data repository to store historic sensor data or an alarm service that warns a healthcare professional in case of an emergency.

These persistent clients can receive their data either by connecting to each MBU-service that is exported into the lookup service; or it is possible that the surrogate connects to these clients (which, in that case, are services). We suggest using the first approach, most importantly to keep the surrogate as simple as possible and also to allow additional or changed services to be added more easily. If the surrogate connects to a persistent backend 'service' itself and also exports a service for similar backend clients, this will add to the complexity of the surrogate.

These persistent backend clients may export new JINI health services themselves; of which a sensor data repository is a good example: it receives sensor data from the surrogate and stores this locally; other (transient) backend clients may request this data for analysis of the patient's well-being. The current BAN Data Repository (BDR) provides such functionality; however, it offers the historic sensor data via an HTTP connection and is not a JINI service itself.

6.3.2 Transient backend client

The opposite of a persistent backend client, is a transient backend client; these clients are not always available and are used to temporarily communicate with the MBU via the surrogate. The most important transient backend client is the sensor data viewer, which is the application that the healthcare professional uses to view the patient's vital signs on screen. Transient backend clients may be either very simple; communicating only with a single health service and having limited functionality, or they can be very complex, communicating with multiple health services concurrently.

A transient backend connects to the JINI lookup service and looks for MBU services that match its requested template; using the service type and *Entry* objects. When a service is found; the proxy is retrieved from the lookup service and used to further communicate with the backend.

PortiLab is an example of a transient backend client; however, it is not JINI enabled and therefore relies on persistent backend services (the BAN Streaming Service) to receive the sensor data via HTTP.

6.3.3 Backend design

Depending on the complexity of the backend application; the application developer can choose to implement the backend using the core components of the mobile e-health platform by using the sensor and event bus and by implementing a backend core component. The core components are designed to be reusable and can reduce backend implementation effort. If a simple backend application is required; the core components can be ignored safely.

The design of the backend requires extensive domain knowledge and is out of the scope of this research.

6.4 Surrogate/Backend communication

This section describes some approaches for surrogate to backend communication. Since the exported services and backend applications are left for future development, it is possible to use a different approach, depending on the application developer's preference and the required functionality.

6.4.1 Connection framework

Although JINI uses Java Remote Method Invocation (RMI) extensively; it is not required for JINI services to communicate with their clients via RMI. If multiple clients are developed for a single service; it may be necessary to support multiple protocols for surrogate backend communication. The connection framework as presented in the previous chapter allows decoupling of the service implementation and the protocol being used. For example, an extensive sensor viewer that is a stand-alone application may use RMI, while a web-based sensor viewer uses raw sockets.

The client requests a certain protocol and the smart proxy object checks if it supports this protocol; for each protocol, the proxy must contain connection information, like a host/port combination for raw sockets or an RMI-address for RMI communication. The smart proxy requests a connection at the exported service, possibly using authentication information; the service then loads and registers a communicator for the backend client and returns additional connection information to the proxy. The proxy then creates a communicator for the backend client. The backend client and exported service now both have a communicator and can communicate data as desired. It is not necessary to implement a similar protocol as defined between MBU and surrogate on top of this communicator; the usage of the PDUs and output stream can be defined per exported service. It is also possible for a single communicator to use different protocols; for example, use RMI for PDU communication and use raw sockets for sensor data communication.

This approach is based on the design of the Generic Connection Framework (GCF) that was introduced in Java2 MicroEdition [ORT03] and is also ported to Java2 Standard Edition [JSR197]. SUN initially designed this framework to constrain connectivity memory usage for mobile devices.

6.4.2 Connection setup

Figure 6-4 shows how a connection is established using the connection framework, the health service and proxy.

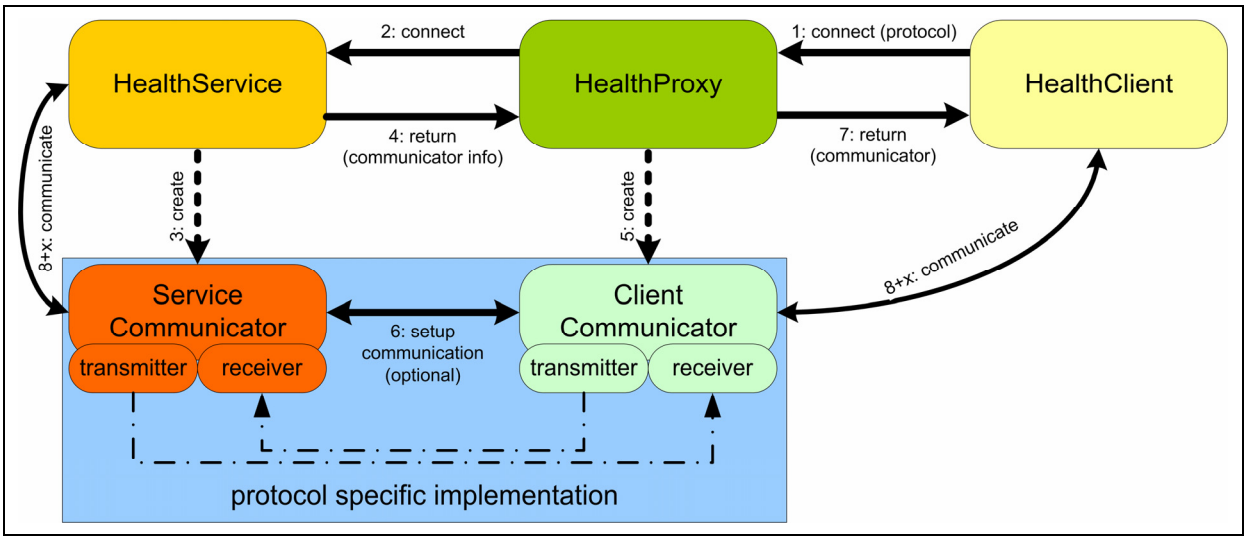


Figure 6-4 Generic backend communication setup

The health client has retrieved the proxy object from the lookup service and calls the connect-method, supplying the desired protocol (step 1). The proxy checks if it supports the protocol and uses a protocol specific method to connect to the health service using the service address as set in the proxy (step 2); the health service creates a

communicator that will deal with this client's communication (step 3). If complex functionality is required for client handling; the health service may use a client-handler intermediate that contains this functionality. When the communicator is created, optional communicator info is returned to the health proxy that may be required for the client communicator to connect to the service communicator (step 4). It is also possible that this communication is already established, depending on the protocol. The health proxy creates the client side communicator (step 5), which is linked to the service communicator via the transmitter/receiver pairs (step 6). This communicator is returned to the health client (step 7) and the communication is now established. The health-client and health service can now exchange data, using their communicators (step 8+x).

6.4.3 Sensor data communication

Sensor data can be communicated using the output stream functionality of the communication framework. Since MBU to surrogate communication was based on the MSP; the MBU communicator only supported a single output stream for sensor data communication. Backend communication is not limited by the MSP and therefore this communicator may support opening multiple output streams. In this case, the service and client may use a single outputstream per sensor data channel, which simplifies sensor data communication, because multiplexing is not required. The exact design of this functionality is left for future work.

Is most cases, bandwidth is not important for backend communication, because high-speed connections are available. Therefore, optimizations like compression and multiplexing are not necessary for surrogate to backend communication, although they can be implemented. Sensor data security can be an issue and must be considered in the protocol specific implementations or in the sensor data stream itself.

6.4.4 Event and data block communication

Although the protocols used for backend communication may support transferring event objects or large blocks, it is not certain if all possible protocols support this. The events and data blocks are therefore encoded onto PDUs to support the communication framework. Internally, the communicators for each protocol can define how they transfer these PDUs. The JINI architecture supports *RemoteEvents;* for a JINI/RMI enabled communicator they provide a possible transport method for PDUs.

6.5 Conclusions

The surrogate design presented in this chapter is totally different compared to the MobiHealth surrogate; we have tried to keep it as simple as possible, using the platforms core components. To support the implementation flexibility required for backend applications, we introduced the exported services framework, this allows scenario developers to implement their own service and define their own protocol for sensor data communication. The chapter also provided an introduction to backend design, but this is mainly left for future work.

7 Prototype implementation

To verify our design and to test whether or not the design was usable at a mobile device, we implemented a prototype; this chapter describes the implementation of this prototype. We will not discuss the complete implementation, but focus on the usage of the components to illustrate how scenario developers can use the platform (Section 7.1). This chapter also describes the generic functionality and deployment of the prototype (Section 7.2).

7.1 Implementation

This section describes how the functionality of the platform can and should be used by providing code snippets in Java. We chose Java, because both the JINI surrogate architecture and the Mobile Service Platform are available for Java and also because Java is a platform independent programming language. This allows us to deploy the system on different operating systems, which is beneficial for the backend servers, the MBU software and the backend client. The MBU runs at a mobile device that may have limited resources available; moreover, these devices often can not run a complete Java virtual machine. We have implemented and tested the MBU software with the IBM J9 JVM, which is a Java virtual machine that is available for Microsoft Windows Mobile as well as Linux PDAs and implements the Java 2 Standard Environment (J2SE). Devices less powerful than a PDA, like a smart-phone, often do not support J2SE, but implement the Java 2 Micro Edition (J2ME), which is restricted in functionality. Due to increasing resources and decreasing size of the contemporary PDAs, we state that a PDA is the best device to use as an MBU. Therefore we have not tested our implementation with J2ME.

7.1.1 Components

It is important to have single instances of the core components; the design implies that the SiteCore is responsible for creating these components. Other components can get a reference to these core components by accessing the site core. The SiteCore must therefore be accessible by all locations, which is implemented by using an abstract singleton approach: The SiteCore is defined as an abstract class with an *_instance* variable that can be requested using the *instance()* method; the site-specific core component (e.g. MBUManager) must extend this class and call its constructor with its site ID number. The SiteCore class provides non-static access to the core components using get*Component* method, but also contains a short-cut method that simplifies accessing them. Table 7-1 shows a partial implementation of the SiteCore.

```
public abstract class SiteCore implements Component {

   protected SiteCore(int id) throws CoreInitializationException {
      if (_instance != null) {
         throw new CoreInitializationException("SiteCore already initialized");
      }
```

```
    // Assign local variables
    _ID = id;
    _instance = this;

    // Initialize core components
    initCore();
}

public static SiteCore instance() {
    return _instance;
}

public SensorBus getSensorBus() { /* implementation not shown */ }
public static SensorBus SensorBus() { /* implementation not shown */ }
// ...
```

Table 7-1 SiteCore initialization code and sensor bus reference

Core components for a specific site, can be requested at the site specific core. For example, the CommunicationManager is always available at the MBU, but not at the surrogate. Components that depend on this manager are site specific and can therefore use an explicit reference to the MBU-core. Since the MBUManager is an implementation of the SiteCore, it can be type-casted. Table 7-2 shows how we have implemented this.

```
// Type-cast the sitecore instance to the site-specific core
MBUManager mm = (MBUManager) SiteCore.instance();
CommunicationManager cm = mm.getCommunicationManager();
Commmunicator c = cm.getCommunicator();
```

Table 7-2 Site specific core referrals

All core components can be requested using the SiteCore instance-method; however, to reduce inter-component dependency, non-core components are not available in SiteCore. If a non-core component requires a reference to some other non-core component, it must use the component repository. Therefore all components must register in the component repository. This registration is facilitated with the *ComponentLocation* address, which, for example, can be retrieved from an incoming event. When a component stops, it must unregister itself from the repository. Table 7-3 illustrates this.

```
// Register component in Repository using SiteCore reference
ComponentLocation loc = SiteCore.instance().getComponentRepository().register(this);

// Locate a component in Repository using a ComponentLocation (loc2)
Component c = SiteCore.instance().getComponentRepository().getComponent(loc2);

// Unregister component from Repository
SiteCore.instance().getComponentRepository().unregister(this);
```

Table 7-3 Component registration and lookup code

7.1.2 Sensor data

Device drivers and plug-ins can publish sensor data at the sensor bus and this sensor data can be retrieved by any component. For a data publishing component, the first step is to register the sensor channels at the sensor bus and the second step is to publish the data. Optionally, the data publisher can mark its data, as discussed in section 4.3.2. Registering a channel requires a channel description, it is assumed that the publishing component has knowledge of the data it is publishing and therefore can construct the channel description and corresponding sensor descriptions. The registration results in a sensor channel, which basically is an OutputStream and can be

used to *write* data. The *mark* operation is used to provide data marks or timestamps. Table 7-4 shows illustrates how the methods are used.

```
// Register a sensor channel using its ChannelDescription (chanDesc)
SensorChannel sc = SiteCore.SensorBus().registerChannel(chanDesc);

// Publish a byte-array containing one second of sensor data into the sensor channel
byte data[] = new byte[chanDesc.getByteSize() * chanDesc.getFrequency()];
/* fill data[] array from data source (device input or signal processing */
sc.write(data);

// Optionally publish a mark (int seqnr) for this sensor data
sc.mark(seqnr++);
```

Table 7-4 Sensor data publishing

A component that requires this sensor data can request it at the sensor bus; first it must subscribe for the channel data after which it can pull the data from its designated channel input. The component requires the channel description for the component to be able to subscribe. It can obtain this channel description via events or by locating a specific channel in the sensor bus. To facilitate this, the sensor bus keeps an internal SensorConfiguration that matches with the available channels. After subscribing, the consumer can read the sensor data as a regular inputstream, requesting as many bytes as it requires. The consumer should take sensor data marking into account, if this is enabled. Table 7-5 shows how data consuming is implemented.

```
// Obtain a reference to the sensor bus configuration
SensorConfig sc = SiteCore.SensorBus().getConfig();

// Locate a specific channel by its name (note: this name is defined by the data publisher)
ChannelDescription cd = sc.findChannel("FingerSensor");

// Subscribe for the channel and get the InputStream
ChannelInputStream cis = SiteCore.SensorBus().subscribeChannel(cd);

// Determine buffersize at one second of data and create a buffer
int bufSize = cd.getByteSize() * cd.getFrequency();
byte data[] = new byte[bufSize];

// Read sensor data
cis.read(data);

// optionally, read the mark here
```

Table 7-5 Sensor data consuming

7.1.3 Events

As with sensor data, events can be published and received; however, events are pushed to the event-consumer and therefore follow a different structure. Furthermore, the event bus supports unicast events that are directed to a specific component.

Publishing a broadcast event is quite straightforward; the publishing component creates an Event-object that describes the nature of the event and includes the event source. It then publishes this event at the event bus, which ensures delivery to all subscribed components. Events that should also be transferred to other sites must be transferred by the relay-component, which are a subscriber as well. Table 7-6 shows how a simple chat message is encoded as an event and published at the event bus. It also shows how it can be sent as a message.

```
// Initial variables available
String msg = "hello world";
ComponentLocation _loc; // = this component's location

// Create an event object, set its broadcast-reach and add the chat-message as payload
// The Event.CHT_MSG_PATIENT is a unique ID number for that event
Event e = new Event(Event.CHT_MSG_PATIENT, _loc);
e.toSurrogate();
e.toBackEnd();
e.setPayload(in.getBytes());

// Publish the event at the event-bus
SiteCore.EventBus().publishEvent(e);

// .. OR .. send the event to a specific component
ComponentLocation _dest; // The remote destination. For example from broadcast event
SiteCore.EventBus().sendMessage(e, _dest);
```

Table 7-6 Broadcast and unicast event publishing

Components can only receive events if they have subscribed for them, this goes for both broadcast as well as unicast. As discussed in the event bus design section, components can subscribe for single events, for events in a specific group and for all events. Furthermore, components must register at the event bus if they want to receive unicast events. Depending on how a component subscribed for an event, its *onEvent* method is called if the event occurs. For incoming messages, the event bus uses its *onMessage* method. Table 7-7 illustrates the subscription and reception of events.

```
// constructor
public EventConsumer() {
   // subscribe this component for incoming chat messages from the patient and the system
   SiteCore.EventBus().subscribeEvent(Event.CHT_MSG_PATIENT, this);
   SiteCore.EventBus().subscribeEvent(Event.CHT_MSG_SYSTEM, this);

   // subscribe for unicast events (messages)
   SiteCore.EventBus().register(this);
}

// handle incoming broadcast events
public void onEvent(Event in) {
   switch(in.getType()) {
      case Event.CHT_MSG_PATIENT:
         String msg = new String(in.getPayload());
         //handle patient message
      case Event.CHT_MSG_SYSTEM:
         String msg = new String(in.getPayload());
         //handle system message
   }
}

// handle incoming unicast events
public void onMessage(Event msg) { /** Implementation not shown, similar to onEvent **/
```

Table 7-7 Event subscription and reception

7.1.4 Communication

Communication between the MBU and surrogate is implemented using the generic communication framework; components can send PDUs and a data stream. This section shows how the framework should be used to transmit and receive data.

The MBU only has one communicator and uses its transmitter to send data; this communicator is available when the system is connected and can be requested at the CommunicationManager. The transmitter supports the *sendOneWay* and *sendRequest* operations for transmitting a PDU and *getOutputStream* for a single outputstream. The

outputstream is used for sensor data relaying, while the PDUs can be used for any type of discrete data, although it can be limited by a certain payload size. Table 7-8 illustrates event-relay components transmits an event, which encoded onto a PDU.

```
// Get the transmitter from the communication manager using sitecore type-casting
CommunicationManager cm = ((MBUManager) SiteCore.instance()).getCommunicationManager();
Transmitter t = cm.getCommunicator().getTransmitter();

// Create a PDU from an event
Event e = new Event(Event.CHT_MSG_PATIENT, this); // for illustration purposes
PDU p = new EventPDU(e);

// Transmit the PDU using the transmitter
t.sendOneWay(p);
```

Table 7-8 Sending an EventPDU

Incoming PDUs and streams must be handled by a Receiver component; the receiver defines what happens with what type of incoming data. For example, EventPDUs must be decoded and sent into the event bus. Table 7-9 shows an example receiver that handles a few PDUs and an incoming stream.

```
// Class definition
public class MyReceiver implements Receiver {

   // Handle incoming OneWay-PDUs (not requiring direct reply)
   public void receiveOneWay(PDU p) {
      switch (p.getPDUType()) {
          case PDU.T_EventPDU:
             // handle incoming broadcast Event
             EventPDU ep = (EventPDU) p;
             SiteCore.EventBus().publishEvent(ep.getEvent());
             break;
          case PDU.T_MessagePDU:
             // handle incoming unicast Event
             MessagePDU mp = (MessagePDU) p;
             SiteCore.EventBus().sendMessage(mp.getEvent(), mp.getDestination());
             break;
          /* handle other types */
       }
   }

   public PDU receiveRequest(PDU p) { /** not shown **/ }

   // Handle data stream
   public void offerInputStream(InputStream is) {
      // delegate stream to the sensor data receiver
      SensorDataReceiver sdr = new SensorDataReceiver(is);
      sdr.go();
   }
}
```

Table 7-9 Example receiver

7.1.5 Configuration elements

Configuration elements play an important role in the runtime flexibility of the system. The configuration defines what dynamic components (plug-ins and device drivers) are loaded and what communicator is used. The XML-file that contains the configuration is parsed using the kXML library [KXML], which produces XML-element objects. An element is supplied as parameter to the dynamic component, which knows how to parse the object and retrieve its settings. Since an element can contain other elements, it can be used to describe complex configurations. Table 7-10 shows how we use a plug-in element to load all active plug-ins.

```
// method called at initialization of the plug-in manager
public void parseConfig(Element plugins) {
  // get # of child plug-in elements
  int cnt = plugins.getChildCount();

  // Iterative the child plug-in elements, each represents a plug-in configuration
  for (int i=0; i < cnt; i++) {
     Object o = plugins.getChild(i);

     if (o instanceof Element) {
        Element e = (Element) o;
        String name     = e.getName();
        // get plug-in information
        String isActive = XMLUtil.getAttributeValue(e, "active", "true");
        String clazz    = XMLUtil.getAttributeValue(e, "class", "ehealth.mbu.plugin." + name);
        String jar      = XMLUtil.getAttributeValue (e, "jarfile", "");
        // only load active plugins
        if (isActive.equalsIgnoreCase("true") || isActive.equalsIgnoreCase("yes")) {
           Plugin p;
           try {
              // Dynamically load the plug-in and initialize it
              p = PluginLoader.loadPlugin(clazz, e, jar);
              p.init();
              _plugins.add(p);
           } catch (Exception e) {
              /** error handling **/
//...
```

Table 7-10 Plug-in configuration parsing

7.2 Prototype

To evaluate the validity of our design, we implemented a small prototype. This prototype can also be used to demonstrate the abilities of the design and forms a basis for further implementation. This section shortly describes the prototype: the first section discusses the included functionality, the second section shortly explains the implementation strategy, the third section discusses the operation of the prototype and the fourth section elaborates on the deployment of the prototype.

7.2.1 Functionality

The prototype contains simple implementations of all core components; furthermore, we have defined simple site behaviour for the MBU and surrogate. The prototype does not contain extensive error handling nor is tested extensively.

The MBU contains all its core components and uses a communicator that is based on the MSP HTTP-interconnect protocol. We have developed two device drivers; one for Bluetooth Mobi communication and one to read from a MobiHealth stored session file. It contains a sensor relay component that can enable relaying of sensor channels and also enable/disable certain sensors within this channel. The plug-in mechanism is also fully implemented and allows loading plug-ins from a separate JAR file. The user interface contains button to connect to the surrogate and to connect and stream data from the devices, it also provides a list of channels available and allows the user to start relaying these channels. To demonstrate the abilities of events and data blocks, we have included a simple chat-interface and a file-sending mechanism.

The surrogate contains the core components, but does not contain the exported service framework. We did implement a medical service that also contains chat and file-sending functionality. The service only provides the

functionality to demonstrate the system and is not very stable or versatile, for example, it does not use the generic communicator approach as discussed in 6.4. It can handle multiple clients at the same time.

For demonstration purposes, we also implemented a simple backend client. This backend client plots sensor data like commercial medical monitors; however, the data is automatically scaled and would most likely not be of any use for a healthcare professional. The client allows querying a lookup service for available MBUs, selecting an MBU and requesting sensor data at the surrogate. The surrogate will ask the MBU to start relaying the sensor data and the client will then receive it. Furthermore, the client can receive files from the MBU and provides a simple chat-interface to communicate with the patient. Sensor data from different channels is *not* synchronized.

7.2.2 Implementation

As explained in section 7.1, we implemented the system in Java. We use the Java packaging hierarchy to order our classes; site-specific components are all in the site root package and the shared components are in the shared package. This distinction allows easy identification of a component and restricts coupling between components at different sites if two simple rules are obeyed: shared components can not use site-specific components and site-specific components may only use components from their site, or the shared components. The exported services and their clients share classes that are service specific (like the proxy) and may therefore ignore these rules. Within the site-specific and shared packages, sub-packages are defined per component. Appendix D gives an overview of the packages we created for the prototype. For general purpose functionality, we have used several existing Java libraries; Appendix E lists the used libraries and describes each library in short.

Figure 7-1 shows the user-interface of the MBU for our prototype. The user-interface contains some expert functionality that for a patient should not be exposed; however, we use this functionality to demonstrate the capabilities of our prototype. The first screenshot (a) shows the connectivity overview, containing the connection status, an estimate for bandwidth usage and the amount of channels being relayed. The second screenshot (b) shows the status for the devices and the amount of channels and sensors connected to those devices. The third screenshot (c) illustrates the sensors that are available in the MBU sensor bus and thus can be relayed. The last screenshot (d) shows the simple chat-screen that can be used to communicate to the physician, if he is available. Note that in some of the screenshots the CPU-usage indicator (green bar in top of screen) shows high CPU usage, this is related to the capturing of the screenshots.

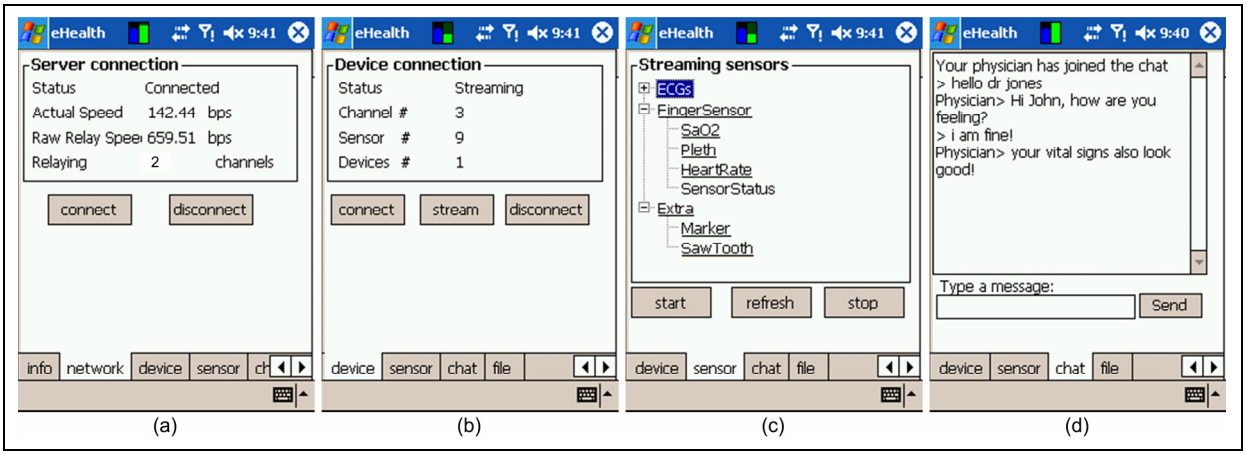


Figure 7-1 MBU user interface: network connectivity (a), device connectivity (b), sensor overview (c) and chat screen (d)

Figure 7-2 shows the backend client we developed for visualizing the data our platform produces. The upper-left corner shows two configurations; the remote configuration shows the sensors that are available at the MBU and the right shows the local configuration, with the sensors that are currently being relayed and can be visualized. The upper-right corner shows three discrete vital signs, like oxygen saturation and heart rate and the centre can show up to four a graphs, like plethysmogram or ECG. The bottom shows the chat-interface to communicate with the patient. It is important to note that this backend client is only developed for visualization of our data and is not suitable to interpret the data correctly, since we use some simple scaling and plotting algorithms. The message box in the lower-right corner shows what happens when a patient transmits a file: a pop-up appears asking the user if he wants to open the file or not.

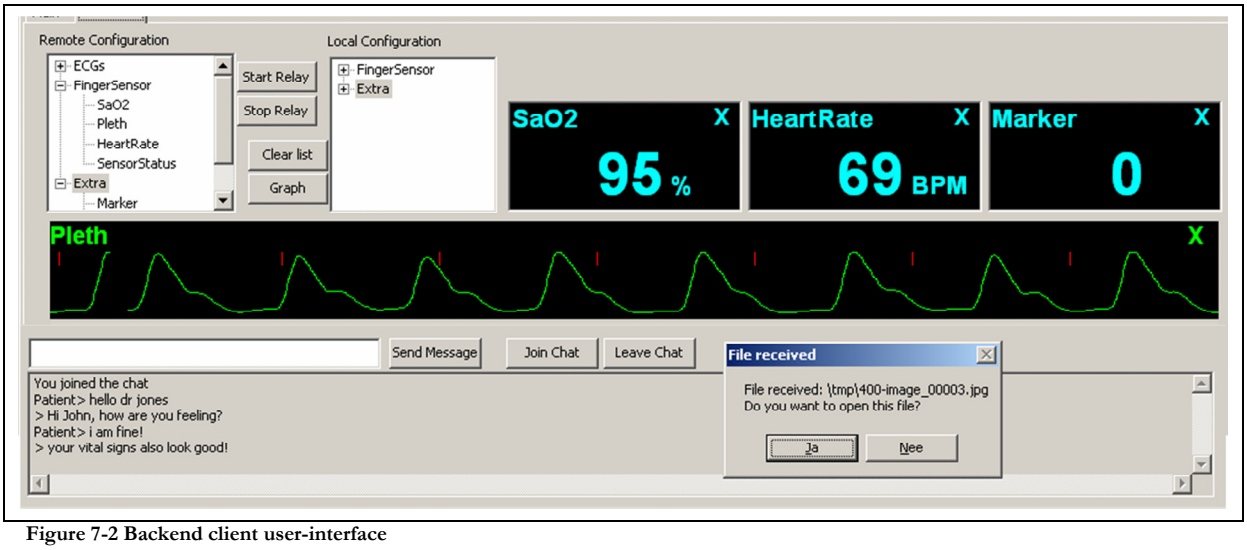


Figure 7-2 Backend client user-interface

7.2.3 Deployment

The deployment of a nomadic mobile service, like the MBU in the mobile e-health platform, is quite complex due to the amount of code downloading that the JINI surrogate architecture involves. This section tries to explain the basic deployment using the UML deployment diagram [BOO99] of the prototype in Figure 7-3.

The Mobile Base Unit has three main components; the Windows Mobile operating system, the IBM J9 virtual machine and the MBU software, which is part of our prototype implementation. The software uses the surrogate host (implemented by SUN as *Madison*) to create an MBUSurrogate, which is also part of our prototype implementation. Before the surrogate host can load the MBUSurrogate, it downloads the surrogate code from the eHealth codebase, the surrogate host implementation does not require any knowledge of the eHealth system, since all eHealth related code is downloaded from this codebase.

When the surrogate is loaded, it can communicate to the Mobile Base Unit using the HTTP interconnect protocol that is defined for the Mobile Service Platform. In addition, the MBUSurrogate loads the medical service and registers this in the lookup service. This lookup service is also implemented by SUN to facilitate the JINI architecture as *Reggie*. A Lookup Server runs this service.

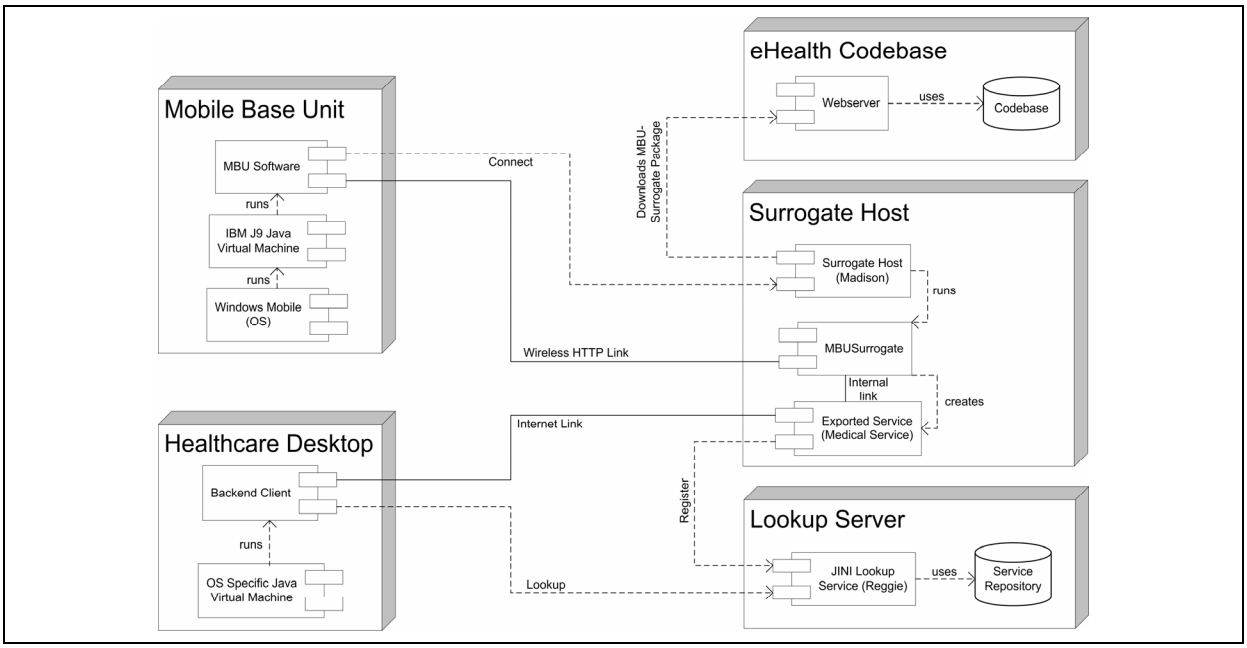


Figure 7-3 Prototype deployment diagram

The healthcare professional can run his Java backend client on any desktop operating system using a Java Virtual Machine, the backend client uses the lookup service to find the medical service and can then communicate to this service using an Internet link. The details of this communication have been discussed in detail in the previous chapter.

8 Conclusions

In this thesis we presented a design and discussed a prototype of a mobile e-health platform; this chapter focuses on evaluating our design and prototype and provides answers to our research questions posed in the first chapter. Furthermore we draw our conclusions on the main objective using the evaluation and research answers. The last section describes future work on this platform or in areas related to this platform.

8.1 Evaluation

This section evaluates the design in general terms, discussing the design rationales and the major difficulties encountered in our design. Furthermore, it discusses the prototype and looks into the evaluation criteria we determined in the introduction.

8.1.1 Design evaluation

The MobiHealth implementation is not suitable for the flexibility that is required for a mobile e-health platform. Due to its complex implementation structure it is a difficult, time-consuming and error-prone task to improve this implementation such that it supports the variety of scenarios as discussed in section 3.1. Therefore we chose to start designing the mobile e-health platform from scratch, using the MobiHealth implementation and experiences as a reference.

From the MobiHealth implementation we have learnt that it has too much component coupling, which reduces flexibility and inhibits making changes. Furthermore, the lack of design documentation is a problem, because it makes it difficult to understand the system. Besides these two problems, we have also found that the Mobile Service Platform and JINI Surrogate architecture provide a stable base for mobile service provisioning.

Based on the requirements for the mobile e-health platform, we presented a design that incorporates all essential requirements; this design is based on the JINI/Surrogate architecture, which is one of the few things borrowed from the MobiHealth design. Our design uses a totally different approach for internal sensor data propagation and added functionality to transport new types of data, like events and data blocks.

We focused on a modular design, which was enforced by separating concerns into components. This allows components to be designed and implemented separately; the design of the components presented in this thesis should form a basis for these components. By introducing several core components that are used at multiple locations, we reduce design and implementation effort and produce a more straight-forward design that allows future developers to understand the system more easily.

The device drivers and communicators at the MBU complement the modular design, ensuring the platform is independent of (medical) devices used in the scenarios and decouples the platform from a specific lower-level protocol. In addition, the plug-ins facilitate adding functionality independent of the core design, which enables application developers to implement the system with the functionality they require for a specific scenario. The device drivers, communicators and plug-ins can all be used for multiple scenarios, when similar functionality is required. We have designed our own structure for these pluggable components and implemented this partially in our prototype; however, since using standards is preferred, we recommend future developers to look at the applicability of the OSGi platform [OSGIWEB], which allows software components to be added, removed and updated on the fly.

The JINI architecture allows us to export (medical) services from the surrogate; the exported services framework described in this thesis decouples these services from the core platform allowing them to be designed independently for a specific scenario.

All in all, our design offers a fresh start with extended functionality and the required flexibility to support a variety of scenarios in comparison with the MobiHealth implementation. Appendix B lists the evaluation of the requirements and reveals that most requirements are incorporated in the design or supported by the plug-ins or exported services. The few requirements that are not satisfied should be tackled in future research and in the implementation phase.

Sensor data synchronization is embedded in the design, but not used at the MBU, because we have not foreseen any specific use for synchronized data at this location. In client applications, it is often necessary to display sensor data and thus this client application should contain synchronization. Defining a standard approach for data synchronization would be very useful and should also be considered as future work.

8.1.2 Prototype evaluation

The developed prototype shows that the concepts and components presented in this design can all be implemented using current technologies. The prototype also demonstrates the various data types (i.e., continuous, messages, blocks and events) that can be transmitted and the flexibility of this design with regard to sensor data that is relayed. Simple performance tests on CPU usage indicate that the prototype uses significant less processing power at the MBU than the MobiHealth implementation, which leaves more processing power for signal processing and other possible future components or can be used to reduce power consumption.

The prototype does not contain extensive error handling, so profound conclusions on the stability of the design can not be discussed. The prototype will not respond to disconnect by automatically reconnecting or disconnect from the sensor device. It also lacks a proper definition of events that can occur from within the core components and additional events need to be added as well.

While implementing the prototype, we found that the JINI surrogate architecture can be complex to understand; however there is sufficient documentation available. Due to time constraints, we have not implemented the exported service framework in the prototype; but we have learnt that such a framework is feasible from the multi-purpose service that we implemented.

We have tested the prototype using GPRS and both the sending of sensor data and the chat screen were fully functional; the file sending however, increased sensor data delay drastically and eventually caused disconnects for large files. An inefficient encoding mechanism in the Mobile Service Platform partly causes this problem; furthermore, data block encoding onto PDUs should be designed more thoroughly.

8.1.3 Reflection on evaluation criteria

In the first chapter, we posed several evaluation criteria to verify our design. The first four criteria were related to flexibility, since that is one of the focus points of our design. The last two criteria are related to resource usage of the MBU.

The first flexibility criterion was: "The designed platform must support the scenarios and trials that are defined for the MobiHealth and Awareness projects". The Awareness scenarios are described in section 3.1.1 and the MobiHealth trials can be found in Appendix C. Both Awareness scenarios are supported by the platform; although they do require certain plug-ins. The activity and alarm functionality is quite simple to implement and although digital signal processing requires more work, it is supported via the plug-in framework.

Most MobiHealth trials depend on vital sign transmission, which is basic platform functionality; there are several trials that require video transmission and some require location services. Video transmission is not considered in the design, but is supported via a plug-in that uses a sensor-channel to transmit video or it can be sent as data blocks. Location services are discussed in [ESU05] and can be included in the MBU as a plug-in as well.

The second flexibility criterion was "The design must support run-time flexibility with regard to enabling and disabling sensors on-the-fly"; which is included in the basic design and therefore considered to be supported by our platform. The third criterion "The design must include implementation flexibility, such that it is possible to use different sensors for each scenario and use scenario specific healthcare professional applications" is also supported with the device drivers and the exported services framework respectively.

The fourth criterion "When the design is implemented, it must support a very simplistic scenario, which can be seen as a light-weight version of the platform", is demonstrated in the prototype: it can send simple vital sign data and supports connecting to devices and backend network separately. However; the design itself leaves several aspects (like MBU behaviour after certain events) open that must be determined even for a light-weight version.

The first resource related criterion "The prototype implementation must be able to run at the device that is currently used to run the MobiHealth system" is demonstrated with the prototype as well. Our prototype runs at the Qtek9090 device that is currently used in Awareness demonstrations and for HS24 trials. Therefore we assume that a fully implemented platform can also run on these devices.

The last criterion "CPU usage of a light-weight platform in regular operation must be below 60 percent" is also verified using our prototype. In general operation, while relaying several sensor channels, the prototype uses less than 25 percent CPU, while MobiHealth uses over 50 percent. We have measured this using both an own developed CPU usage measuring tool as well as using a tool called SuperTasks [SUPER]. It is very unlikely that the core functionality that is not yet included in the prototype requires an additional 35 percent, and thus this

criterion is also met. Although we must stress that due to the unimplemented behaviour, the CPU usage may increase.

8.2 Conclusions

This section discusses the overall conclusions for this research; the first part tries to answer to our research questions and the second part presents the overall conclusions on our main objective.

8.2.1 On research questions

This section provides answers to our research questions, based on the evaluation of the previous section.

(1) What are the requirements for the mobile e-health platform that supports the main functionality that is required in a variety of scenarios?

The requirements are discussed in the third chapter, which provides the most complete answer to this question, we summarize our findings here. The main sources for the requirements were the various scenarios in the related projects and the bottom-up technical requirements that are imposed by the usage of mobile wireless communication technologies and portable devices. We have described the leading scenarios for the Awareness project and generalized them into four scenario categories; scenarios that are part of the same category often share part of their system behaviour. Using these scenarios, we defined several use-cases and used them to scope our requirements analysis. The requirements listed in section 3.3 were useful for the design of our platform and are evaluated in Appendix B.

We also shortly touched upon other requirements that are very important in the design of a mobile e-health platform, but could not be dealt with in the scope of this project. Security is very important, since the system deals with privacy sensitive patient data; we have delegated the design and implementation of security mechanisms to the communication layer and the GUI design, since these are the most likely points of attack. The legal and certifications are especially important for commercial deployment of such a platform; a thorough analysis is required and section 3.5 gives a starting point for this analysis.

Since we have limited domain knowledge, we have scoped the requirements to work towards a facilitating platform that can be used by domain experts to produce a scenario specific product. The requirements are also focused on such a platform. For each scenario, an end-user requirements analysis by domain experts is still required; this will help the scenario developers especially with regard to user-interface design and MBU behaviour implementation. Domain knowledge is also required for the design of a back-end application; for both functionality and its user-interface design.

(2) To what extent can the current design and implementation of the existing MobiHealth system be used to work towards a design and implementation of a mobile e-health platform that supports the required flexibility?

Our analysis of the MobiHealth system shows that its top-level design was very usable; the Mobile Service Platform solves various challenges in mobile service provisioning. However, the internal component design was very complex and unstructured. Combined with the lack of documentation and the fact that efficient transferring of discrete data was not included, we decided to only use the top-level design for the design of our platform. Other components were unusable as such, although we did use the implicit design knowledge of the system for

our refined design. The main concepts of MBU, BAN, surrogate and backend are reused, but their internal implementation is changed drastically. We have not tried to incorporate the PortiLab application in our design; however, a backend client similar to the current BSS may provide this legacy support.

The platform design as presented in this thesis offers many possibilities, due to its flexible structure. The plug-in mechanism, combined with the three data-types (sensor data, events and data blocks) and the exported services framework allow future developers to add and change functionality according to scenario requirements. Based on the discussion of the first evaluation criterion in the previous section, we state that sufficient flexibility is built in to support a variety of scenarios.

There are several major improvements with regard to the MobiHealth platform, Table 8-1 lists these improvements.

- Improved internal structure and use of a clear component definition;
- Added (unicast and broadcast) event and data block types, for efficient sending of discrete data;
- Increased communication possibilities for application developers;
- Different approach to sensor data distribution by means of a sensor bus; from push-mechanism to push-pull;
- Added plug-in framework to add functionality (like signal processing or sensor data storage) more easily;
- Allow simple exposing of services to clients using the exported services framework;
- Increased communication capabilities from backend to MBU using events;
- Improved flexibility allows easier development of a scenario specific implementation.

Table 8-1 Improvements with regard to the MobiHealth implementation

The prototype demonstrates the capabilities of our design and illustrates these capabilities by showing the use of the various data types. Sensor data channels can dynamically be switched on and off and illustrate how to use the sensor bus and its relaying components. The chat capabilities show how events can be used for small discrete messages and the file sending component demonstrates the usage of the data blocks.

(3) Is the design suitable for the scenarios as defined in (1) and what is necessary to implement a specific scenario?

We have evaluated the usage of the platform in the previous section and the mentioned scenarios are all supported, although some work is still required for all of them. Since the requirements were based on these scenarios, the evaluation in Appendix B also gives a good indication to what extent our design is suitable.

To determine what future work is required on the platform to allow a specific scenario to be implemented, we distinguish two developer roles, *platform developers* and *scenario developers*. Platform developers continue to work on this platform and work towards the stable base platform that can be used for the variety of scenarios; we have identified the following tasks for these developers:

- **Design core components:** The core components that this thesis discusses are not designed in full detail; for the majority we have specified some black box behaviour and a possible interface. Due to certification requirements, sufficient design documentation must be available for all components;
- **Define platform events:** System behaviour depends on the internal events that components generate, these events are also used for status information towards both patient and healthcare professional. The platform designers must define the events that each component can generate and explain its purpose; the events should be used for erroneous situations, component state transitions et cetera;

- **Develop basic flexible components:** Since many scenarios share functionality, the platform developers may choose to implement a set of plug-ins, device drivers, exported services and communicators that are commonly used. Good examples are a signal processing plug-in, the Mobi device driver and an exported service for sensor data;
- **Define quality assurance plan:** To ensure the system can be deployed commercially, certification is required; for the European Union, having a well-defined and approved quality assurance plan on software development is a part of this certification process.

Scenario developers will use the base platform to implement a mobile e-health solution for a specific scenario by adding scenario specific functionality and by defining scenario specific site behaviour. The scenario developers require a functional implementation of the base platform, which is implemented by the platform developers.

- **Define scenario specific behaviour:** The scenario specific behaviour is related to the scenario characteristics and mainly deals with what to do on disconnect; when to send sensor data and et cetera. The scenario developer must define this in terms of events and implement this in the SiteCore components, like the MBUCore.
- **Implement scenario specific functionality:** Many scenarios require a specific back-end application; or MBU functionality, like a special signal processing algorithm.
- **Develop a graphical user interface:** Most scenarios require a different user-interface for the MBU; the design and implementation strongly depends on the user and the exposed functionality. Sometimes, only showing status information suffices, while for other scenarios detailed interaction with the system is required.
- **Define intended use:** Certification requires a declaration of intended use for all medical devices; this defines what the system should be used for, what it can do and also what it should not be used for. For example, when the system is used to monitor ECG signals, but will not support defibrillation, the intended use must clearly specify this.

8.2.2 On the main objective

In the first chapter, we stated our main objective as follows:

> "*Design a mobile e-health platform that can be implemented and deployed using available technologies and contains sufficient run-time and implementation flexibility to support a variety of scenarios in which remote patient monitoring can increase quality of care, using the design of the MobiHealth system as a reference.*"

Based on the evaluation of the design and the answers to our research questions; we can say that we have successfully designed a platform that is suitable for the variety of scenarios. The flexible components give flexibility on several axes: the first is *functionality*, plug-ins allow adding functionality like signal processing or GPS location determination without changing the core platform, combined with an exported service, this functionality can also be exposed to backend clients. The second axis is *device connectivity*; using the device drivers, different devices can be connected to the system independent of their connectivity means or protocol. The third axis, *network connectivity*, allows easy integration with specific network protocols or changing encoding schemes for a different network type; it also allows different security schemes to be used. The last axis, *exposed services*, ensures that the backend client communication can be specified per scenario and that a variety of backend clients can be developed.

MobiHealth proved to be useful for the top-level design as well as implicit design knowledge on such a platform; unfortunately, its complex structure and lack of documentation may have contributed to loss of some interesting design knowledge. In this thesis, we have tried to record our design choices and design knowledge, allowing developers to understand the structure of the platform such that they can start implementing for it after only a short study.

The clear structure of the design, in which functionality is divided over a number of components, allows allocation of tasks. Research projects like Awareness, where multiple parties are involved in the design and implementation of components can benefit from this.

A certain amount of work is still required on both the base platform as for each specific scenario; in the course of time, a number of dynamic components may become available that, combined with a fully implemented base platform, enables fast composition of a scenario specific implementation.

8.2.3 Concluding remarks

This section lists some concluding remarks that are related to this research, but fall outside the scope of our conclusions.

- The design of this platform is based on the usage of (unstable) wireless communication technologies and connectivity to the backend cannot be guaranteed; therefore we state that this platform must always be used as an addition to treatment and can never be used for critical patients;

- Some Awareness scenarios and deliverables discuss the usage of actuators; we have not included actuators as such in our design. If these rely on a stream of data for controlling the actuators, some inverse relaying mechanism must be added that allows this stream to be published on the sensor bus of the MBU from the origin; if the actuators work with commands, the event mechanism can be used.

- During the course of this research, the HS24 and Awareness projects continued work on the MobiHealth system and several improvements have been made. We have not included these improvements in our system analysis.

8.3 Future work

This section gives some interesting areas for future work that were outside the scope of this thesis. Future work on the platform itself and for each scenario is discussed in section 8.2.1 and will not be repeated here. The first part describes the work that is left out due to time constraints, while the second part of this section focuses on future work that is related to this research in general.

8.3.1 On this design

An interesting task that is left for future work is analyzing the sensor data synchronization. We introduced a few concepts on how to encode synchronization information in the sensor data stream, and their efficiency and usability must be tested.

We did not define how data blocks should be encoded onto PDUs. This is an essential task, since data blocks are included in the core functionality part of the platform. Our prototype contains a preliminary implementation of this encoding, but is not studied thoroughly.

The sensor bus is internally based on pipes, there are several strategies available for implementing these pipes and these all have their advantages and disadvantages. It is important to compare these strategies and pick the best pipe mechanism for the sensor bus.

The CommunicationManager in the MBU has not been designed in detail; a CommunicationManager that is able to switch between various networking technologies, based on availability and context input is very desirable, especially within the Awareness project. Preliminary research showed that such functionality is supported by the Linux operating system [PED05], so new research should focus on Windows Mobile.

8.3.2 In research area

A more refined prototype should be tested thoroughly with regard to performance, stability and scalability. According to [DOK03], the interconnect protocol as used in the Mobile Service Platform can support at least 100.000 BANs, when proper load-balancing schemes are used. Although testing for 100.000 may not yet be necessary, it would be interesting to see what the maximum amount of surrogates per surrogate host is. This performance study can also be aimed at determining internal component performance and requirements, which may be used to prioritize threads and streams in the system.

Although quite different, but very interesting, is researching how real-time sensor data can be processed by a Data Stream Management System (DSMS) and how this can be integrated into a backend client for this platform. These systems are like database systems, but focus on processing data streams. Since sensor data is also a data stream, these systems may be used for large-scale sensor data signal processing to detect alarms, for which a DSMS can generate events [CHE04].

Similar to DSMS is another new technology, Data Distributions Service (DDS), a DDS is a middleware technology that deals with real-time data distribution and is focused on a publish/subscribe approach [PAR05]. The DDS can be used to distribute sensor data to multiple listeners and provide a generic approach to subscribing for this data. If a large number of sensor data listeners are required, this technology may offer certain benefits.

Instead of the component-based approach we implemented ourselves, interesting future work lies in analyzing the possibilities of the OSGi framework and other existing component models. These models may allow run-time changing of specific components, such that the data streams are not interrupted.

The IEEE 1073 standard defines communication between medical devices and may play an important role in device driver design or integration of the platform with external hospital systems.

Acronyms

API	Application Programming Interface
BAN	Body Area Network
BDR	BAN Data Repository
BSS	BAN Streaming Service
ECG	Electrocardiogram
FIFO	First-in, First-out
FSM	Finite State Machine
GCF	Generic Connection Framework
GPRS	General Packet Radio Service
GPS	Global Positioning System
GUI	Graphical User Interface
HCP	Healthcare professional
HS24	HealthService 24 project
HTTP	Hypertext transfer protocol
IDE	Integrated Development Environment
LLS	Lower-level service
LUS	Lookup service
MBU	Mobile Base Unit
MSP	Mobile Service Platform
NMS	Nomadic Mobile Service
PDA	Personal Digital Assistant
PDU	Protocol Data Unit
RMI	Remote Method Invocation
SOA	Service Oriented Architecture
SWT	Standard Widget Toolkit
TCP	Transmission Control Protocol
TMSi	Twente Medical Systems International
UML	Unified Modelling Language
UMTS	Universal Mobile Telecommunications System
VAN	Vehicle Area Network
XDR	External Data Representation
XML	Extensible Markup Language
XML-RPC	XML - Remote Procedure Call protocol

Bibliography

[AW/D1.1] van Sinderen, Batteram, Meeuwissen et al; D1.1v2: Scope and scenarios; Freeband Awareness Project; June 2005.

[AW/D4.7] Broens, Halteren, Jones, Shishkov, Widya; D4.7: Modelling Body Area Networks – from technologies to models to components; Freeband Awareness Project; December 2005.

[BAR99] Barro, Presedo, Castro, Fernández-Delgado, Fraga, Lama, Vila; Intelligent telemonitoring of critical-care patients; IEEE Engineering in Medicine and Biology magazine; July/August 1999.

[BOO99] Booch, Rumbaugh, Jacobson; The Unified Modeling Language Reference Manual; Addison-Wesley, 1999. ISBN: 0-201-30998-X

[BOR01] Borking, Van Blarkom; Protection of personal data; Registratiekamer; 2001. ISBN:90-74087-27-2
Available at: http://www.cbpweb.nl/downloads_av/AV23.pdf

[BRO03] Broens; Bandwidth optimization of the TMSi fibre protocol; Internship report TMS International; November 2003.

[BUL05] Bults, Wac, van Halteren, Nicola, Konstantas; Goodput Analysis of 3G wireless networks supporting m-health services; Proceedings of ConTEL 2005 - 8th International Conference on Telecommunications, 15-17 June 2005, Zagreb, Croatia.

[CHE04] Chen, Agrawal, Cochinwala, Rosenbluth; Stream Query Processing for Healthcare Bio-sensor Applications; Proceedings of the 20th International Conference on Data Engineering (ICDE04), 30 March – 2 April 2004, Boston, USA.

[DOC04] Dockhorn Costa, Ferreira Pires, van Sinderen, Pereira Filho; Towards a Service Platform for Mobile Context-Aware Applications; In Proceedings of the First International Workshop on Ubiquitous Computing - IWUC 2004, April 2004, Portugal.

[DOK03] Dokovsky, van Halteren, Widya; BANip: enabling remote healthcare monitoring with Body Area Networks; FIDJI 2003 International Workshop on Scientific Engineering of Distributed Java Applications; 27-28 November, 2003, Luxembourg.

[ESU05] Schoot-Uiterkamp; Positioning Services for a Mobile Services Platform; Master's thesis Telematics - University of Twente; August 2005.

[FDA99] U.S. Department Of Health And Human Services, Food and Drug Administration, Center for Devices and Radiological Health; FDA Guidance: Off-The-Shelf Software Use in Medical Devices - #585; 9 September 1999.
Available at: http://www.fda.gov/cdrh/ode/guidance/585.pdf

[FDA05] U.S. Department Of Health And Human Services, Food and Drug Administration, Center for Devices and Radiological Health; Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices; 11 May 2005.
Available at: http://www.fda.gov/cdrh/ode/guidance/337.pdf

[FRI01] Fries; Handbook of Medical Device Design; Marcel Dekker Inc; 2001. ISBN: 0-8247-0399-5

[GAM94] Gamma, Helm, Johnson, Vlissides; Design Patterns – Elements of reusable Object-Oriented Software; Addison-Wesley; October 1994. ISBN: 0-201-63361-2

[GAN04] Ganesh; E-health - drivers, applications, challenges ahead and strategies: a conceptual framework; Indian Journal of Medical Informatics; May 2004.

[HER95] Herrmann, Zier; Using IEC 601-1-4 to Satisfy FDA Software Guidance Requirements; Medical Devicelink, December 1995.
Available at: http://www.devicelink.com/mddi/archive/95/12/013.html#ref1

[HS/D3.1] HealthService24 project; D3.1: HealthService 24 – end user requirements; restricted project document.

[KON02] Konstantas, Jones, Bults, Herzog; MobiHealth – innovative 2.5/3G mobile services and applications for healthcare; Proceedings of the IST Mobile & wireless telecommunications Summit 2002.

[KON04] Kon et al; Design, implementation, and performance of an automatic configuration service for distributed component systems; Software – Practice and experience; Vol. 34, 2004.
Available at: http://www.ime.usp.br/~kon/papers/spe04.pdf

[KUL03] Kulak, Guiney; Use cases: requirements in context (2nd edition); Addison-Wesley; July 2003. ISBN: 0-321-15498-3

[KYR03] Kyriacou et al; Multi-purpose HealthCare Telemedicine Systems with mobile communication link support; BioMedical Engineering OnLine; Vol 2, 24 March 2003.

[LIN99] Lin; Applying telecommunication technology to health-care delivery; IEEE Engineering in Medicine and Biology magazine; July/August 1999.

[MAR03] Martí et al; Security in the MobiHealth BAN (D2.5); MobiHealth Project; July 2003.

[MDD03] Council of the European Communities; Medical device directive 93/42/EEC; November 2003; CONSLEG-1993L0042.

[MH/D1.3] MobiHealth project partners; D1.3: Detailed description of trial scenarios; MobiHealth project; v2, Februari 2003.

[MH/D5.1] Melander-Wikman, Jansson, Herzog, Konstantas, Scully; D5.1: Overall evaluation of the MobiHealth trials and services; MobiHealth project; v2.6, September 2004.

[MUR04] Murphy; Development And Deployment Using The Jini Network Technology Starter Kit - Tips, Tools, And Idioms; Seventh Jini Community Meeting, 23-25 March 2004, Massachusetts, USA.

[NEL97] Nelwan, Meij, Fuchs, van Dam; Ubiquitous access to real-time patient monitoring data; IEEE Computers in Cardiology proceedings; 7-10 September, 1997, Lund, Sweden.

[ORT03] Ortiz; The Generic Connection Framework; Sun Developer Network; August 2003.
Available at: http://developers.sun.com/techtopics/mobility/midp/articles/genericframework/

[PAR05] Pardo-Castellote; Data Dsitributions Service Standard Drives Power to the Edge; Mil/COTS Digest – feature article; March/April 2005.
Available at: http://www.milcotsdigest.com/Articles/2005/Mar-Apr/Real-Time/default.htm

[PED05] Peddemors, Eertink, Niemegeers; Communication Context for Adaptive Mobile Applications; In Workshop Proceedings of the 3rd Conference on Pervasive Computing - Middleware Support for Pervasive Computing Workshop; March 2005.

[PRE00] Pressman, Ince; Software Engineering: A practitioner's approach; 5th edition 2000. ISBN: 0-07-709677-0

[PRU04] Pruijn; Web services in the MobiHealth service platform; Bachelor thesis Telematics – University of Twente; July 2004.

[QUA01] Quartel, Pires; Implementation of Telematics Systems (lecture notes); University of Twente; November 2001.

[RFC1014] Sun Microsystems, Inc.; XDR: External Data Representation Standard; RFC-1014 Network Working Group; June 1987.
Available at: http://www.faqs.org/ftp/rfc/pdf/rfc1014.txt.pdf

[SAU02] Sauerwein, Linnemann; Manual for processing personal data – personal data protection act; Dutch department of justice; April 2002 (in Dutch).
Available at: http://www.justitie.nl/images/handleidingwbp_tcm74-38038.pdf

[SCH00] Schoch; An authentication and authorization architecture for JINI services; Technical University of Darmstadt; October 2000.
Available at: http://www.vs.inf.ethz.ch/publ/papers/da-schoch.pdf

[SOM03] Sommers; Jini Starter Kit 2.0 tightens Jini's security framework; Javaworld – Jiniology; May 2003.
Available at: http://www.javaworld.com/javaworld/jw-05-2003/jw-0509-jiniology.html

[SSR00] Schmidt, Stal, Rohnert, Buschmann; Pattern-oriented software architecture: Patterns for concurrent and networked objects; Wiley & Sons, 2000. ISBN: 0-471-60695-2

[SUN99] Sun Microsystems; Jini Technology Architectural Overview; Sun Microsystems Inc.; January 1999.
Available at: http://www.sun.com/software/jini/whitepapers/architecture.pdf

[SUN03] Sun Microsystems; Jini Technology Surrogate Architecture Specification 1.0; Sun Microsystems Inc.; October 2003.
Available at: http://surrogate.jini.org/

[THI05] Thiele; Commercialization of MobiHealth: a technical feasibility study; Internship report Yucat B.V.; February 2005.

[TOL05] van Tol; Design and development of LEGO Mindstorms based nomadic services; Master's thesis Telematics – University of Twente; December 2005.

[VIS02] Vissers, Pires, Quartel, Sinderen; The architectural design of distributed systems (lecture notes); University of Twente; March 2002.

[WAC05] Wac, Bults, van Halteren, Konstantas, Nicola; Measurements-based performance evaluation of 3G wireless networks supporting m-health services, proceedings of Twelfth Annual Multimedia Computing and Networking (MMCN '05), 19-20 January, 2005, San Jose, California, USA

[WID03] Widya, van Halteren, Jones, Bults, Konstantas, Vierhout, Peuscher; Telematic requirements for a mobile and wireless healthcare system derived from enterprise models; 7th International Conference on Telecommunications ConTEL 2003; 11-13 June, 2003, Zagreb, Croatia.

[YUC05] Backx; MobiHealth – current system analysis; Internal document – Yucat Mobile Business Solutions; July 2005.

[ZAJ99] Zajtchuk, Gilbert; Telemedicine: A new dimension in the practice of medicine; Disease-a-month Vol.45 Nr.6; June 1999.

All used URLs verified 10 February 2006

Web references

[AAMIWEB]	*Association for the Advancement of Medical Instrumentation - AAMI Standards website* http://www.aami.org/standards/
[AWARENESS]	*Freeband Awareness – Project website* http://awareness.freeband.nl
[CENELEC]	*European Committee for Electrotechnical Standardization - CENELEC Website* http://www.cenelec.org
[HS24]	*HealthService24 – Project website* http://www.healthservice24.com
[JINIWEB]	*SUN Microsystems - Jini Network Technology website* http://www.sun.com/software/jini/
[JSR197]	*Java Community Process – JSR197: Generic Connection Framework Optional Package for the J2SE Platform* http://www.jcp.org/en/jsr/detail?id=197
[JSR82]	*Java Community Process – JSR82: Java APIs for Bluetooth* http://www.jcp.org/en/jsr/detail?id=82
[KXML]	*kXML – Website* http://kxml.sourceforge.net
[MOBIHEALTH]	*MobiHealth – Project website* http://www.mobihealth.org
[OSGIWEB]	*OSGi Alliance – OSGi Technology website* http://www.osgi.org/osgi_technology/
[RRD]	*Roessingh Research and Development – Corporate website* http://www.rrd.nl/
[SUPER]	*Software and Son – SuperTasks product website* http://www.softwareandson.com/SuperTasks/
[TMSI]	*Twente Medical Systems International BV – Corporate Website* http://www.tmsi.com
[XMLRPC]	*XML-RPC.Com – Website on the XML-RPC specification* http://www.xmlrpc.com
[YUCAT]	*Yucat Mobile Business Solutions B.V. – Corporate website* http://www.yucat.com

All used URLs verified 10 February 2006

Appendices

Appendix A Example configuration

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- MBU Configuration file -->
<MBU>

    <!-- Patient information -->
    <Patient>
        <ID>MST-10-1000</ID>
        <Name>John Doe</Name>
        <Phone>06-12345678</Phone>
        <Physician>
            <Name>Dr. James Johnson</Name>
            <Hospital>Medical Center Townsville</Hospital>
            <License>1234</License>
        </Physician>
    </Patient>

    <!--
        Communication properties, communicators will be loaded according to priority,
    -->
    <Communication>
        <Communicator name="YCT-MSP" communicator="msp" priority="1">
            <!-- Local @ Yucat -->
            <SurrogateJar>http://10.0.0.7/msp/eHealthSurrogate.jar</SurrogateJar>
            <JiniSH>jinish://10.0.0.7</JiniSH>
            <StreamTranscoding>
                <!-- Deflater compresses the outputstream, by retrieving
                      chunks of data, and then zipping them
                   -->
                <ChainElement type="deflate">
                    <ZipLevel>4</ZipLevel>
                    <BufferSize>8192</BufferSize>
                </ChainElement>
            </StreamTranscoding>
        </Communicator>

        <Communicator name="Remote-MSP" communicator="msp" priority="2">
            <!-- Remote @ Yucat -->
            <SurrogateJar>http://yucat.mobihealth/msp/eHealthSurrogate.jar</SurrogateJar>
            <JiniSH>jinish://yucat.mobihealth</JiniSH>
            <StreamTranscoding>
                <ChainElement type="deflate">
                    <ZipLevel>4</ZipLevel>
                    <BufferSize>8192</BufferSize>
                </ChainElement>
            </StreamTranscoding>
        </Communicator>

    </Communication>

  <!-- BAN (SensorNetwork) configuration -->
    <DeviceConfig LastUpdate="200602051530">

       <!-- inactive Dummy device driver -->
        <Device active="false" driver="dummy" file="/dummy_input_file.xml" />

       <!-- Active Mobi device driver configured for Yucat´s Mobi4
            Discrete=”true” indicates that the channel will be displayed as a discrete vital
              sign in the backend application
        -->
        <Device active="true" driver="mobi" Type="mobi4-3e1as" Brand="TMSi" Serial="924030032"
BaseFrequency="1024" Frequency="128" BTStreamer="d0i2h4i3h3" MAC="00:A0:96:1D:86:FB">
            <Channel Name="ECGs" Frequency="128" Active="true">
                <Sensor Name="ECG1" ByteLength="3" Type="1" Unit="uV" Resolution="0.0715"
Min="-6000" Max="6000" Highpassfilter="0.8" Active="true"/>
                <Sensor Name="ECG2" ByteLength="3" Type="1" Unit="uV" Resolution="0.0715"
Min="-6000" Max="6000" Highpassfilter="0.8" Active="true"/>
                <Sensor Name="ECG3" ByteLength="3" Type="1" Unit="uV" Resolution="0.0715"
Min="-6000" Max="6000" Highpassfilter="0.8" Active="true"/>
            </Channel>
```

```
            <!-- Yucat does not own a Resp sensor, so disabled -->
            <Channel Name="Resp" Frequency="128" Active="false">
                <Sensor Name="Resp" ByteLength="3" Type="3" Unit="uV" Resolution="1.4305"
Min="-10000" Max="10000" Highpassfilter="5" Active="false"/>
            </Channel>

            <Channel Name="FingerSensor" Frequency="128" Active="true">
                <Sensor Name="SaO2" ByteLength="1" Type="11" Unit="%" Resolution="1" Min="90"
Max="100" Discrete="true" Highpassfilter="0" Active="true"/>
                <Sensor Name="Pleth" ByteLength="1" Type="11" Unit="" Resolution="1" Min="0"
Max="255" Highpassfilter="0" Active="true"/>
                <Sensor Name="HeartRate" ByteLength="1" Type="11" Unit="BPM" Resolution="1"
Min="30" Max="150" Discrete="true" Highpassfilter="0" Active="true"/>
                <Sensor Name="SensorStatus" ByteLength="1" Type="11" Unit="" Resolution="1"
Min="0" Max="255" Discrete="true" Highpassfilter="0" Active="true"/>
            </Channel>

            <Channel Name="Extra" Frequency="128" Active="true">
                <Sensor Name="Marker" ByteLength="1" Type="9" Unit="" Resolution="1" Min="0"
Max="1" Highpassfilter="0" Discrete="true" Active="true"/>
                <Sensor Name="SawTooth" ByteLength="1" Type="10" Unit="" Resolution="1"
Min="0" Max="65" Highpassfilter="0" Active="true"/>
            </Channel>

        </Device>

    </DeviceConfig>

<!-- Each element in here will be parsed and the plugin will be loaded if active -->
    <Plugins>
       <!-- Demo plugin that just logs the 'initialization calls' -->
        <TestPlugin active="true" class="ehealth.mbu.plugin.TestPlugin"
jarfile="/eHealth/plugin/test.jar" />

       <!-- Our demonstrator GUI, implemented as plugin, not as core element! -->
        <DemoGUI active="true" class="ehealth.mbu.gui.SwtFrame" />
    </Plugins>

    <Surrogate coresleep="1500">
        <!--
            Surrogate Configuration for this specific MBU, currently unused,
            coresleep property is used for testing purposes
        -->
    </Surrogate>

    <Core>
        <!-- Can be used to define MBU Core specific configurations (unused currently) -->
    </Core>

</MBU>
```

Appendix B Evaluation of requirements

The table below discusses an evaluation of the requirements from chapter three. The satisfied column indicates whether the design presented in this thesis either implements the requirement; or provides adequate functionality to implement this requirement as a plug-in.

Req	Evaluation	Satisfied
Generic requirements		
1.1	Gathering is handled by the device drivers and communication by the relay service between MBU/Surrogate and by a (yet to be implemented Medical JINI Service for surrogate/client communication. The sensor bus facilitates all these components by providing a generic approach for sensor data distribution within components.	Yes
1.2	The event bus can be used to facilitate the messages and notifications; while the event relay service is responsible for delivering the events. Exact messages and notifications depend on the scenario and should be implemented in MBU GUI, possibly using plug-ins.	Yes
1.3	The event bus also facilitates these messages and notification.	Yes
1.4	Although a priority is included in the event type; it is not used within the event bus, since the event handling occurs in the publisher's thread. The priority is also ignored for communication; which should actually be changed by adding a priority to a PDU; if the communicator supports prioritization, it can use this field in the PDU.	No
1.5	The data block manager facilitates bi-directional communication of data objects; although a plugin is often required to handle these events.	Yes
1.6	The device drivers, plugins and the exported service framework all contribute to the flexibility of the platform. Also the generic communication framework allows changing of lower level protocols, without the need to redesign other components.	Yes
1.7	The component-interface and component repository allow this modular approach.	Yes
1.8	Temporary loss of connection is partly handled by the Mobile Service Platform; however, the platform itself does not support connection loss. This can be either defined in the communicators, moving responsibility to a lower level; or it can be implemented in the relay components.	Part
1.9	The sensor relay service allows on-the-fly enabling of sensors and channels, but exact behaviour should be specified for each scenario.	Yes
MBU Requirements		
2.1	The relay services ensure that only required data is transmitted; therefore optimizing bandwidth usage. Further optimization is often protocol dependent and should be included in the communicators; for example, data compression.	Yes
2.2	The GUI can be included as a plug-in, allowing easy replacement. Furthermore, since all components have access to the event bus and sensor bus; they have access to all required information.	Yes
2.3	Status information is not specifically included in the design, although several components maintain their state. By listening to events that affect overall system status, the GUI can display the required amount of status information.	Yes

2.4	This behaviour is not implemented, but responsibilities are moved for a plug-in component. This components can use the block transfer method to transmit the sensor data.	Yes
2.5	The generic communication framework allows the underlying protocol to change without affecting other parts in the system.	Yes
2.6	The design does not support seamless switching yet; the communication manager does provide a place to implement this behaviour. The biggest problem is the loss of data that is in the communication buffers.	No
2.7	The device drivers allows a generic approach to device communication, allowing device manufacturers to write their own device driver and use different devices at the same time.	Yes
2.8	Status information can be seen as a system generated message and is covered by the event bus and event relay component (see also requirement #1.3).	Yes
Backend requirements		
3.1	Delay calculations are not included in the design; however, the design supports estimating the delay by using timestamps as sensor data marks and a plug-in to synchronize clocks. Furthermore, status information on buffers can give an indication on system delay.	No
3.2	Sensor configuration objects can be serialized and transmitted using the block transfer mechanism.	Yes
3.3	Sensor data should be offered via an exported service; such a service is not designed thoroughly and is left for future work; however, for prototype testing, we have included this demonstrated that it is possible.	Yes
3.4	Again, this must be implemented in the exported service; however, the event bus and relay components facilitate these notifications.	Yes
3.5	Although this depends on the implementation of the exported service; the surrogate design supports multiple subscribers.	Yes
Non-functional requirements		
4.1	The prototype uses less than 25 percent CPU in normal (relaying) operation. We assume that the required extra functionality, like error handling, does not require much extra CPU power.	Yes
4.2	The design does not discuss buffering, the only buffer implicitly design is located in the sensor bus (to allow concurrent reading and writing). Its properties are not configurable by config; however, this should be easy to implement. We did discuss some pipe-properties that are important.	No
4.3	The JINI surrogate architecture solves these scalability issues; adding an extra surrogate host and/or lookup server only requires changes in configuration. More scalability issues are expected for a sensor data repository backend system.	Yes
4.4	Remote device manageability can be implemented using an exported service; we have not analysed this for commercial solutions.	Part
Application developer requirements		
5.1	See evaluation of requirement #1.7; on-the-fly replacement of components is not discusses in this design; however, offline replacement is possible.	Yes
5.2	The device drivers approach supports this.	Yes
5.3	The generic communication framework provides this API.	Yes

Appendix C MobiHealth trial descriptions

The following trial descriptions are taken from MobiHealth deliverable D1.3, which is available at http://www.mobihealth.org.

Telemonitoring of patients with ventricular arrhythmia

The target group in this trial are patients with ventricular arrhythmias who are undergoing drug therapy. Cardiac arrhythmias are very common, especially in elderly patients, and in many cases are related to coronary heart disease. Around one million patients suffer from coronary heart disease in Germany today.

In patients suffering from arrhythmia, ECG measurements have to be taken regularly to monitor efficacy of drug therapy. In order to save time and reduce costs, in this trial the patient is able to transmit ECG and blood pressure via GPRS from home or elsewhere to the health call centre, where the vital signs are monitored by cardiologist. The intention is that irregular patterns in these vital signs will be quickly detected and appropriate intervention can be effected. The trials are designed to determine whether 2.5-3G wireless communications can support such services.

The Lighthouse Alarm and Locator Trial

The target group involved in the trials are patients at the Lighthouse care resource centre and also clients living at home, but with the common characteristic that all have an alarm system located in their room at the Lighthouse Centre or in their home. The current system does not allow the patient any freedom related to mobility and forces the patient to be trapped at home or in their room at the Centre. By replacing the fixed alarm system with the mobile MobiHealth system the patient can move freely anywhere. Additionally positioning and vital signs are monitored and video communication is planned when UMTS is available. The main expected benefit of using the MobiHealth BANs in this trial is to increase mobility and to allow patients to lead a more normal life than they did before.

The purpose of the Lighthouse trial is to test the effectiveness of the new GPRS/UMTS-based alarm and locating device (a variant of the MobiHealth BAN) according to several determining factors: safety, convenience, empowerment of user, mobility of user and improvement in efficiency of care given.

Physical activity and impediments for activity in women with RA

Trial subjects will be women with Rheumatoid Arthritis. The use of the BAN together with the 2.5–3 G wireless communications will enable collection of a completely new kind of research data which will enhance understanding of the difficulties and limitations which these patients face. The objective is to find solutions that will make their lives easier. By this collection of data, the scarce knowledge about what factors impede normal life will be supplemented and quality of life of RA patients may thereby be improved.

By use of the BANs, the activity of the patients will be continually monitored. Parameters measured include heart rate, activity level, walking distance and stride length.

Monitoring of vital parameters in patients with respiratory insufficiency

The group of patients involved in the trial suffer from respiratory insufficiency due to chronic pulmonary diseases. These people need to be under constant medical control in case they suffer an aggravation of their

condition. Besides needing regular check ups, they also need oxygen therapy at home, which means oxygen delivery and close supervision.

The use of the MobiHealth BANs in this trial is designed to help in the early detection of this group of diseases and also to support homecare for diagnosed patients by detecting situations where the patient requires intervention. The expected benefits are a reduction in the number of checks-ups and hospitalisations needed thus saving both time and money for the hospital. Parameters measured are pulse rate, oxygen saturation and signals from a motion sensor (accelerometer).

Home care and remote consultation for recently released patients in a rural area

Home care services and the possibility of monitoring health conditions at a distance are changing the way of providing care in different situations. If suitable home-based services are provided, patients do not need to be in hospital when the risk is not very high or when they are recovering from an intervention. By investing in home care, hospitals have been able to significantly reduce pressure on beds and on staff time dedicated to the kind of patients named above.

This trial tests transmission of clinical patient data by means of portable GPRS/UMTS equipment from patients living at home in a rural low population density area, to a physician or a registered district nurse (RDN). The subjects are patients who have been recently discharged from hospital. The expected benefit is that this intervention will reduce the number of cases where the patient is moved to hospital unnecessarily. Parameters measured include blood pressure, heart rate, oxygen saturation and blood glucose.

Support of home-based healthcare services

This trial involves use of GPRS for supporting home-based care for elderly chronically ill patients including remote assistance if needed. Patients are suffering from co-morbidities including COPD. The MobiHealth Nurse-BAN will be used to perform patient measurements during nurse home visits and the MobiHealth patient.

BAN will be used for continuous home monitoring outdoors during patient rehabilitation. It is very important to facilitate patients' access to healthcare professionals without saturating the available resources, and this is one of main expected outcomes of the MobiHealth remote monitoring approach.

Parameters to be measured are Oxygen saturation, ECG, spirometry, temperature, glucose and blood pressure.

Outdoors patient's rehabilitation

The patients involved in this trial are chronic respiratory patients who could benefit from rehabilitation programs to improve their functional status. The study aims to check feasibility of remotely supervised outdoors training programs based on control of walking speed enabled by use of the BAN. The physiotherapist will receive on-line information on patient's exercise performance and will provide feedback and advice.

It is expected that by enabling patients to perform physical training in their own local settings, the benefits, in terms of cost and social acceptance, can be significant.

Parameters to be measured are pulse oximetry, ECG and mobility with audio communication between patient and remote supervising physiotherapist.

Tele Trauma Team

MobiHealth BANs will be used in trauma care both for patients and for health professionals (ambulance paramedics). The trauma patient BAN will measure vital signs which will be transmitted from the scene to the

members of the trauma team located at the hospital. The paramedics wear trauma team BANs which incorporate a video camera, an audio system and a wireless communications link to the hospital.

The purpose of this trial is to evaluate whether use of 2.5-3G communications can improve quality of care and decrease lag-time between the accident and the intervention. When using telemetry technology, time can be saved and thus treatment and chances for patient recovery improved. Faster intervention is expected to increase survival rates and decrease morbidity.

Parameters to be measured are breathing frequency, oxygen saturation, pulse rate, blood pressure, pupil size and reactions, amount of fluids infused. Video from the scene will be transmitted assuming UMTS availability.

Integrated Homecare in women with high-risk pregnancies

The trial will use the MobiHealth BAN to support Integrated Homecare for women with high-risk pregnancies.

Women with high-risk pregnancies are often admitted to the hospital for longer periods of time because of possible pregnancy-related complications. Admission is necessary for the intensive monitoring of the patient and the unborn child. Homecare with continuous monitoring of women with high-risk pregnancies, when feasible, is desirable and can postpone hospitalisation and reduce costs.

In this trial, patients are monitored from home using the MobiHealth BAN and the (maternal and foetal) biosignals are transmitted to the hospital. The objective of the trial is to evaluate if such monitoring services can be supported by 2.5-3G communications such that hospitalisation can be postponed and costs reduced.

Appendix D Prototype packaging

This appendix gives an overview of the packages used in the prototype, including a short description of what kind of classes it contains.

ehealth.shared.

communication	Generic communication classes; the communicator, transmitter and receiver interface
components	Component related classes, including the generic component interface and the ComponentRepository core component
configuration	Configuration parsing related classes
configuration.desc	Description classes that define a specific configuration object
core	SiteCore class and related core classes
datablock	DataBlock related classes, the datablock type itself and DataBlockManager
eventbus	Event bus related classes, including the EventBus core component
eventbus.events	The event data types (Event and MessageEvent)
eventbus.listeners	The listener-interfaces for the event bus
interfaces	Several shared interfaces for backend-clients
msp.pdu	PDU data types; PDUs are currently related to the MSP messaging scheme, this should be decoupled as future work, recommendations are given in code comments
sensorbus	Sensor bus related classes, including the input/outputstream classes and the SensorBus core component
sensormodel	BAN model descriptions and factory methods, for defining sensor and channel descriptions
util	Common use utilities and interfaces
util.io	IO related utilities (DeflaterOutputStream and InflaterInputStream)

ehealth.mbu.

communication	MBU specific communication classes, like the DataRelayService and the MBUReceiver
communication.msp	MSP specific communication classes, an implementation of the communication framework for the HTTP interconnect protocol
core	MBU core components, like the DeviceManager and PluginManager
core.mbumanager	Package for the MBUManager (MBU SiteCore)
devicedrivers	Generic device driver classes, including the DeviceDriver interface
devicedrivers.dummy	Dummy device driver implementation classes
devicedrivers.mobi	TMSi Mobi device driver classes
gui	SWT implemented Graphical User Interface
plugin	Generic plug-in classes, like the plug-in interface and PluginLoader
plugin.analyser	An example plug-in class that analyzes the Mobi saw tooth (not demonstrated/explained in thesis)
sensormodel	Contains MBU specific BAN components (only necessary at MBU), like the DeviceSensorConfig (should be moved to shared)
utils	MBU specific utility classes

ehealth.surrogate.

communication	Surrogate specific communication classes (includes the HTTP interconnect specific protocols)
core	Surrogate core components (the Surrogate SiteCore)
jini	JINI related helper classes for registration and discovery
services	Generic services classes (place for the Exported Services Framework classes in future)
services.medical	Demo service for sensor data, chatting & file transmission

ehealth.backend.

client	Generic client-classes (connectivity and display)
client.monitor	Medical graphics implementation (like graphs and discrete vital signs display)

Appendix E Prototype library dependencies

This appendix gives an overview of the libraries that the prototype requires, distributed per site.

MBU

mbu_core.jar	MBU core components library
ehealth_shared.jar	Shared library
mbu_communication.jar	MBU communicator library
mbu_plugin.jar	MBU plug-ins library
swt.jar	SWT library for GUI
mspio.jar	Mobile Service Platform (MSP) communication library for mobile devices
messages.jar	MSP messages library
commons-httpclient.jar	HTTP communication library for java, required by MSP
commons-codec-1.3.jar	Required encoding/decoding for HTTP communication
commons-logging.jar	Generic logging-API, required by MSP
oncrpc.jar	Required for XDR encoding and MSP
kxml2.jar	XML parsing and writing library
log4j-1.2.12.jar	Extensive logging facilities (only partially used) for Java
javabeans.jar	Required for log4j when using IBM J9 JVM

SurrogateHost

MBUSurrogate.jar	Contains all eHealth related classes required at the surrogate, including the shared and exported services. Also contains downloadable classes for the client (medical-dl.jar)
HTTPinterconnect.jar	Contains the MSP classes for the surrogate host
messages.jar	MSP messages library
FixedExportServer.jar	A fix that allows the SurrogateHost to properly export jars for downloading
commons-logging.jar	Generic logging-API, required by MSP
oncrpc.jar	Required for XDR encoding and MSP
kxml2.jar	XML parsing and writing library
log4j-1.2.12.jar	Extensive logging facilities (only partially used) for Java
org.mortbay.jetty.jar	HTTP server for Java, required by MSP

Backend Client

PatientManager.jar	Core classes for the PatientManager, contain all backend-specific classes and also shared classes
medical-dl.jar	Exported Service specific library, contains service related classes that are required at the backend. Is downloaded from SurrogateHost
swt.jar	SWT library for GUI
jini-core.jar	JINI core classes
jini-ext.jar	JINI helper classes for easy discovery and downloading
sun-util.jar	SUN / JINI helper classes