ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Faculty of Engineering and Computer Science

Institute of Databases and Information Systems

Design and realization of a middleware for mobile task coordination

Master Thesis, Ulm University

Submitted by:
Georgy Karpenko
georgy.karpenko@uni-ulm.de

Reviewers:
Prof. Dr. Manfred Reichert
Dr. Ralph Bobrik

Advisor:
Dipl.-Inf. Julian Tiedeken

2012

October 29, 2012

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

The trend towards interconnection of applications has long been recognized as a key challenge for information systems design. Following this trend, organizations have developed and introduced many distributed systems with different functionalities. Furthermore, computing becomes today increasingly mobile; performances of mobile devices (i.e. PDAs and smartphones) as well as the expansion of high-speed mobile networks allows many tasks to be performed beyond stationary workspaces.

The dramatic growth of stand-alone and partly incompatible applications will negatively affect the integration, coordination and communication for entire solution. Contemporary solutions focus on stationary systems only; the usage of mobile devices is limited to simple scenarios (i.e. information access). In order to support the seamless integration of mobile devices, future distributed solutions should take services and service meta-information into account (e.g. variation of network bandwidth, battery power, availability, connectivity, reachability, sensors data and locations of services and service providers).

In this master thesis we want to analyze how a distributed environment with variety of separated (mobile) service providers - implemented with different technologies - can be integrated and coordinated. Finding compromises between performance, comfort and intelligent intercommunication is the main goal of this thesis. Therefore, it is concentrated on the conceptualization and design of a central middleware component that provide the coordination and communication functionalities for stationary and mobile entities. In order to prove some possible communication scenarios, the thesis provides a middleware-based scenario.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

BPI	Business Process Intelligence
BPMS	Business Process Management System
CDATA	Character Data
COM	Component Object Model
CORBA	Common Object Rejuest Broker Architecture
CR	Common Requirement
CRM	Customer Relationship Management
CRUD	Create, Read, Update, Delete
DCOM	Distributed COM
DOM	Database-oriented Middleware
ECG	Electrocardiography
EJB	Enterprise Java Beans
EOI	Event-Of-Interest
ERP	Enterprise Resource Planing
FIFO	First In - First Out
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol

IIOP ..... Internet Inter-ORB Protocol

J2EE ..... Java 2 Platform, Enterprise Edition

LBS ..... Location Based Services

LTE ..... Long-Term-Evolution

MIME ..... Multipurpose Internet Mail Extensions

MOM ..... Message-oriented Middleware

NFC ..... Near Field Communication

OOM ..... Object-oriented

ORB ..... Object Request Broker

PAS ..... Patient Administration System

PDA ..... Personal Digital Assistant

PPM ..... Process Performance Management

QR ..... Quick Response

RDBMS ..... Relational Database Management System

RDF ..... Resource Description Framework

REST ..... Representational State Transfer

RMI ..... Remote Method Invocation

ROA ..... Resource-oriented Architecture

RPC ..... Remote Procedure Call

SCM ..... Supply-Chain Management

SDK ..... Software Development Kit

SOA ..... Service-oriented Architecture

SOAP ..... Simple Object Access Protocol

SPARQL ..... SPARQL Protocol and RDF Query Language

SQL ..... Structured Query Language

TCP ..... Transmission Control Protocol

TR ..... Technical Requirement

UCR ..... Use Case Requirement

UML ..... Unified Modeling Language

UML ..... Unified Modeling Language

UMTS ..... Universal Mobile Telecommunications System

URI ..... Unique Resource Locator

USB ..... Universal Serial Bus

W3C ..... World Wide Web Consortium

WCF ..... Windows Communication Foundation

WfMS ..... Workflow Management System

WLAN ..... Wireless LAN

WSDL ..... Web Service Description Language

WWW ..... World Wide Web

XML ..... Extensible Markup Language

1 Introduction

Mobile devices, such as mobile phones, personal digital assistants (PDAs), and tablets become more and more popular. These devices will be continuously networked and software development kits (SDKs) are available that can be used to develop third party applications. Mobile computing is rapidly moving from the realm of the research to the main stream. Until now, most applications are "content consuming" (e.g. news or emails readers, games). Nowadays, mobile devices allow building of "intelligent" applications, which imply the integration of mobile devices into business processes and enterprise information systems. Introducing this kind of devices and enabling new classes of applications presents a challenging task to application developers and system architects.

Modern mobile devices have sufficient processing power and storage for complex computation and processing, and are equipped with multiple network interfaces, including WLAN, UMTS (Universal Mobile Telecommunications System), and LTE (Long-Term-Evolution). It is now feasible to host services on mobile devices and participate in a service-oriented environment (SOA) [12]. Applications running on mobile devices can provide context information, such as their position or current communication environment. It is also possible to connect additional sensors and equipment to mobile devices via Bluetooth, NFC (Near Field Communication) or USB cable. All these allows the building of context-aware mobile services [11] and the support of flexible (context-aware) process distribution [20, 25]. By context it refers to any information that can be used to characterize the situation of an object, where an object can be a mobile device or mobile service provider. Context can be the device's current location, a user profile, the display size, sensor data, or present connection bandwidth. Such context-aware services can be used in many areas, for example:

- Presenting information to mobile users depending on mobile device characteristics - e.g. picture pre-processing dependent on receiver's screen size.
- Triggering actions on the occurrence of a set of properties - e.g. advanced filtering possibilities and fine flow control.
- Execution of process activities on appropriate selected providers (by context) - e.g. tasks that require experienced executor.
- Location-based services (LBS) - selection of providers based on their spatial location.

1.1 Running example

The use case of this thesis depicts the communication process for administration, treatment and incident management in a clinical context. In the clinical domain many law regulations have to be considered: documentation and traceability of medications, decisions, treatments, and patient agreements. Therefore, a doctor must protocol all steps and tasks during examination and treatment processes. Nowadays, many implementations of patient administration systems exists [16]. In many cases, the documentation is still made manually with papers. The typical communication is face to face - doctors give instructions to nurses. Figure 1.1 illustrates a typical clinic treatment process: if the patient's blood needs to be analyzed in a special laboratory, the doctor delegates this task (①) and the nurse ensures that the blood test will be gathered from the patient (②) and examined in a laboratory (③). After the examination results are evaluated, the nurse picks them from laboratory (④), and protocols them(⑤). The doctor regularly controls the patient's current health state (⑥) and discusses with the patient the next steps of the treatment (⑦). All these operations are time consuming (especially human communication). In many cases, this time loss is very critical for the patient's treatment success (i.e. in urgent incidents where decisions have to be made quickly).
Digitalization of paper documents can be made with concepts like MEDo [16], which provides a replacement for paper case histories with a tablet computer

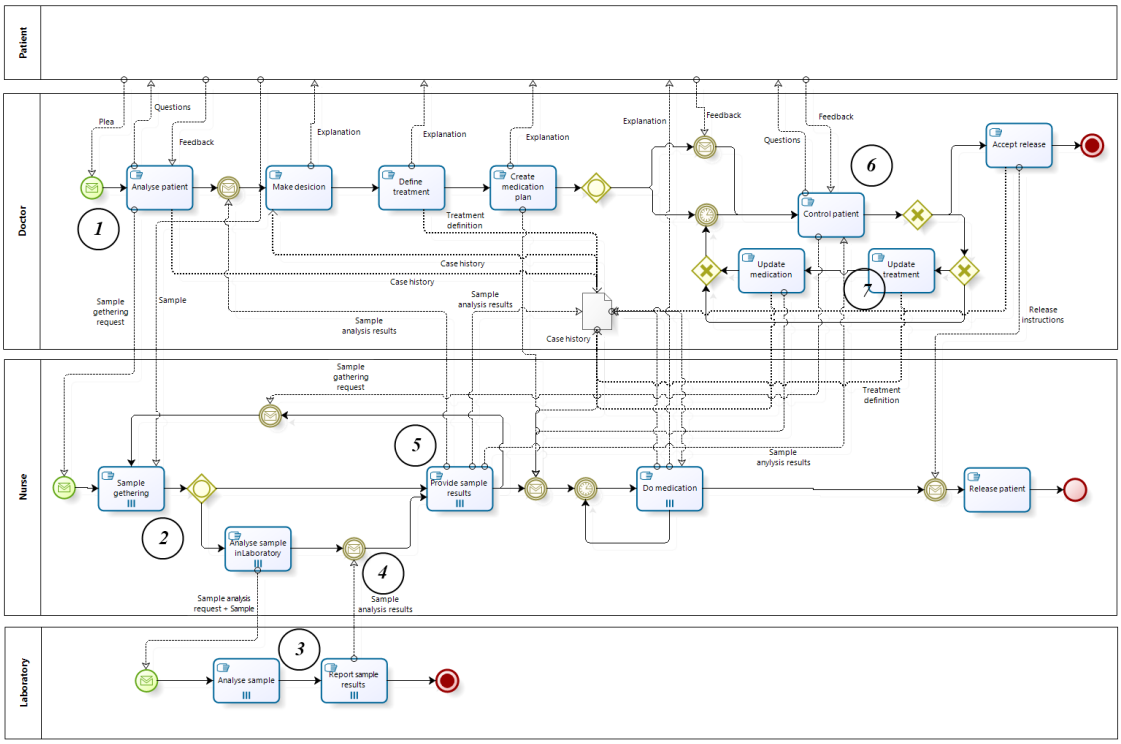


Figure 1.1: Stationary treatment process (in BPMN 2.0) (see B.1 for larger view)

application. With digitalized case data handling the described treatment process could be accelerated. Human-based transfer of the case histories or manual updating are time-consuming. This is comfortable and more responsible in order to improve the quality of patient data collection processes. Concepts like MEDo can accelerate the capture of information, but process execution and the automation of communication remain a challenging task. Ideally, all communication operations and data collection tasks are integrated and coordinated with the help of mobile devices. The following list presents a possible scenario, where the process consists only of required steps, and communications are automated:

1. The doctor analyses the patient data and defines next steps of treatment: analysis of patient's blood by a laboratory.

2. Laboratory defines the blood collection appointment depending on the laboratory schedule, priority and severity, and informs the patient about it.

3. At the defined time point the patient comes to the laboratory for a blood collection.

4. If the patient cannot come to the laboratory, a free and closest nurse comes to the patient for the blood collection and brings the sample to the laboratory.

5. The doctor will be notified about the task execution and about changes in patient data (e.g. blood analysis results were added) and can continue the treatment.

First, the only required manual tasks in this case are: decision by a doctor, blood collection, and test analysis. All other tasks and operations are made automatically (communication, notification and reporting).
Second, manual scheduling requires a high degree of coordination and can be error prone. Executing it in an automatic manner (e.g. as shown above) will accelerate the communication and the treatment process.
Third, automation of notification and event-processing can also drastically increase the reaction time. This is very important, especially for emergencies.
Nowadays, companies (including hospitals and clinics) integrate business process management systems (BPMS) in order to define the process flow, to execute and control the execution of processes and to allow the analysis of process logs (e.g. with BPI (Business Process Intelligence) and PPM (Process Performance Management) concepts). BPMS can be realized with the SOA paradigm, where services can realize tasks and the BPMS calls the related service when the task should be executed. BPMS can also handle the data flow by providing the service required data and saving the service execution result.
From this point of view, the provided stationary process can be mediated by a process management system by defining appropriate services. By analyzing the treatment process, it is noticeable that many tasks are constantly repeated. These tasks could be grouped and served as a service candidate. Figure 1.2 shows the appropriate grouping for possible service candidates:

1. Reception service: A doctor meets a patient and makes an anamnesis, decides if a treatment is needed, and creates a case history based on symptoms and then defines a treatment with necessary medication. Additional tests may be required for a correct treatment decision. In that case, the doctor can request the "make tests" service.

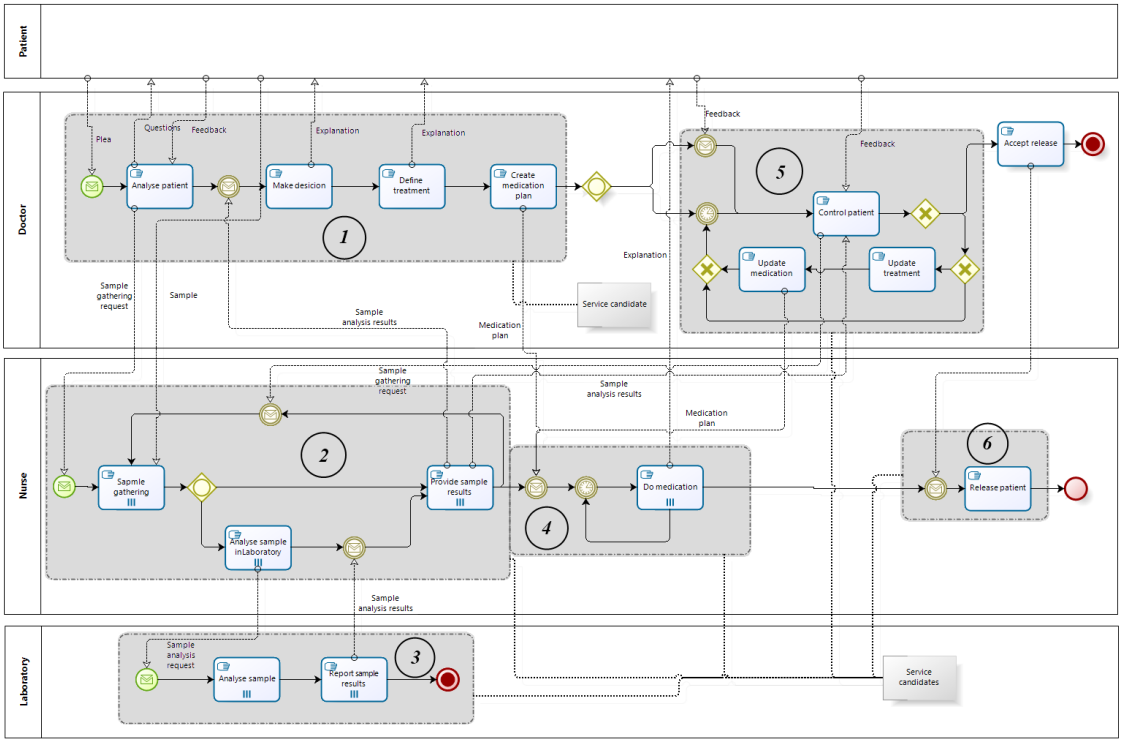


Figure 1.2: Stationary treatment process with appropriate service candidates (without data objects)(in BPMN 2.0) (see B.2 for larger view)

2. Make tests: A nurse makes defined tests with the patient or gathers required samples from patient and provides the results to the doctor. Some samples (e.g. blood, urine) must be analyzed in a laboratory - "sample analysis" service by a laboratory.

3. Sample analysis: A laboratory receives a sample, makes necessary tests and reports the results.

4. Medication: A nurse regularly executes scheduled activities for medication: provides tablets or makes injections.

5. Patient control: A doctor controls the actual state of a patient and makes updates and changes in the treatment plan. The doctor may also request new tests.

6. Patient release: After the treatment was successful and accepted by a doctor, a nurse has to prepare the patient for the release: make final documentation and close the treatment process by finalizing required organizational and bureaucratic tasks.

The usage of mobile devices (i.e. smartphones) allows doctors and nurses, who are constantly moving in the clinic area, to be integrated into an automated treatment process. Providing services on handheld devices, so called "mobile service providers", will ensure that doctors and nurses will be updated with necessary patient data and tasks without delay. Execution of any service in the provided use case on mobile device usually requires user interaction (i.e. doctor meeting with patient). Therefore, mobile services become stateful. The core problem of stateful services is that they stay blocked until completion (parallel execution of the same service instance is not possible).
Another problem of mobile services are possible connection losses; roaming from one wireless network to another, mobile devices constantly change their physical service addresses.
These common problems cannot be resolved locally by extending systems and mobile applications with a service wrapper. To support heterogeneous systems and provide a single system, distributed systems can be extended with a special layer [22]. The name of this layer is called "middleware". The main focus lies on communication between application components [13] and will be discussed in the following thesis.

1.2 Thesis structure

Developing of systems, which are based on context-aware mobile services, is a complex and time-consuming task. So, the main goal of this thesis is to find how a distributed environment with variety of separated (mobile) service providers, implemented with different technologies, can be integrated and coordinated. Therefore, the conceptualization and design of a central middleware component, that provides the coordination and communication functionalities for stationary and mobile entities, will be described.
The main part of this master thesis will be the analysis for appropriate technologies, concepts, design and architecture decisions of the middleware for mobile services. Chapter 2 examines different middleware types and also covers available technologies that can be used for interoperable communication and trans-

mission of data. Usage scenarios, requirements and the derivate middleware architecture are illustrated in Chapter 3. Here, the conceptual design of the middleware based on requirements and defined required technologies will be shown. Chapter 4 provides the detailed definition of the message template, processing logic, interface specifications, and communication processes. To sum up the work, Chapter 5 provides a middleware usage demonstration. Achieved goals and an outlook of possible further developments is discussed in Chapter 6.

2 Basics

In this chapter, the term middleware will be defined. Additionally, this chapter discusses concepts, architectures, and standards for distribution of systems.

2.1 Synchronous vs. Asynchronous interactions

The main characteristic of a software communication is either synchronous (blocking calls [1]) or asynchronous (non-blocking [1]). So, if the code on the client side blocks until a result arrives, it is a synchronous interaction. If, instead of blocking, a client performs other tasks after sending a request, it is an asynchronous interaction. In the following, details about both scenarios will be explained in detail.

2.1.1 Synchronous interactions

In synchronous interaction, a thread of execution calling another thread must wait until the response comes back before it can proceed (cf. Figure 2.1). It is easier an application programmer to understand this interaction scenario as it follows naturally from the organization of procedure or method calls in any program. As a result, synchronous calls are used in many types of middleware.
The fact that the calling thread must wait can be seen as a disadvantage, especially, if the call takes a long time to complete. Waiting is a particular source of concern from the performance point of view [1]. Similarly, since every call opens a new connection, there is a danger of running out of connections, if there are too many waiting calls. Finally, a tight integration between distributed components is needed: a system should “understand” the information that is coming

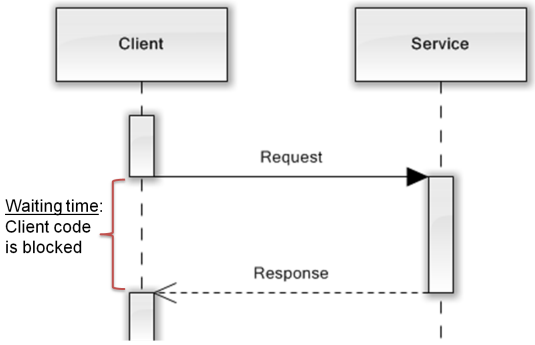


Figure 2.1: Synchronous interaction (UML Sequence diagram) [1]

from the communication partner, and both systems should support the same communication protocol. This can be impossible in highly distributed, heterogeneous environments. The best use cases for synchronous interactions are closely cooperating systems with short procedures calls [13].

2.1.2 Asynchronous interactions

The alternative to synchronous interaction is asynchronous communication. In a wide range of application scenarios it is not at all necessary to work synchronously. Distributed systems can be built using a similar approach: instead of making a call and waiting for the response to arrive, a message is sent and, some time later, the program checks whether an answer has arrived. This allows the program to perform other tasks and eliminates the need for any coordination between both ends of the interaction (cf. Figure 2.2). Synchronization and asyn-

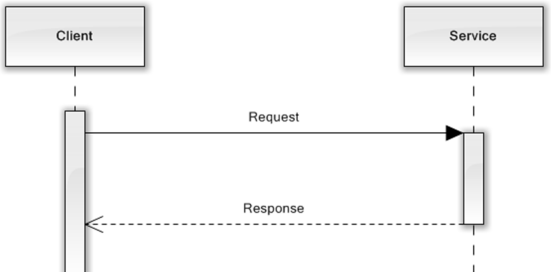


Figure 2.2: Asynchronous interaction (UML Sequence diagram) [1]

chronization describe the process (thread) point of view to a remote procedure call. Technically, the necessary interaction can be achieved by extending the

code with programming language techniques: making a synchronous call in a separated thread (process) for asynchronization or blocking the process and wait until receiving a response for synchronization.

2.2 Middleware

The term "middleware" is, like many others in scientific discussions, not uniformly defined. A broad definition can be summarized as follows: The middleware is basically any type of software that facilitates communication between two or more software systems [22, 13]. This definition can identify the main features of a middleware: First, middleware is a software-based approach. Another aspect of the definition is that middleware is used to enable communication. In terms of software systems, communication means the exchange of data and information. This can be either between two, or more software systems. One problem for integration of multiple software systems is heterogeneity. The heterogeneity of software systems can relate to many areas of software systems, such as the programming language, platform and content. Hiding the heterogeneity of underlying networks, hardware, operating systems, communication patterns, and programming languages is the central goal of the middleware [13]. The name "middleware" is given due to the fact that the offered services are performed between the operating system layer and application layer.
Middleware can be classified differently depending on the purpose. A common classification divides middleware by examining their software systems, and the information exchanged into Remote Procedure Calls (RPC), Object-oriented Middleware (OOM), Database-oriented Middleware (DOM), and Message-oriented Middleware (MOM). RPC and OOM work with synchronous communication mechanisms, whereas MOM uses an asynchronous mechanism. These three types of middleware will be described in more detail below.

2.2.1 RPC

The Remote Procedure Call (RPC) enables communication between applications running on heterogeneous platforms. RPC is based on procedural concepts and supports remote procedure calls. RPC hides the details of communication and low-level network communication to application programmers. The central concern of RPC is to make a local program function remotely. For the local program a remote function call behaves like a local call. An RPC is transparent for the local program. This type of transparency requires that the calling application blocks and waits for the response.

2.2.2 Object-oriented middleware

Object-oriented middleware (OOM) supports communication between distributed objects and components. OOM is usually implemented using an Object Request Broker (ORB). It allows local objects or components, methods of remote objects or components to be used by interfaces. In principle, OOM is an additional layer on top of RPC. OOM is a middleware that uses synchronous communication mechanism and hides details of the communication. The manufacturer of ORB decides about the design of interfaces to support interchange between heterogeneous and distributed system component implementations in terms of programming languages and programming platforms. The three most important standards for OOM are CORBA (Common Object Request Broker Architecture) from OMG[1], JAVA RMI[2], as well as COM / DCOM / COM+ [3]. Nevertheless, a connection between these technologies is possible. Many products are available with the ORB specifications, and only several implementations are compatible with each other: RMI-IIOP-based implementations. In particular, the support of RMI-IIOP is important, because it uses the same communication protocol such as CORBA and IIOP (Internet Inter-ORB Protocol). Distributed objects are the foundation of Enterprise Java Beans (EJB), and accordingly, the central means of RMI or RMI-IIOP communication carried out part of the overall J2EE platform[4].

[1] http://www.corba.org/, viewed 14.10.2012
[2] RMI - Remote Method Invocation
[3] http://technet.microsoft.com/en-us/library/cc722925.aspx, viewed 14.10.2012
[4] http://docs.oracle.com/javase/1.4.2/docs/guide/rmi-iiop/tutorial.html, viewed 14.10.2012

2.2.3 Message-oriented middleware

RPC and OOM are examples of synchronous communication based middleware. Message-oriented Middleware (MOM) allows asynchronous communication via messages. They are not sent directly from a transmitter-receiver application to an application, but go through an intermediary to their destination. This seals the MOM and the technical realization, which is called the *Message Server* or *Message Broker*. Through the usage of messages and an intermediary with a high level of interoperability, the communication between heterogeneous software systems is possible. Though, the communicating applications get decoupled. The sender application sends the message to the mediator and continues to work immediately. The message server is responsible for forwarding the message to the recipient application. If it is temporarily unavailable, the message server keeps the message until the application is listening. Since the communication takes place via messages, the application does not need to know any details about processing.
The usage of a message server and the associated central architecture allows easy implementation of not only point-to-point, but also many-to-many communication scenario. The applications communicate directly with the message server. Thus, the entire architecture is flexible, because changes are easily to perform. A message server is responsible for the transmission of messages. Often, it provides additional services, such as load balancing and transactions support. A central server, however, can have a big disadvantage, because the whole architecture is server-dependent. If the server goes offline (e.g. due to technical problems), the whole underlined system will be unavailable.

2.3 RPC vs. Messaging

More important is the fact how and which data will be passed to a remote procedure during the call. At this point, it is necessary to distinguish between RPC/RMI and the Messaging model.

2.3.1 Messaging model

The well-known example for messaging (message-passing, message queuing) is the email: a sender writes a message and defines a receiver (receivers) by setting a receiver email address(-es); the receiver becomes the message and reads it [1]; the receiver can answer the message with a new message or forward it to a new receiver for additional processing. There is not necessarily a one-to-one correspondence between messages sent and received (request-reply/two-way messages), and indeed, a response way may not even be required (one-way messages). Data and calls are packaged in the form of messages and transmitted. The message format is specified by the respective communication parties [13].
The core part of messaging is *queuing*. The queue is responsible for storing the messages (until a receiver is online and can get associated messages) and for distribution of messages between multiple receivers (cf. Figure 2.3). Queues for sending and receiving of messages will be commonly separated from each other.

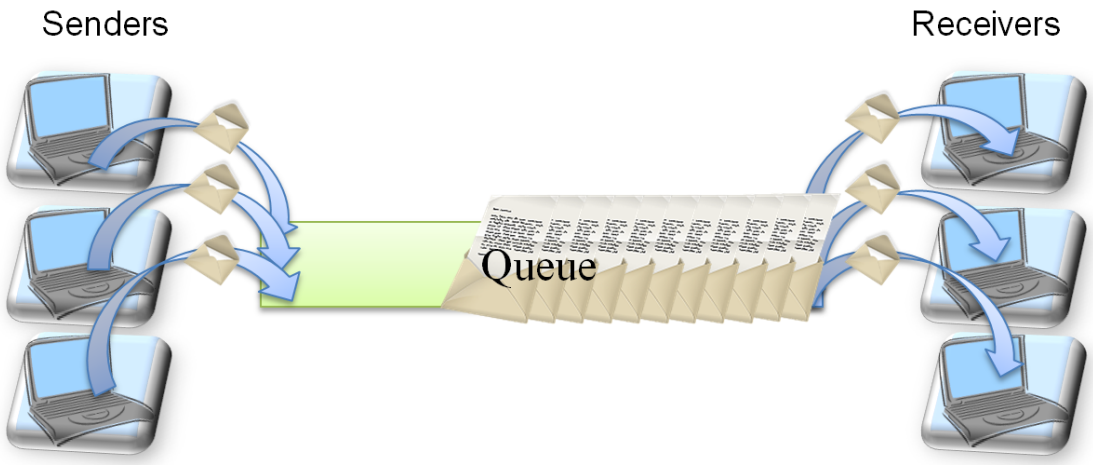


Figure 2.3: Message queue

Using the central queue component allows many communication scenarios:

- **Ono-To-One**. In the one-to-one communication scenario a sender defines exactly one receiver. As in the email example above, the one-to-one sce-

nario can be represented as “private” message for a concrete (and only one) receiver.

- **One-To-Many**. In a one-to-many scenario a message will be sent to a group of receivers.

Messaging mechanism is used mainly in scenarios where information dissemination is required but the actual usage of information (reaction) is not known: like publish-subscribe or event distribution. These are systems in which interaction between components does not occur through explicit calls or explicit exchange of messages, but through the publication of event or signals (tasks) that inform those interested that a particular system state has been reached. This will be ensured through publish-subscribe patterns, where components continuously make information available by publishing it to the system, while other components indicate their interest on parts of the published information by subscription to it. The middleware is then responsible for matching published information to subscriptions and delivering the information to the subscribers [1]. This activity requires an intermediate component where the messages (publications) are stored until they are received by a receiver. Following this idea, many queuing systems that were used in the past simply to forward messages between components are now being used as message brokers. These brokers can filter and control the message flow, implement complex distribution strategies, or manipulate the format or even content of messages as they transmit through the brokers.

2.3.2 RPC/RMI model

The messaging model requires that both sides are responsible for the creation of messages in a format, which is understood by both sides. However, most standalone applications do not make use of message-passing techniques due to additional efforts for the definition of a messaging format and implementations of communication standards. Generally, the preferred mechanism is the remote function (or method or procedure) call, which is commonly supported by many programming platforms (e.g. RMI in Java, WCF[5] in .Net). In this style, a program

[5]WCF - Windows Communication Foundation

will call a function with a list of parameters, and after completion receives a set of return values. These values may be the function value, or addresses. RPC combines synchronous communication with the procedural programming paradigm. And RMI combines synchronous communication with the object-oriented programming paradigm.

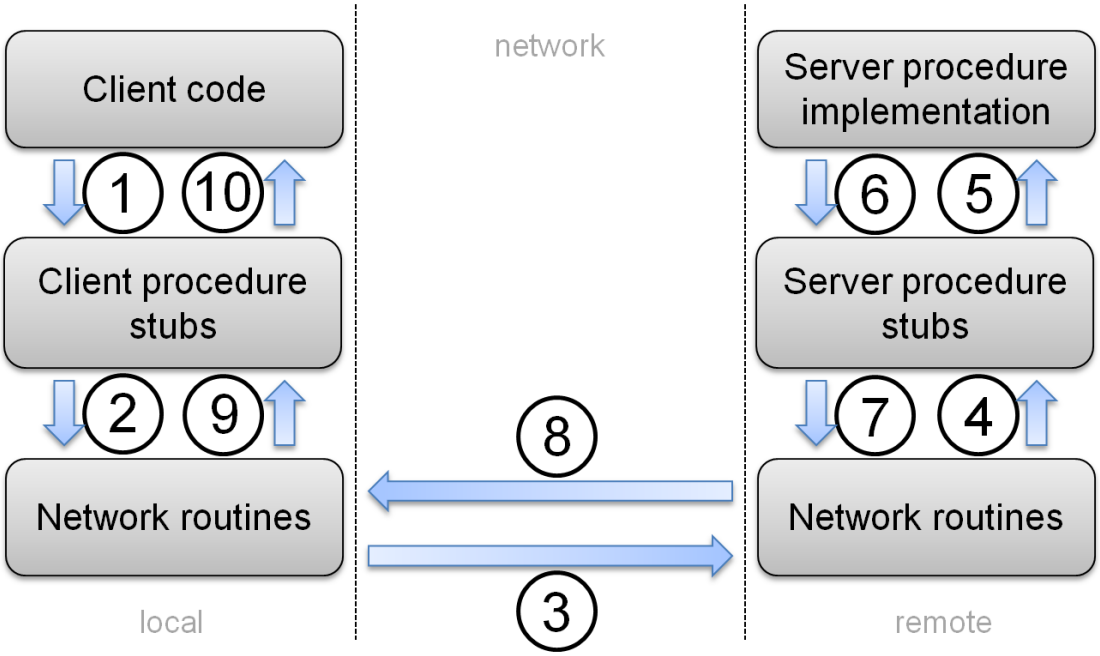


Figure 2.4: RPC communication pattern

The remote procedure call (cf. Figure 2.4) is an attempt to bring this style of programming into the network world. The client calls a procedure that seems to be local (①). The client-side will package parameters into a network message (②) and transfers it to the receiver (③). The receiver will unpack this (④) and turn it back into a procedure call on the receiver side (⑤). The results of this call will be packaged up for return to the sender (⑥-⑩). There is a strong correlation between the code that makes the call and the code that deals with the response. Logically, it is easier to understand what happens in a RPC-styled communication since the different components are strongly tied to each other in each interaction, which greatly simplifies debugging and performance analysis. As a result, RPC has dominated almost all forms of middleware [1]. For systems where the presentation layer was moved to the client, this was generally done through RPC. Similarly, when the application logic and the resource management layer

were separated, most systems used RPC for communication between two layers. The strong binding between components results in close coupling and in an impossibility to call a function by many providers in parallel: only one-to-one communications are possible. That means that the requestor-component and the response-component communicate with each other with permanent connection and should both be online during communication and remain operational for the entire duration of the call. The tied integration between the components imposed by RPC may be impossible to maintain in highly distributed, heterogeneous environments, and, also, very complicated when there are many tiers involved. This has obvious implications because of the reduced fault tolerance and the more complex maintenance procedures. These problems become more important when the number of components increases.

2.4 SOA and ROA

Systems that are based on technologies discussed above are commonly having been rather successful in focusing on special problems by distribution of systems. In special cases, companies have developed own formats to exchange data and own standards for communication. Mostly, these formats are proprietary and the system is designed for (company) internal use only. In cases where several internal systems should be integrated in cross-company environment scenarios (i.e. procurement, SCM (Supply-Chain Management), or CRM (Customer Relationship Management), proprietary data formats, proprietary communication protocols, and different programming platforms are main stumbling blocks. Again, the automation of business processes across autonomous and heterogeneous systems is a big challenge.

The usage of standard technologies, by contrast, reduces heterogeneity of internal systems. This standardization can be achieved through the use of Web services. Web services are the way to expose the functionality of an information system and make it available through standard Web technologies [1]: “Web services are self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces. Web services communicate directly with other Web services via standards-based technologies. These standards-based

communications allow Web services to be accessed by customers, suppliers, and partners independently of hardware, operating system, or even programming environment" [6].
Web services are an evolution of approaches for building distributed systems. The term "Service-oriented architecture (SOA)" was first described by Gartner in 1996 [21, 19]. Since then, there were numerous different definitions for SOA. In general, SOA represents an approach for separating operations. It allows to solve, construct, carry out and manage logic of a large problem through dividing the problem into many smaller, related pieces. Each of these smaller pieces, within SOA is known as a service, should be concentrated on a specific part of a problem and can be distributed over the network. The HTTP (Hypertext Transfer Protocol) transfer protocol and XML (Extensible Markup Language)-based communication improve interoperability, platform and vendor-independence. It should be noted that SOA in not a standard to build distributed system, but rather an architectural paradigm which describes the possible roles and principles how SOA can be designed.

2.4.1 SOA

Within SOA there are three types of roles (cf. Figure 2.5): a service provider (service), a service consumer (client) and a service registry (also known as a service repository or service broker). However, in some models the service repository is not provided.
The **service consumer** is an application, a software module or another service that requires a service. It initiates enquiry of the service in a registry, binds to the service over a transport protocol and executes the service functions [7].
The **service provider** is a network-addressable unit that accepts and executes requests from consumers. It publishes its' services and interface contracts to the service registry so, that the service consumer can discover and access the service [7].
A **service registry** is an enabler for service discovery. It contains a list of available services and allows the lookup of service interfaces to interested service

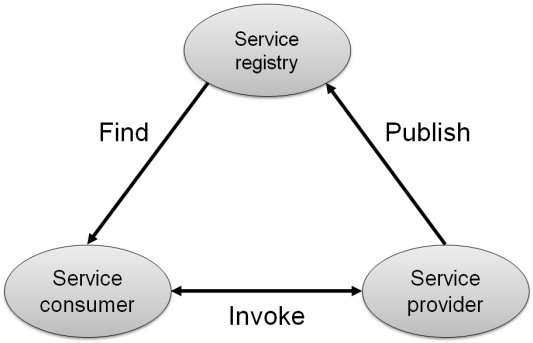


Figure 2.5: Roles in a SOA

consumers [7]. A SOA model without the service repository is called *well-defined* [8]. Based on this approach, service providers must be unique, static, and defined with constant addresses, so there is no need for a service repository.
To ensure the relationship between all units inside a SOA, each item (service, client, or repository) must implement corresponding role-based operations defined by the basic SOA model [7, 2]:

- **Publish**: To be accessible, a service description must be published by the service registry so that it can be discovered and requested by a service consumer.
- **Find**: A service consumer localizes a service by querying the service registry for a service that meets the requestors' criteria.
- **Invoke**: After retrieving the service description and the service address, the service consumer requests the service accordingly to the information in the service description.

The SOA approach is not a specification to build applications, which share own functionalities over the network. SOA is a paradigm how to design applications, which provide and consume the distributed operations (services) of each other. It is commonly accepted [8, 2] that several service-orientation principles have their roots in the object-oriented design paradigm. Hereafter is a list of specific service-orientation principles common to all primary SOA platforms [8]:

- Services are *reusable*: Services needs to be designed to support potential reuse.

- Services share a *formal contract*: For services to interact, they need to share a formal contract that describes each service and defines the terms of information exchange.
- Services are *loosely coupled*: Services must be designed to interact without the need for cross-service dependencies.
- Service *abstracts underlying logic*: The only part of a service that is visible to the outside world is what is defined in the service contract. Implementation details are invisible to service requestor (consumer).
- Services are *composable*: Services may request other services.
- Services are *autonomous*: The logic governed by a service resides within an explicit boundary.
- Services are *stateless*: Services are not required to manage state information.
- Services are *discoverable*: Services should allow their descriptions to be discovered and understood by human and service requestors.

Thus, the principles defined by different authors are almost the same. In general, a service should share a well-defined contract (description) to be accessed, only input and output types are visible for a consumer, and a service should enclose unique (reusable) program logic, which could be requested at any time by a consumer or other services.

2.4.2 ROA

Whereas the main part of SOA is concentrated on services as a unit of logic, ROA (Resource-oriented Architecture) is another approach that, in contrast to SOA, is focused on resources and resource representation. The term ROA was born by combining of SOA and utilization of REST (Representational State Transfer) services. REST principles were first introduced by Roy Thomas Fielding in his dissertation in 2000 [10]. The central point of a ROA is a resource that can be accessed from any client. The client is responsible for the interpretation

of the resource and internal processing. Therefore, a REST service provides a representation, or current state, of a resource (object).
The REST paradigm is based on principles of the World Wide Web (WWW): each resource on the network has a locator – an URI[6]; by knowing of this locator, a client can access the located resource on respective Web server by using the HTTP protocol. The REST does not require additional protocols for data transfer as well as additional standards for service description. There are no protocol conventions need for the communication between clients and servers. A central role play HTTP methods GET, PUT, POST, and DELETE. Each REST resource has a generic interface in form of these HTTP methods. Almost all applications can be covered with these four methods: it can be compared with an application that uses SQL, and SQL generic commands like SELECT, INSERT, UPDATE, and DELETE provides necessary handling of objects in a database. These operations are typically subsumed by the acronym CRUD: Create (INSERT), Read or Retrieve (SELECT), Update (UPDATE) and Delete (DELETE). The following list describes the meaning of CRUD used by REST:

- **GET**: GET retrieves the representation of a resource - Read.
- **POST**: with POST, a client can add something to the resource: for example, a product can be added to a shopping cart - Update.
- **PUT**: New resources can be created with PUT. Also, the content of existing resources can be replaced with PUT - Create.
- **DELETE**: resources can be deleted with DELETE - Delete.

To sum up, ROA is an architectural model, which describes how the Web should work. The model serves as a guide and as a reference for future enhancements. ROA is not a product or standard; ROA describes how web standards can be used in a Web-friendly manner.

[6] URI - Unique Resource Identifier

3 Requirements and Design

The research goal of this thesis is the design and implementation of a middleware component for mobile task coordination. In Chapter 1, a use case was introduced. In brief, a middleware should enable tasks distribution for on mobile devices (handhelds, PDAs, smartphones, tablets) and the whole process should be controlled by a BPMS. A task is a sequence of assembled activities to be executed. Also in Chapter 1, it was defined that all tasks are distributed through all involved parties of the presented scenario. These distributed tasks represent services (not to be confused with Web services) of entire process participants.
In this chapter, the design of the middleware will be described. First, based on the use case, possible application scenarios of the middleware usage will be defined. Second, requirements for the middleware to be designed are observed from technical and from qualitatively points of view. Finally, the middleware architecture will be derived from the found requirements and described in detail.

3.1 Common requirements

In this section, scenarios for the use of a middleware in a clinic environment, where services (tasks) are distributed through mobile devices, will be researched. Figure 3.1 illustrates the placement of the middleware in a possible clinic environment. All possible service executors, such as legacy systems, Workflow Management Systems (WfMS), patient administration systems (PAS), and service clients are represented on the top, above the middleware. All available services providers are on the bottom: this includes mobile services and Web services as well. The middleware abstracts the service access by providing a consistent view of underlined services. The requestor does not differ between a mobile service call and a Web service call.

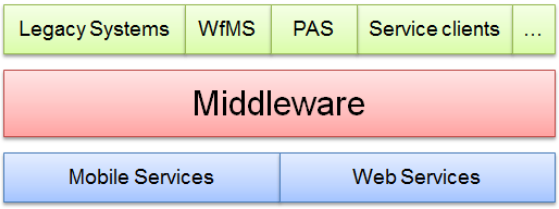


Figure 3.1: Middleware placement

By analyzing the middleware placement, the following common requirements (CR) for the middleware usage can be proposed:

3.1.1 CR1: Communication

Communication is the core scenario for any middleware. The middleware should enable transparent communication between two or more separated and distributed software components. So, a service requestor does not need any special connectors for communicating with different kinds of services (mobile or non-mobile services). All connectors are replaced by the middleware, and the middleware provides everything needed for successful communication. By implementing the connector to the middleware, a service requestor will be able to communicate with all supported services.

3.1.2 CR2: Cross-parties communication

In addition to CR1, the middleware should allow not only requestor-provider communication, but rather all possible communication scenarios. In some cases, even a service acts as a client (services are composable and reusable (see Section 2.4.1)). Therefore, the middleware should allow the usage of communication possibilities from stationary as well as from mobile or Web platforms, because any component can act as a service requestor (client).

3.1.3 CR3: Central bus

The number of connections needed to ensure fully meshed point-to-point communication (like within RPC) between *n* endpoints can be calculated with the

following simple formula: $\frac{n(n-1)}{2}$. Thus, for 10 points to be fully integrated 45 connections are needed. By integrating a central bus component, any point will communicate only with the central bus (cf. Figure 3.2). The total amount of connections required in this case is equal *n* (or 10 for the example above). Another problem of point-to-point communication is the need of the knowledge

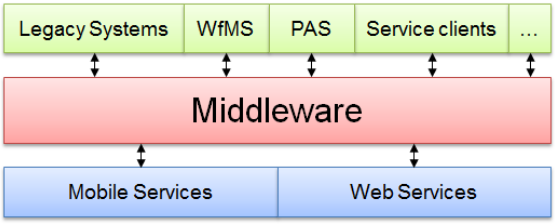


Figure 3.2: Middleware as central bus

of used communication protocols: without exact information about required protocols it is impossible to communicate. From CR1 and CR3 follows that the implementation of a middleware as a central communication bus can improve the reuse and the substitutability of services. For example, any Web Service can be re-implemented as a mobile service, and vice versa. As defined in CR1, this substitute will be completely transparent for requestors, since a standardized connection to the middleware exists, which is the only possible communication way for services execution.

3.1.4 CR4: Routing

Since all services are accessible at one place (CR3), a client (or sender of a message) does not need to know the receiver's (service provider) address. By

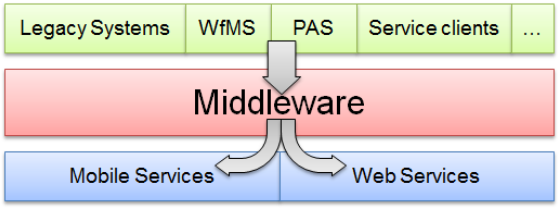


Figure 3.3: Middleware as router

sending a request, the client specifies properties (location, owner role, display size) of a service provider and sends this information to the middleware. Based

on these specified properties, the middleware routs the request to a concrete service (cf. Figure 3.3). The routing is done by the middleware and does not require to be considered by the service requestor.

3.1.5 CR5: Service transparency

As described above, the middleware hides all details of a service implementation and provides a consistent view for all underlined services. Thus, a service requestor will not recognize whether the called service is running on mobile device or not. In addition to CR2 and CR3, the middleware requires a common standard for service properties description (CR4) by the requestor. Only common descriptions - without the definitions of use technologies for service implementation (mobile or web services) - ensures the transparency of service implementation.

3.1.6 CR6: Repository

Services, being published in a repository of the middleware, should be discoverable by a requestor. Since a client does not communicate directly with a service (CR3), and therefore, does not require the concrete details about service implementations (e.g. WSDL (Web Service Description Language) [5]), listings of available services can be used only for administrative purposes and also during the development. As a result, the repository participates in the middleware internal request processing, and can be used from outside for administrative purposes solely.

3.2 Use case requirements

At the next stage, the clinic process will be analyzed. Clinic process is highly important for scenario and requirements analysis for the researched middleware, because it builds the usage context, and as a result this usage context predefines concrete use case requirements (UCR) for the middleware. Figure 3.4 shows the treatment process from Chapter 1.

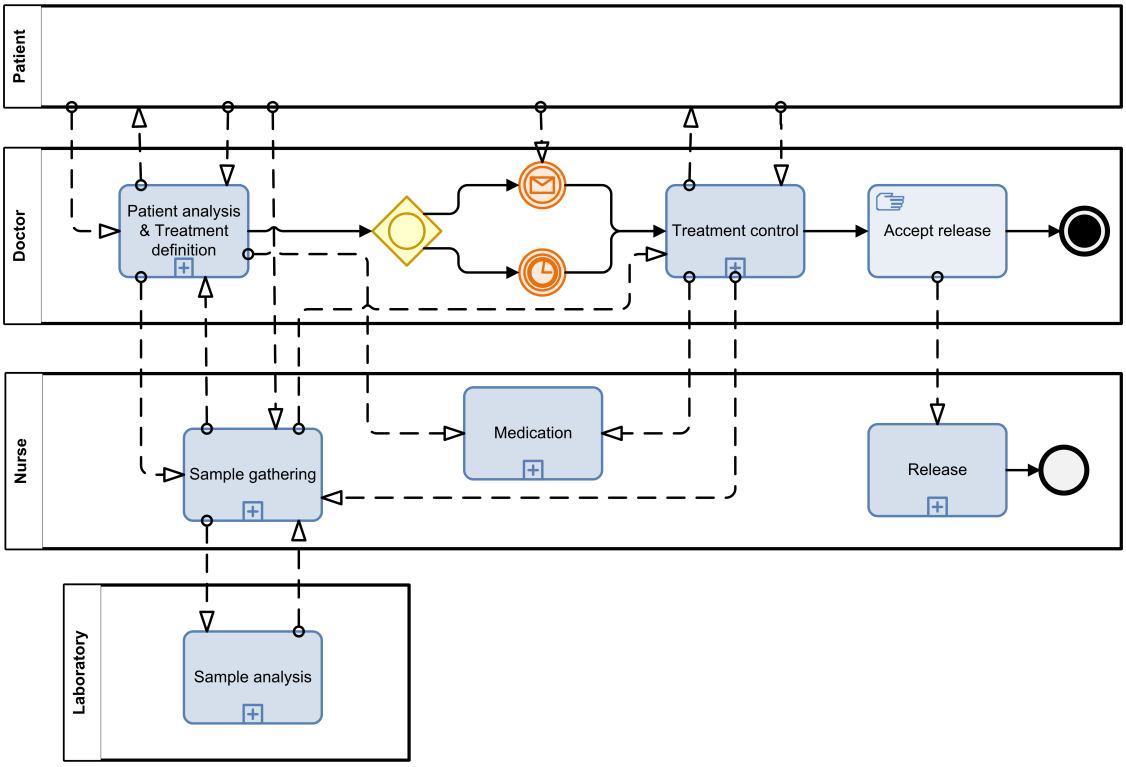


Figure 3.4: Treatment process (collapsed) (in BPMN 2.0) (see B.3 for larger view)

The communication within this process has some critical paths:

First, during patient analysis, a doctor may request a sample gathering which is executed by a nurse. Finding a free nurse can be a time-consuming activity, and furthermore all nurses can be busy.

Second, during treatment definition, a doctor defines tasks, which should be done in the future. The treatment will be stored in the patient treatment history; a nurse, who will execute treatment activities, needs to regularly look inside the treatment history of every patient and, therefore, make her daily work plan. In cases of many patients, this planning activity will be almost impossible and requires big administrative effort.

Third, a similar problem occurs, when a doctor must wait for patient sample results. Some decisions should be made very quickly (e.g. in case of an emergency) and the doctor does not have much time, routine checkups must be postponed. Continuously checking of treatment history updates will distract the doctor from his main tasks.

The researched middleware should solve these use case problems and automate routine activities. Thus, required use case scenarios are listed below:

3.2.1 UCR1: Queuing

The discussed (mobile) services are stateful (see Chapter 1). Consequently, it can happen that at the point of task processing no services are available. In such cases, the middleware should save the task and try to process it later, when an appropriate service is online. Therefore, queuing of services/tasks (e.g. medication or sample gathering) is needed.

3.2.2 UCR2: Scheduler

It should be possible to schedule the processing of the message (receiving of the message by a receiver) by setting the time interval of validity and the priority of the message. So, a doctor, by defining a treatment plan, should have the possibility to define the exact time point of medication. The scheduler will solve the problem by creating daily plans for nurses, as described above. A nurse will be automatically notified about coming tasks. The middleware will process the scheduled task on the defined time point, and transmit the task to an appropriate (and free) nurse. As a result of scheduler implementation and integration, the daily work plans will be generated dynamically by the middleware.

3.2.3 UCR3: Event bus

Some activities within the treatment process are event-driven, for example: a doctor can finalize the patient analysis only when sample analysis was done and reported by a nurse. Therefore, the middleware must support events. Like in CR3, the middleware should build a centralized event bus, which delegates all incoming events to corresponding subscribers. Hence, the doctor will have the possibility to subscribe (and be notified) to patient's case history events: case

history updates (changed), sample analysis results ready. In the same way, the doctor will be notified if the patient reports current health state, and the doctor participation is urgently needed.

3.2.4 UCR4: Context handling

The execution of many services is depending on the current activity context. For example, medication should be done by a nurse who is currently free, with prerequisites that this nurse can perform the required task; the other important point is that the nurse location should be the closest to the patient. Based on this information, the most appropriate nurse will be selected by the middleware. This is why we state that mobile services can be context-aware (see Chapter 1). Therefore, the middleware should gather actual context data of available services. The gathered context should be handled to support efficient routing (context-based routing).

3.2.5 UCR5: Logging and statistics

Due to law regulations for traceability of all activities, the middleware should protocol all communication actions. The provided logs can be used then for generating statistics, processes analysis and monitoring.

3.3 Technical requirements

Nowadays, many implementation technologies for mobile devices are available. The same solution can be developed with different technologies and programming platforms. So, for example, a doctor during patient's analysis and control can use an application on the desktop computer (e.g. realized with Microsoft .Net or Java), or by using a web application within intranet (e.g. Adobe Flash, Microsoft Silverlight, JavaScript), or by using any mobile device like a tablet or smartphone (Objective-C, Java, .Net). All these solutions have own advantages and disadvantages. The same user action will result in different formats of data

and different communication protocols. Thus, it is required to find a reasonable “common denominator” or application-independent format that will provide the same data appearance and is available for any kind of chosen implementation technology[1]. In this section, the common criteria for a middleware that can communicate with multiple programming (mobile) platforms, legacy systems, BPMS and patient administration systems will be defined. Accordingly, this section provides technical requirements (TR) for the researched middleware.

3.3.1 TR1: Standard protocols

Widely used protocols like HTTP or TCP are supported on all kinds of devices and client platforms. HTTP was an enabler for the Internet and so the most popular protocol of the world. Each programming platform supports the sending and receiving of HTTP-requests and of HTTP-responses. However, not each platform is suitable for running a web server.

3.3.2 TR2: Cross-platform clients

For example, a TCP server can be implemented on the Android platform[2], or for iOS[3], except for Windows phone. Web based clients and service calls are supported by all three platforms. In other words, a doctor can control a patient treatment and update the medication plan by using a web page within intranet. Thus, the usage of the middleware for mobile task coordination and execution should be possible from any platform: cross-platform clients are the key criteria for an interoperable middleware. That means, a possible client can be implemented for any platform: as Web, mobile or desktop computer solution (see CR2). Consequently, the data format should allow interoperability, since every platform uses a different programming language: a message that was sent from Java code should be understood by C# or by Objective-C code. In Chapter 2,

[1] Theoretically, all data formats and communication protocols can be available by adding related extensions. This implies additional development costs, depending on extension supplier, or performance, stability decrease. So, it will be observed only those standards that allow development "from scratch" - without adding/installing extensions

[2] http://developer.android.com/reference/java/net/ServerSocket.html, last visited 23.09.2012

[3] https://github.com/robbiehanson/CocoaHTTPServer, last visited 23.09.2012

the interoperability can be achieved through the Messaging pattern. As a result, in the rest of this thesis, a message-oriented middleware will be designed.

3.3.3 TR3: Dealing with communication problems

Another advantage of using a message-oriented middleware is the ability to deal with communication problems during communication with stateful services: if it is impossible to open a waiting socket, regular connection outages require additional techniques like “server push” or client polling for non-permanent communication. Since a MOM is based on queues, messages can be saved if a receiver is not accessible.

3.3.4 TR4: Context recognition and handling

Since discovered mobile services are context-aware, flexible context recognition and dynamic context handling on the middleware side is needed. Therefore, a service should be able to post and update own context data on changes. Intelligent context handling can contribute to better task execution: for example, a doctor uses a mobile device and a desktop client at the same time. The desktop computer has a greater display size compared to a mobile device. Also, connection used by a desktop computer is faster and persistent compared to a mobile device. If both are available, the better choice would be to transmit the task to a desktop client.
A flexible context storing format is necessary: a mobile phone can have multiple sensors, and each sensor can have multiple undefined properties. Even components can be divided into subcomponents with further subcomponents. An exact definition of context data structure cannot be defined in advance. Storage of context structure and data could be realized with a relational database management system (RDBMS). Other choices for these purposes are techniques from the semantic web: RDF (Resource Description Language) and SPARQL protocol and RDF query Language (SPARQL). These techniques are more appropriate, because RDF has features that enable data merging even if the underlying schemas differ, and it specifically supports the runtime evolution of schemas

without requiring all the data consumers to be changed [17].
RDF is a W3C[4] recommendation. A RDF model consists of so-called triples: Subject, Predicate and Object (cf. Figure 3.5) [15]. A triple is an expression where the *Object* characterizes the *Subject*, and the *Predicate* defines the kind of relation. In a RDF structure, so-called graph, a Subject can be described

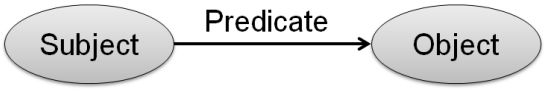


Figure 3.5: RDF triple

through variable of characteristics (*Objects*). Again, each Object can be characterized through own relations (*Predicates*): from an Object it obtains a *Subject*.

3.3.5 TR5: Location recognition

Location of service providers helps to select the most suitable task executor: for example, the most suitable nurse for the medication task is the nurse who is closest to the patient (least distance to the patient). Other nurses do not need to go to the patient, if one nurse is already there. This function of the middleware will save time and makes the task processing more efficient. The common location determination method could be realized via the global positioning system (GPS). The GPS receives from a satellite global coordinates (Latitude and Longitude) of the current position of the sensor. The main problem the GPS positioning is that GPS requires direct visual contact with satellites. For location determination in buildings, these methods are not applicable. The current position of a person (in this case of a nurse or of a doctor) can be made through the definition of according departments, or sectors (areas). Another possibility is to define architectural plans of a building (clinic) and rooms in which the person is. The distance could be calculated with special algorithms. If it is not necessary to have high accuracy in distance calculating, the best way is the manual definition of the current position (definition of department). The manual input can be automated through the usage of NFC modules, or scanning of QR (Quick Response) codes. The

[4]W3C - World Wide Web Consortium

location recognition methods are not part of this thesis. In the rest of this thesis manual positioning will be assumed.

3.4 Subsumtion

To gain an overview of all defined requirements, scenarios and qualitatively criteria, the following table sums ups all discussed points:

CR1(Communication): The middleware should enable communication between two or more separated and distributed software components.	**UCR1(Queuing)**: The middleware saves tasks until an appropriate service provider will be available.	**TR1(Standard protocols)**: Used communication protocols should be available on every platform.
CR2(Cross-parties communication): Communication possibilities are the same for provider as well as for requestor.	**UCR2(Scheduler)**: The service requestor can define the time point of task processing.	**TR2(Cross-platform clients)**: Required middleware connectors can be implemented for any platform.
CR3(Central bus): Any client will communicate only with the middleware – the middleware hides all additional communication protocols.	**UCR3(Event bus)**: Events have to be gathered and processed. The middleware builds a centralized event bus, which delegates all incoming events to event subscribers.	**TR3(Dealing with communication problems)**: Additional techniques for non-permanent communication like server push and client polling are required.
CR4(Routing): The routing is implemented in the middleware and does not require to be considered by the service requestor.	**UCR4(Context handling)**: The middleware gathers actual context of service providers to support efficient task transmission.	**TR4(Context recognition and handling)**: A service should be able to post and update own context data on changes.

CR5(Service transparency): The middleware hides all details of a service implementation and provides a consistent view for all underlined services – clients do not differ between mobile and other service implementations.	**UCR5(Logging and statistics)**: The middleware protocols all communication actions. Logs are used for traceability, monitoring, statistics, and analysis.	**TR5(Location recognition)**: The middleware requires explicit location disposition definitions.
CR6(Repository): Listings of available services can be used for administrative purposes.		

Table 3.1: Requirements overview

3.5 Middleware definition

This section summarizes the scenario and the requirements analysis and defines necessary functionalities and criteria for the middleware to cover requirements and scenario goals.

Another question related to standard or individual solutions is which kind of these can be closer and faster integrated in an existing environment? Standard software defines standard processes and supports standard tasks. Some incompatibility during process execution means a re-design of existing processes. Also, the technology plays an important role. Is this supported in environment's technology or platform? If not, it is necessary to re-develop related parts or components. If there is a need for many points of re-work, it is important to define the role of standard solution to be introduced.

The approach studied in this thesis belongs to emerging technologies. The mid-

dleware market has no already implemented solutions, which can be named as de-facto standard. Existing approaches and middleware solutions (for wired services) can be adapted to the problem areas defined in this thesis. This is what can be called as application-oriented middleware. The application-oriented middleware is concentrated on concrete challenges and problem solving of a given use case [13]. But, the adaptation process for another use case (usage context) may require implementing additional communication possibilities or additional middleware logic. This adaptation will take more effort as specialized research and realization of approach that was described in this thesis.

To sum up all requirements and usage scenarios, this section describes concrete properties of a middleware that will be used for mobile task coordination. It should allow integration with existing systems like PAS that protocols treatment, or BPMS that defines a global process and controls the execution.

From one side, the middleware should have the possibility to be integrated within an existing environment with different systems. Existing systems can be of different kind (e.g. BPMS, ERP or administration systems) and can be implemented with different programming platforms (.Net, Java, C++, Python). Accordingly, the middleware solution should be interoperable and provide multiple communication scenarios and be able to act as a provider and as a requestor. For these purposes, Web services can be used for interactions between a system and the middleware. By using Web services, the necessary interoperability can be achieved, and the integration of the middleware can be done simpler. This is because many systems nowadays have support of Web services, or are developed based on service orientation approaches. The integration through Web services can be achieved through the utilization of Web service technology, and every extension can be made relatively simple and fast (in comparison with hardcoding of components and manual development of client/server stubs). The challenge remains to find an appropriate Web service technology. SOAP[5] and RESTful services are two possibilities. To make this decision, we will analyze the approximate duration of service calls. As described in Chapter 2, REST services are more applicable for resource-oriented environments compared to SOAP services. Within REST, the execution takes not much time, because resources are

[5]SOAP - Simple Object Access Protocol

already available. The time to execute a REST service is equal to the time to find and serialize a resource. SOAP services are remote functions and therefore can have a complex logic inside. Thus, the SOAP services are, theoretically, more suitable for long running processes and, accordingly, for durable communication. In any case, the communication between legacy systems and the middleware should be asynchronously, because tasks can take up to several hours or days to complete the execution. In such cases, even SOAP services cannot wait in a two-way communication model. This is why the researched middleware must be message-oriented and asynchronous. To solve this problem, we will observe it from another point of view: task definition, which is a resource. By creating a resource (putting a resource to the middleware), the middleware will, by itself, begin with processing of the message. By definition, the middleware deals only with communication between components. So, the middleware will begin with forwarding of the message to a task processor (task executor). This task processor should recognize the task and execute it through appropriate services. The legacy system can request the actual state of the task processing. From this point of view, using REST, the middleware access interface is more intuitive and provides fewer problems during implementation (e.g. there is no dealing with connection's timeouts compared to SOAP services).

From another side, the middleware will mainly communicate with mobile devices with limited network access. These mobile devices should receive instructions (task notifications) from the middleware and answer with reports when the work (task) is done. Due to the fact that not all mobile platforms can host accessible services, the notification should be done through server push technology (see Section 4.3). Also, mobile devices should provide their context information regularly: who is using the device (nurse or doctor), what is the current location of the device, and so on.

The interface for using the middleware from mobile devices is more complicated, because some devices can also act as enablers for task execution (e.g. MARPLE process engine for supporting mobile collaboration [20]). Services should have the possibility to login and logut to/from the middleware and periodically provide context information. Also, mobile services should be able to receive tasks. Some services, which act as service requestors, should be able

to execute a service and be notified when this task was done.
The following list defines required functions for mobile service providers:

- Check-In/-Out – Through this interface a mobile device informs the middleware about its availability: either the device is online (check-in) and can perform tasks, or the device is offline (check-out).

- Context update – A mobile device notifies the middleware about any context changes: e.g. a mobile device was connected to an ECG device, or the person who, owned the device, has moved to another location.

- Feedback – Device informs the middleware about the current task processing state: the middleware can discover the task processing states and reassign tasks to other providers (if these were interrupted or not not supported by a previous receiver).

- Publish – This interface is part of an asynchronous event pattern called "Publish-Subscribe". Through calling of the publish-method, an event provider (service provider) will post data to the middleware (see Section 4.4.3).

- Subscribe – An event subscriber defines "Events-Of-Interest (EOI)" and will receive event data (see Subscribe-method in Section 4.4.3) if the published event matches the defined criteria.

3.6 Middleware architecture

In this section, concrete middleware architecture will be introduced. The discussed middleware architecture implements all defined criteria and requirements with architecture components.
The reference architecture (cf. Figure 3.6) of the researched middleware can be divided into five component types, based on their roles: core components, context management, event handling, task processing, and external components.

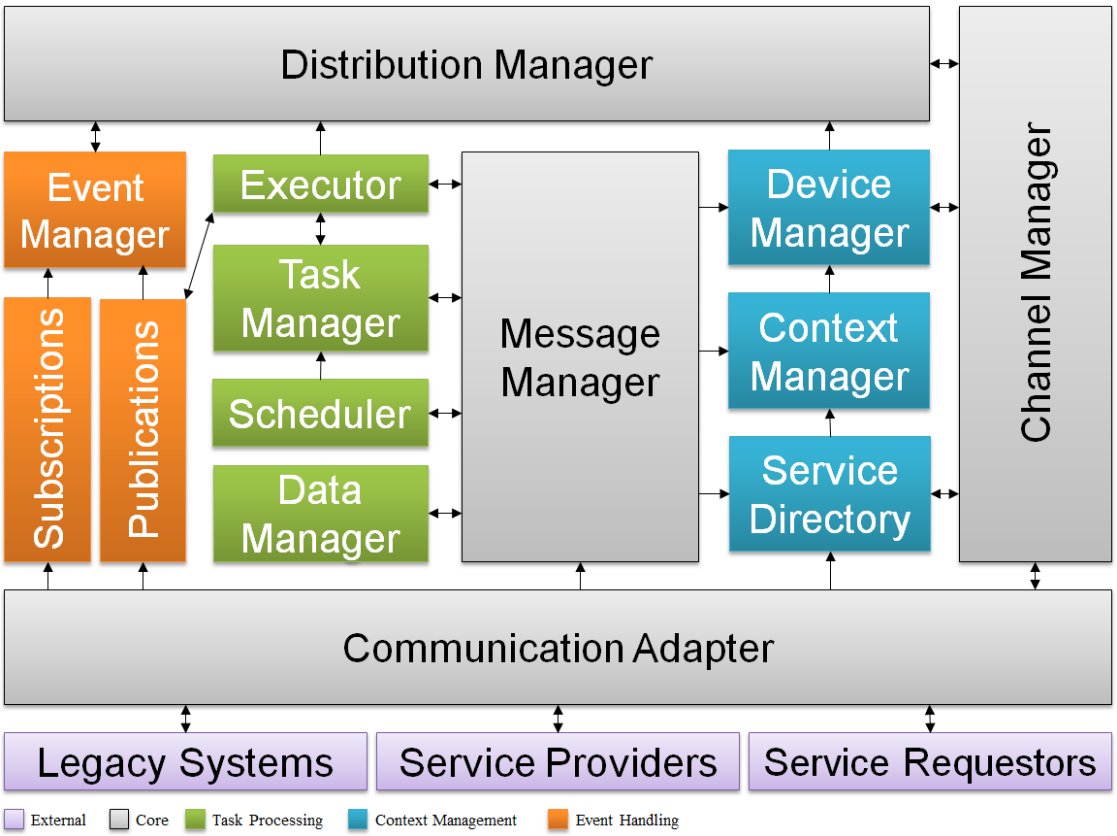


Figure 3.6: Reference architecture of the middleware

3.6.1 External components

The external components represent components that are not part of the middleware implementation but form an integral part of the overall system. An external system could be a BPMS or Patient Administration System; it can initiate, control, and log the execution (in case of BPMS); gather and protocol actual patient's data (in case of PAS). Service providers are mobile services or Web services. Service requestors are any kind of clients that initiate a remote execution of tasks or remote data access.

3.6.2 Core components

Core components of the middleware represent centralized resources that are used for communication initialization (CR1), message queuing, distributing, and data delivery (UCR5) through request processing. The name is derived from

the common middleware definition provided in Section 2.2: the core function of a middleware is to ensure the communication between several distributed parts (CR3). Accordingly, the core components of the researched middleware are responsible for communication: Communication adapters make the middleware accessible. The Message manager is responsible for message recognition and message queuing, the Distribution manager assigns a message receiver based on its definition, and the Channel Manager handles connections for each distributed component.

The Communication Adapter consists of many access interfaces for accessing the middleware from legacy systems as well as from mobile devices (TR2). These interfaces can be implemented with any suitable (or necessary) technology to provide good integration between participated systems (distributed components) (CR2). In the researched middleware, these interfaces are implemented with REST and with appropriate protocols (see Section 4.3) for persistent communications (TR1).

The Message Manager de-serialize messages into an internal representation. As next, based on entrance interface definition, the message manager recognizes the type of the entry message and, additionally, forwards messages to respective message processing components (i.e. the Data Manager, the Task Manager, or the Context Manager). The second role of the Message Manager is the central message queue (UCR1). All tasks will be persisted within the message manager. All additional components (like the Scheduler, the Task Manager) will access and change persisted tasks in the Massage Manager and will not save tasks separately.

The Distribution Manager is responsible for distribution of tasks to the right task receivers (CR4). After receiving a message to be distributed; the Distribution Manager requests a list of appropriate receivers and takes the best candidates. After that, the Distribution Manager forwards the message with chosen candidates to the Channel Manager, where the messages will be passed to communication channels accordingly.

The Channel Manager holds all active connections (channels) and their states. If a connection is broken, the Channel Manager will try to re-establish it, or waits for new connection request and maps the new connection to the device (TR3).

Each external component can have several parallel connections to the middleware (e.g. event subscription and waiting connection for incoming tasks (service provider)). The Channel Manager handles all these open connections, and is responsible for the transmission of messages to the receivers. Also, the Channel Manager is responsible for conversion of the used message formats (CR5): a service client uses a Web Service (SOAP) interface and, therefore, requires a SOAP message as a result. If the requested service is a mobile service, which uses other standards for communication and messaging (see Chapter 4), the required SOAP answer and the original result are incompatible. So, the Channel Manager will provide result data in form of a SOAP message to the requestor by converting the original message.
The whole message processing and message-receiver mapping is handled completely by the middleware. During distribution within the Distribution Manager, it observes currently available and active devices (providers) only. Therefore, it is impossible to inform a receiver about possible incoming tasks in advance (forecast a distribution result).

3.6.3 Task processing components

The task processing components support the intelligent task management and cover message content and attributes processing. The Message Manager decides about the type of a message (based on the entrance interface) only, Task processing components, by contrast, are looking for details: priority and scheduling.
The Data Manager deals with message content data. Since mobile devices take part in communication (TR2) and have limited connection bandwidth, it is necessary to handle massive message content data in an effective way. Thus, the Data Managers removes unnecessary content data, and, by request, provides these to a requestor. As a result, the middleware utilizes relatively small messages and can quicker transfer these messages to a receiver. The possible problem at this point is how to exclude only optional data and keep required data. Described differentiation of data items should be done on the message sender side. So, a sender can define which data items are required and which are of interest for

the task receiver. The Task Manger includes content of required data items and URIs for accessing optional items. By an URI, a receiver of a task can then ask for an interesting data item by the Data Manager.
The Scheduler provides scheduling possibilities (UCR2). The Scheduler component plans the processing of a task - based on his scheduling definition. The scheduler is, in other words, an enabler for overall task processing: according to plan, the scheduler pulls tasks, which should be processed at the time point of pulling, and pushes these to the Task Manager for next processing. Tasks which should be executed at a later point in time are ignored by the Scheduler until this time point is reached.
The Task Manager is responsible for assembling of outgoing messages. The Task Manager will cut receiver definition parts from the message content, include mandatory data and paste URIs for optional items. The Task Manager controls the overall state of a message, and will also send an exception if, for example, the message was not understood by a receiver. Another feature of the Task Manager is the prioritizations of tasks with manually defined priority and defined time range of validity. Following rules can be noted:

- Tasks with defined distribution deadlines have higher priority than tasks without deadlines (if a priority is not defined manually, or priorities of both tasks are equal).
- If a priority is defined manually, the priority of tasks with defined processing time will be increased by 5.
- If the priority of a task with a defined deadline is not set, and the priority of a message without a deadline is defined, then the priority of a task with the deadline is 10.

The Executor takes care about currently executed tasks (messages): the Executor will receive all feedback notifications from task executors (service providers), and, if the execution was interrupted, try to reassign the task to another provider. The Executor, by receiving a task from the Task Manager, will initiate the processing of this task and control the processing until the end. After processing, the Executor will report the processing state back to the Task Manager. The processing state can either be “processed” or “broken”. As mentioned before,

the Executor will try to reassign interrupted execution automatically to a new receiver (until the task is still valid, else the state will be “broken”).

3.6.4 Context management

The main purpose of the context management components is the handling of service descriptions, devices’ (services) context data, and the availability of information for correct decision making (TR4): context management components provides required data to the Distribution Manager, which correlates the right executor to task processing requests. Therefore, the Service Directory provides all currently available services, the Context Manager holds all relevant context data, and the Device Manager summarizes all these data to concrete description of devices, device’s context information and available services on these devices. The Service Directory stores all available services (CR6). During a check-in, a service provider defines its provided services. This definition of hosted services will be saved in the Service Directory. Also, the Service Directory saves manually added Web Services. It should be noted that the Service Directory will save only distinct collection of services. That means that every succeeding publication of the same service will be ignored. A service definition is permanent and will not be dynamically changed. Therefore, saving of duplicates is not efficient and, consequently, not required. If a service is provided by multiple providers, the Context Manager will map this service to service providers. By processing of a task, the middleware will recognize the required service and selects it from the Service Directory. The unique identification number (ID) of the service will be used to select appropriate provider of the service. Thus, the selection of appropriate service providers will be done in the Context Manager.
The Context Manager binds services with providers (devices) and their context (UCR4). Hence, the main purpose of the Context Manager is to dynamically handle context changes. The device context structure is variable: some device can have a camera, some devices have NFC sensors, and some devices do not have any of them. This context structure is represented as a RDF graph.
As shown in Figure 3.7, the start point of the RDF graph is a device. The device (Subject in RDF triple) can have many variable properties: hosted services,

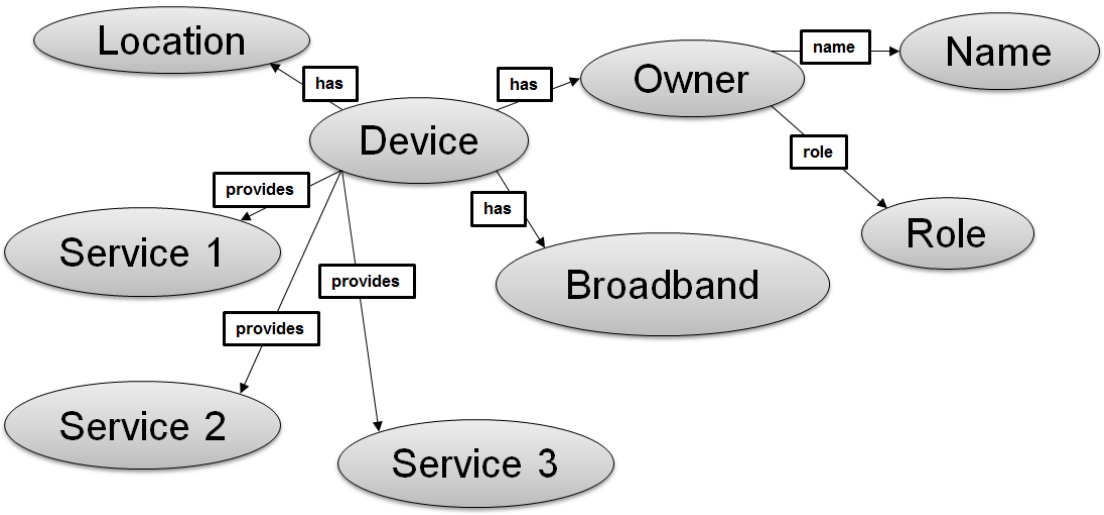


Figure 3.7: A possible RDF graph for device context representation

broadband, location, display size, owner. Each of these properties can be self a subject and have a substructure: for example, the display size can be refined with height and width. Again, each property is a Subject.

The Device Manager will match the context pattern and generate a list of appropriate (and available) service providers. This will be done in two phases: selection of appropriate candidates (Phase I) and prioritization of candidates (Phase II). In phase I, the Device Manager generates a list of appropriate providers by selection of providers that match the defined criteria. Inside the task definition, a requestor defines not rules for searching of appropriate providers, but rather a pattern of a required provider. The Device Manager tries to find the provided pattern in all available graphs (cf. Figure 3.8). Additionally, the Device Manager requires to handle device states: a device can be *free* or *busy* with processing a task (service request). So, only "free" providers are taken into account during the matching.

During the Phase II, the Device Manager will sort the list of service provider candidates by provider performance and by level of experience of the providing device owner (cf. Figure 3.9). At first, the Device Manager will prioritize all Web Services, because they are running on stationary computers: stationary computers (servers) are very performant and have less dependencies. As next come service providers, which owners are experts in their area: for example,

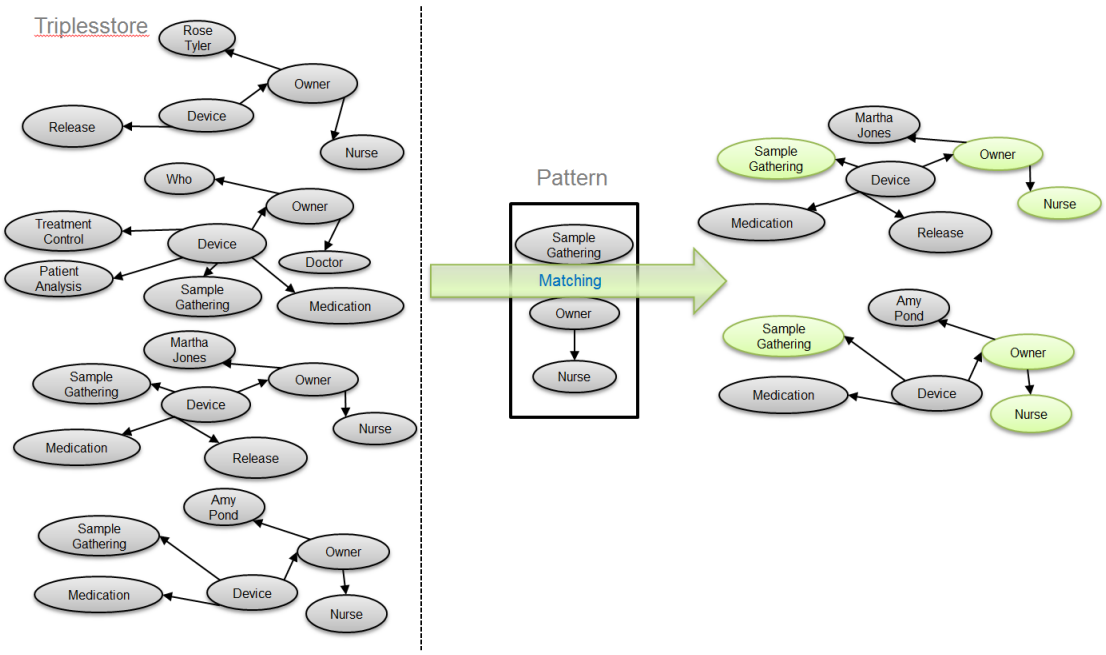


Figure 3.8: Pattern-based matching of graphs

a nurse who makes only sample gathering will be prioritized than a nurse who provides many services. This prioritization is defined for performance issues: a person that can do some special tasks, should be free and not be distracted by a (common) task which can be done by other persons. Another task of the Phase

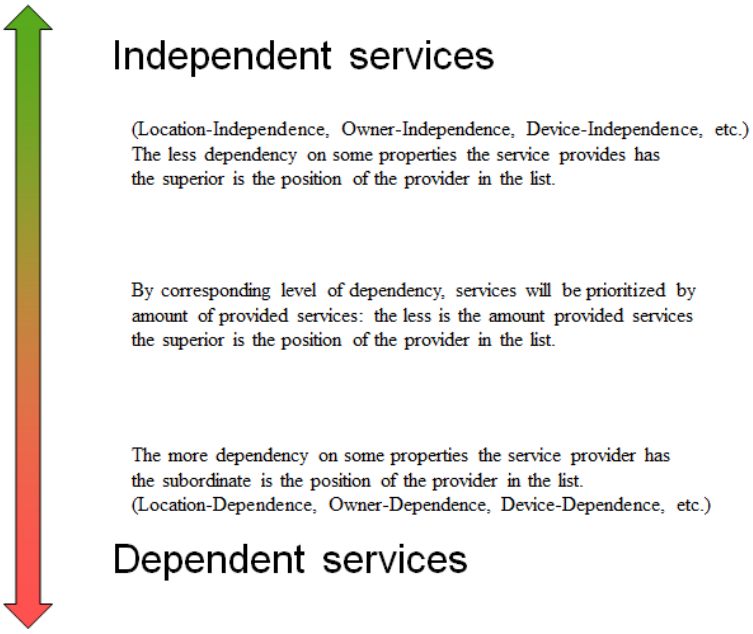


Figure 3.9: Provider ranking

II is the determination of concrete characteristics: as example, calculating of the distance and selecting the closest provider (TR5).

3.6.5 Event handling

The Event handling components provide event support within distributed systems. Many activities in the discussed use case process are dependent on the execution state of preceding or sub activities (UCR3). Since the middleware supports only asynchronous communication, the used event model should be asynchronous also. Thus, the event handling is realized according to the asynchronous publish-subscribe pattern.
The Publications component stores all incoming events or change publications. Publications can be manual or automatic. Manual publications are those publications, which were made explicitly by using the publication interface. Automatic publications are those publications, which were added by the Executor: events like start or end of execution will be added by the Executor who controls the execution of a task.
The Subscriptions component handles subscriptions. A subscription can define either an event provider of interest (provider-based) or a content of interest (content-based). Additionally, an event subscriber can define a time point to filter publications before the time point: all messages which are “older” then the defined time point will not be taken into account. If the time point is not defined, the middleware will transmit all events without considering the age of the events. This can be helpful for monitoring, logging, and statistics (UCR5).
The Event Manager maps event publications to event subscriptions. Like the Device Manager, the Event Manager matches the subscription pattern with publications to select appropriate events. The distribution of events to subscribers is made by the Distribution Manager.

4 Specification

This chapter provides specification details of the researched middleware. Also, this chapter describes communication patterns for concrete use cases from the scenario analysis (Chapter 1) as well as from the requirements (Chapter 3). The challenge of this specification is to find a reasonable way for interoperable communication between heterogeneous systems and platforms. The implementation must cover interoperable data formats as well as interoperable communications protocols. Main goals of this chapter are the description of an appropriate message format, as well as the definition of possible communication scenarios. Additionally, a concrete course within a clinical scenario with the researched middleware will be shown. Finally, technologies and techniques for the implementation of the middleware will be observed.

4.1 Message template

Main point of any communication is information transmission. In order to support communication between heterogeneous systems, interoperability is an important aspect. Interoperability of data formats means that information, generated by one component and transmitted to another component, will be understood by original and will have the same meaning. XML is the most used standard for format interoperability. It provides data structuring possibilities and is supported by many programming platforms. Because the goal of the middleware is the interoperable communication between different entity implementations, the message format of the researched middleware is based on XML.

4.2 Message structure

In Figure 4.1, the structure for a communication message between the middleware, service requestors and mobile device is proposed.

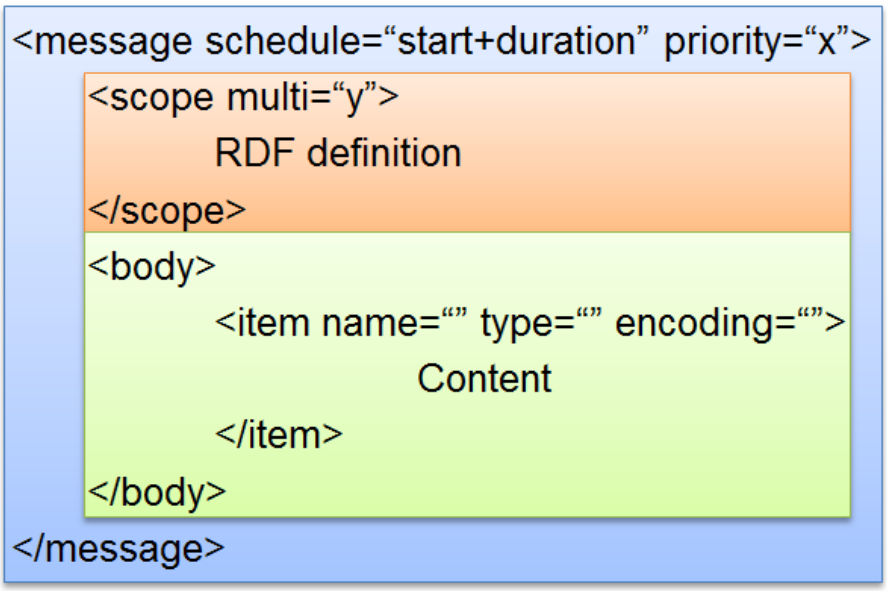


Figure 4.1: Message structure

4.2.1 Message

Like any XML document, the message structure begins with a root element (*message*-Element) and the following attributes:

- The **schedule** attribute defines the valid time range for a message. A valid time range describes the time interval (start point and timeout) when a message is processed (distributed) by the middleware (see UCR2, Section 3.2.2). The schedule attribute is optional and should be used when a specific time interval for message processing is required (e.g. for a medication that should start at a concrete time point the scheduling (processing)). The definition consists of two parts: start point and timeout (separated by a "+" sign). The start point defines a time point when the middleware begins to deliver the message. The timeout defines the end of the time range

(deadline) when the message can be delivered to a task executor (service provider). In Table 4.1 example properties and their semantics are described:

Syntax	Start	Timeout	Semantic
Empty (attribute is not defined)	a.s.a.p.[1]	Not defined	Processing of the message will start as soon as possible and will be processed until the contained task is successfully done.
14.07.2012+1440	14 Jul 2012 at 00:00:00	1440 minutes	Processing of the message will start on 14 July 2012 at 00:00:00 and the middleware will try to transmit the message until 15 July 2012 00:00:00. If the message will not be transmitted to the receiver, the message will be ignored by the middleware.
+1440	a.s.a.p.	1440 minutes	Processing of the message will start as soon as possible (start point will be set to the time point of message entry), and the message will be rejected in exactly 24 hours.
14.07.2012 12:00:00	14 Jun 2012 at 12:00:00	Not defined	Processing of the message will start on 14 July 2012 at 00:00:00, and will be processed until the contained task is successfully done.

Table 4.1: Semantic of schedule usage

- The **priority** attribute defines the priority of a message. In situations, when there are many messages (tasks) in the message queue (3.6.2), which should be processed (executed) within one time range, an ordering of the message processing is required. In such cases, messages can be ordered by their priority: messages (tasks) with higher priority will be processed (executed) earlier than messages (tasks) with lower priority. Higher value of the priority attribute corresponds to higher priority of the message, and vice versa. As mentioned in Section 3.6.3, the priority of a message is increased by 5 if the concrete execution time interval is defined. Further-

[1] a.s.a.p. - as soon as possible

more, if the priority is not defined, but a time range definition exists, then the priority is 10. The priority will be manipulated in these two cases only. When the priority is not defined, messages are processed in first-in-first-out (FIFO) order.

- The identifier (**id**) is required to correlate a response to the request. This attribute will be set by the middleware, and is used in messages from the middleware to a service provider (executor).

4.2.2 Scope

The scope part of the message is dedicated for the middleware only. It describes the task's execution scope, which should be granted by a potential receiver. This element will be parsed by the middleware and then will be removed from the message structure before the message is transmitted to the receiver.
The main purpose of the scope section is the definition of constraints for service providers. So, the scope is responsible for providing a necessary matching pattern for the Device Manager (see Section 3.6.4), which provides a list of service providers based on this pattern.
The **multi** attribute of the scope element is used to define an amount of required receivers (e.g. two nurses are required to transport a patient to a laboratory for sample gathering and sample analysis). The default value is one. The Device Manager will check the amount of available (and appropriate) service providers. The task will be processed, if there are enough providers available. In another case, the distribution of the task will be postponed.
The pattern will be defined in RDF notation (RDF/XML [3]). RDF/XML is a XML format to encode RDF specified by the World Wide Web Consortium (W3C).

```
...
<rdf:Description>
 <rdf:type rdf:resource="http://middlewarehost.de/schemas/
    service" />
 <service:name>Sample gathering</service:name>
</rdf:Description>
<rdf:Description>
```

```
 <rdf:type rdf:resource="http://middlewarehost.de/schemas/
    owner" />
 <owner:role>Nurse</owner:role>
</rdf:Description>
...
```

Listing 4.1: Sample pattern definition in RDF/XML

The XML fragment in Listing 4.1 represents a possible description of a matching pattern to find an appropriate service provider. This fragment defines a service with the name “Sample gathering” and an owner in the role “Nurse”. In other words, this fragment defines a nurse, which provides the “sample gathering” service. The pattern definition is partial: a client defines only necessary parts of the service context.

In RDF/XML, there are two types of nodes: resource nodes and property nodes. Resource nodes are the subjects and objects of statements. Usually, they are *rdf:Description* nodes. Resource nodes contain property nodes only. Property nodes represent single statements: the subject is the outer resource node that contains an object. The predicates are node names of the object node, if the statement has a literal value. Else, if the object is a resource, predicates are wrapper nodes of a represented resource. In the Example in Listing 4.1 there are two statements: service (subject) name (predicate) is “Sample gathering” (object), and owner (subject) role (predicate) is nurse(object). The *rdf:type* node denotes the resource type (in the example above: service and owner). The *rdf:type* is usually be abbreviated by replacing the *rdf:Description* node (see Listing 4.2).

Another task beyond the scope is to provide the service context. Unlike partial pattern definition, the service provider needs to define the complete graph of the context and the service description. As soon as the service check-in message is arrived, it will be processed by the middleware. If the processing is completed, the service is immediately available for execution. Incomplete service context information (e.g. without location or owner information) definition will affect the request distribution, because the distribution is based on the provided context, and insufficient context may result in service execution low quality or in unex-

pected effects. For example, a service is provided by a doctor, who has not defined the device owner information (or without owner role). The middleware will recognize them as an appropriate provider for the service if a doctor (role = doctor) is required. And, when it is a single provider, the task will not be executed. To prevent such problems, a context definition must contain a full service description. Figure 4.2 illustrates the RDF graph from Section 3.6.4.

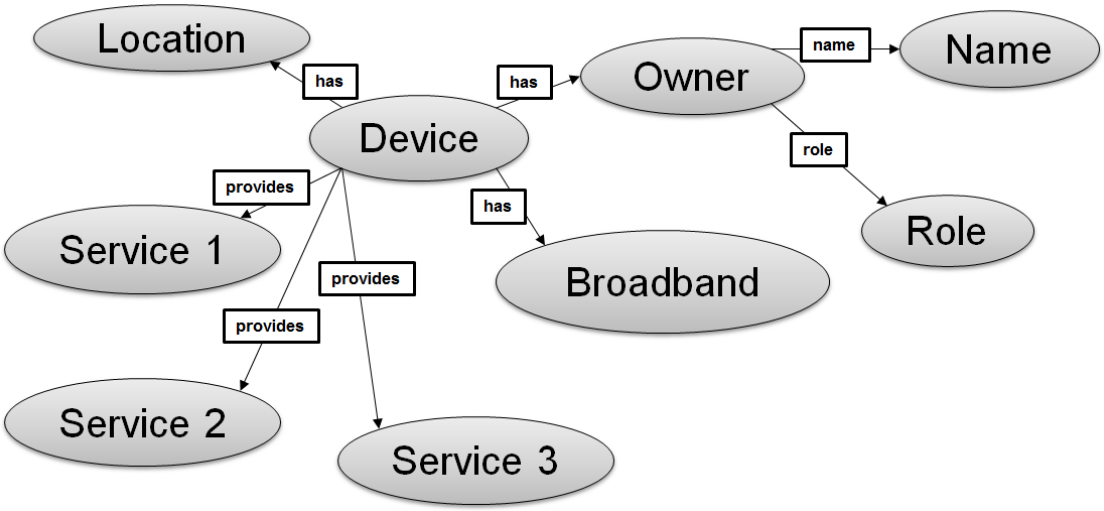


Figure 4.2: Sample device RDF graph

The context definition has to be defined as follows:

```
...
<rdf:Device >
 <device:has>
  <location:department>Surgery</location:department>
 </device:has>
 <device:provides>
  <rdf:Service>
   <service:name>Sample gathering</service:name>
  </rdf:Service>
  <rdf:Service>
   <service:name>Medication</service:name>
  </rdf:Service>
   ...
```

```
 </device:provides>
 <device:has>
  <rdf:Owner>
   <owner:name>Who</owner:name>
   <owner:role>Doctor</owner:role>
  </rdf:Owner>
 </device:has>
 <device:has>
  <device:broadband>3G</device:broadband>
 </device:has>
</rdf:Device>
...
```

Listing 4.2: Context definition example in RDF/XML

4.2.3 Body

The *body* element of the message contains the data payload. The body can contain multiple *data items*. These data items represent concrete values (parameters) which will be delivered to a receiver. This receiver can be a service provider, which requires parameters, or a service requestor, which waits for a result.

Any service call within the observed middleware usage is document styled: a service client defines a list of data items without connection to a concrete service specification. A service provider selects required data items from the list of provided items (e.g. by name and type) and executes the service implementation with these parameters. The document styled messages are suitable for scenarios, where data elements cannot be modeled in advance. For example, an experienced doctor requires less information than a novice (for manual tasks or decisions). Automated services (e.g. Web services), in contrast, are less context-aware than manual services (human tasks). As described in Section 3.1.5, the middleware does not distinguish between mobile services (manual tasks) and Web services. To ensure this, a developer should provide all data

that is required for a Web service call as well as for an execution of manual tasks (mobile services): a client sends all available data, an executor chooses which data are relevant and required for the execution.
Data items are defined by using the *item* element within the body. In other words, a single data item is represented through a single *item* element, and has three mandatory properties:

- The **name** attribute defines the name of a data item.
- The **type** attribute defines the type. The type, can be user-defined and is represented as a string. To provide better interoperability, MIME-Types [4] should be used.
- The **encoding** attribute defines how the value is encoded. The supported encoding can be a plain text (text), XML structure (xml), or Base64-encoded binary value (base64). Plain text is represented as a CDATA element.

4.3 Communication protocol

Web applications are faced with similar problems compared to the researched middleware. Communication between the middleware and a client (service provider, service requestor) can be initiated from the client side only. Connections can be broken. Therefore, we will survey communication protocols from the Web area.

4.3.1 HTTP

Hypertext Transfer Protocol (HTTP) is the fundament of World Wide Web (Internet). HTTP is limited to the request-response communication model: a client sends a request and a HTTP server answers with a response. In the discussed middleware scenario (see Section 4.4.1), the middleware acts as a server and notifies a service provider about service requests (task transmitting to an executor). Due to the fact that the communication can be initiated from client side only,

implementation of such scenarios is not trivial. For these purposes, developers have created many approaches, which belong to the wrapper term ”Server push”: HTTP persistent connections, Long Polling [24], Reverse Ajax[2] (Comet) [18], BOSH (bidirectional-streams over synchronous HTTP)[3] [24].
HTTP allows the definition of persistent connections by setting the ”Connection” attribute of the HTTP header to ”Keep-Alive”. In this case, the server and the client will keep the entire connection opened, instead of closing the connection after receiving a response by the client. Following this, the connection will not be closed, and the server and the client can continue to transmit data. In the provided case, a service provider can request a persistent HTTP connection by registration (check-in), and hold the connection opened until a service request (task) is received. Also, by responding, the connection is used for receiving the next request. The main problem of this technique is that server implementations and browsers have an internal timeout: after expirations of this timeout, the connection will be broken.

Long Polling

To deal with connection timeouts, the long polling mechanism [24] was developed: requests will be sent continuously, if the previous request was broken or a reply was received, the client will send a next request (cf. Figure 4.3). By continuously sending of requests, the client receives actual data changes and notifications from the server.
The long polling mechanism is the simplest and most used technique for server push in Web development[4] [24, 18]. The usage for long polling technique for communication between mobile devices and the middleware can provide the required cross-platform support by the middleware: a tier can be implemented as a mobile application, as well as a web application.

[2] Ajax - Asynchronous JavaScript and XML
[3] http://xmpp.org/extensions/xep-0124.html, last visited on 12.10.2012
[4] Even traditional Ajax is based on "polling" technique

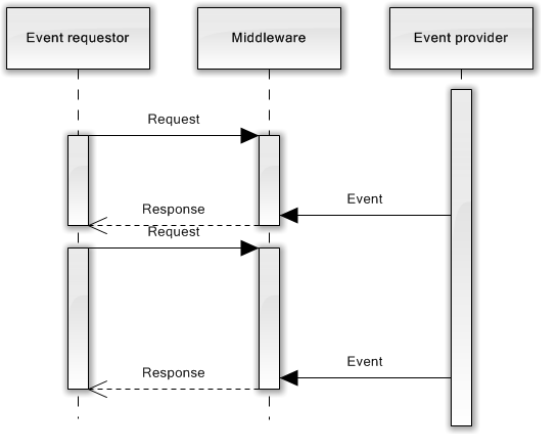


Figure 4.3: HTTP Long Polling

4.3.2 TCP

Transmission Control Protocol (TCP) is a connection-oriented transport protocol that allows reliable, persistent connections between two points. After establishing a connection (Three-Way-Handshake), both connection points can transfer, independently from each other, any data (also streaming) in bidirectional mode. Theoretically, the usage of TCP is the best choice for implementing communication between the middleware and service providers/requestors. The main problem is that TCP communications are not supported by web browsers. Therefore, the implementation of clients or service providers as web applications is not possible.

WebSocket Protocol

To enable support of two-way communication for browser-based applications, the WebSocket Protocol was developed [9]. The WebSocket Protocol provides a single connection for traffic in both directions: a client can send data to a server, and the server can send data to the client at any time. The WebSocket Protocol allows building of web applications like games, chat messengers, or billing applications. It is an alternative to HTTP (long) polling for two-way communication [9]. Conceptually, WebSocket is a layer on top of TCP [9]. It is designed in such a way that WebSocket servers can share the 80 port with HTTP servers, by having its handshake be a normal HTTP request (Connection: Upgrade)[9]:

”The WebSocket Protocol is an independent TCP-based protocol. It’s only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request.” It means that, in contrast to common TCP three-way handshake (SYN-ACK handshake), the based WebSocket connection will be established by sending a normal HTTP request. The WebSocket Protocol is supported in WebKit-based browsers (Google Chrome (16), Safari(6)), Opera from version 12.50, Firefox from version 11, Internet Explorer 10. Also, implementations of the WebSocket Protocol are available in many programming languages: client libraries as well as server implementations.
The protocol consists of two phases: the handshake and the data transfer.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Listing 4.3: Client-side handshake request

Equal to HTTP, the client specifies which resource and which host he wants to access (see Listing 4.3). The randomly generated ”Sec-WebSocket-Key” supplies identification and authentication to the server, and is used to check whether the server actually reads and understands the request. With the ”Sec-WebSocket-Protocol” field, the client has the opportunity to specify additional subprotocols (here: chat). Subprotocols are extensions based on the WebSoket Protocol: for example, message broker subprotocol[5], Extensible Messaging and Presence Protocol (XMPP) subprotocol[6], RPC or Publish-Subscribe.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

[5]http://tools.ietf.org/html/draft-hapner-hybi-messagebroker-subprotocol-03, viewed 14.10.2012
[6]http://tools.ietf.org/html/draft-moffitt-xmpp-over-websocket-01, viewed 14.10.2012

```
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Listing 4.4: Server handshake response

The server agrees to protocol change and replies with the HTTP 101 status code (see Listing 4.4). The returned key in ”Sec-WebSocket-Accept” is the verification that the server has read the client request.

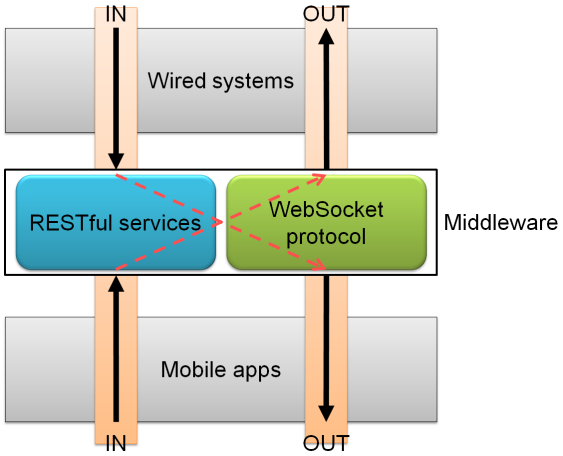


Figure 4.4: The communication protocols correlation

After the connection was established, the remote points can transmit data. The connection can be terminated by a normal FIN-ACK three-way handshake [9].
As illustrated in Figure 4.4, the middleware uses RESTful services for all incoming requests (see Section 3.5). The usage of RESTful services allows accessing the middleware from all possible client implementations: Web applications, desktop applications, and mobile applications. The WebSocket protocol is used for the delivery of requests and notifications to receivers (service provider). This combination provides simplicity for developing middleware access clients, and fast, reliable message distribution to task executors. To achieve a high degree of flexibility, each communication way should provide all functionalities of the middleware (sending task requests by service clients, receiving requests by service providers, events and notifications). So, the developer can choose between different communication modes: RESTful services, WebSocket protocol, or a combination of both (mixed).
Often, during programming a concrete solution, a developer has limited range

of implementation technologies. The usage of concrete technology is dependent on design decisions and availability of required libraries. Therefore, a definition of possible middleware usages is required. In cases, where a solution is Web oriented, and the implementation of Web parts is required, it is advisable to use only Web conform standards, protocols, and techniques. To ensure compliance to Web standards, the researched middleware should provide all necessary interfaces as RESTful services. This operating mode will be called "**RESTful mode**" or "**Web mode**" in the following.
In other cases, where handling of continuous request loops (e.g. Long pooling) seems to be complicated, and the usage of Web conform standard is not required, the developer could use persistent connections ("**Persistent mode**" or "**WebSocket mode**"). The only technical difference between Persistent and Web modes is that RESTful services are provided directly by the Communication Adapter, whereas persistent connections are handled by the Channel Manager. The Channel Manager will forward all messages to the Communication Adapter and create a mapping between requests and corresponding communication channels.
The default operating mode is "**Mixed mode**", where communication is done partial in RESTful mode, and partial in persistent mode.
To provide extension possibilities for software solutions, which are based on the researched middleware and to increase interoperability, the middleware communication should be based on RESTful services or on WebSocket Protocol. Thus, a Web oriented solution can be extended with wired tiers that use persistent connection, and vice versa.

4.4 Implementation aspects

This section describes how the messages will be processed by the middleware. The researched middleware deals with four types of messages:

- **Context description messages**: in order to receive requests for context-aware services a service provider should provide actual context information to the middleware. Additionally, service providers should have the pos-

sibility to change their context information. Consequently, the middleware must provide an update interface. In both cases, the service provider sends context description messages: one during creation of an initial record, and the second for changing or updating this context record.

- **Service requests or task definitions**: the main purpose of the middleware is the coordination and distribution of tasks for mobile services. Therefore, request messages, which define data for service execution and task definitions, are essential part of the researched middleware.
- **Subscriptions**: for reacting on events, each middleware client can subscribe for specific changes (e.g. task processing state changes, availability of service providers, or current state changes).
- **Publications**: to make changes "visible", every change provider, or the middleware itself, publishes the change information with the help of publications. The provided information within a publication will be transmitted to the interested subscribers.

These four message types can be grouped into three categories: context information, task processing and service requests, and event handling.

4.4.1 Task processing and service requests

The task processing and service request part consists of five steps (cf. Figure 4.5): request creation, request delivery, processing feedback, processing result, and result delivery.

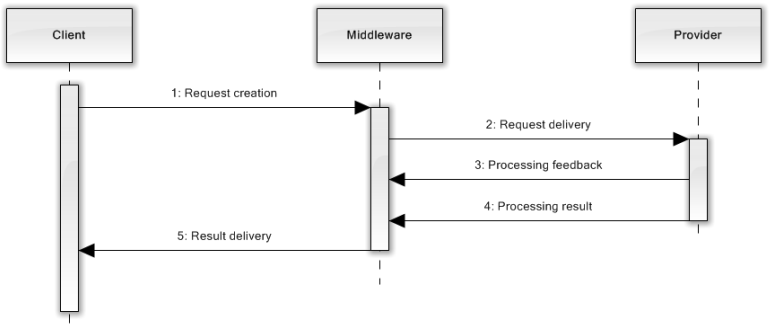


Figure 4.5: Task processing and service request (in UML Sequence Diagram)

First, a service request or a task will be sent to the middleware for further processing, coordination and distribution (**request creation**). Second, the (service or task) request is transmitted to an appropriate service provider (or task executor) (**request delivery**). Third, the provider regularly reports about current processing state (**processing feedback**). Fourth, the provider sends processing result to the middleware (the provider does not know anything about the requestor) (**processing result**). Finally, the result is delivered to the original requestor (middleware client) by the middleware (**result delivery**).

As described in Section 3.5, the interface for sending of tasks and service requests to the middleware will be implemented as RESTful service. The service address will be in the following form: *http://middlewarehost[:port]/requests*. To manage requests, the middleware RESTful service provides CRUD[7] operations. With the HTTP PUT method the client "creates" a new task. The message format requires, in this case, the service description ("*Scope*"-block) and related data ("*Body*"-block). First, the message is stored in the Message Manager (cf. Figure 4.6). The Message Manager de-serializes the received message into an internal representation. Only task definitions and service requests will be stored within the Message Manager in a central queue[8]. Second, the Scheduler regularly iterates through the queue and checks which messages should be processed. If the Scheduler finds tasks, which should be processed, it pushes these tasks to the Task Manager. Third, the Task Manager assembles an outgoing message: it includes values of mandatory data blocks and reference URIs for optional items. The next steps are the distribution of a request by the Distribution Manager and a control of the "execution" by the Executor.

After creating a task or service request, the middleware answers with an identifier of the newly created resource. This identifier is necessary to attach additional data items, to check actual processing status, or to delete the resource.

There is no special parameter within the data item specification to mark optional data: optional data item should be attached separately to the response through HTTP POST request to *http://middlewarehost[:port]/requests/{request identifier}* (or message type "update"). The task executor (or service provider) receives only mandatory data items. If further information is needed, the provider (or ex-

[7] CRUD - Create, Read, Update, Delete

[8] Context handling messages will be processed in time by related components.

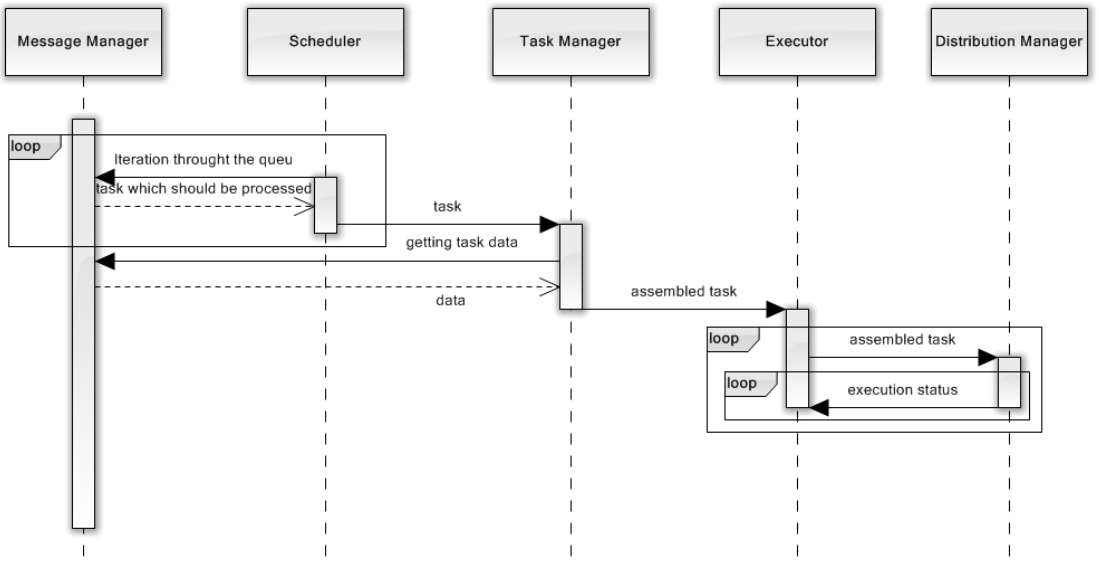


Figure 4.6: Middleware internal request processing (in UML Sequence diagram)

ecutor) can select and request any available item from the list of optional items (see Section 5.2.4 for example). Extra attached data are always referenced as an URI and this data will never be assembled to original message. The “body”-block of the message with items definition(-s) is required only. The attachments message is processed by the Data Manager. In fact, data items, which were provided within the initial response definition, are mandatory, and subsequently, attached data items are optional. The kind of data items (real data or links) does not matter.

The Data Manager correlates data items to identified request. All data items and attachments are stored in the file system. Separate data items are available through URIs with the following schema: *http://middlewarehost[:port]/data/{data identifier}*. Also, this address is used for optional data items. The data access interface allows read-only (HTTP GET) access. Therefore, there is no possibility to change, or delete data items. It is necessary to prevent possible data changes during request execution, which could cause side effects.

Using the HTTP GET method, a client can access the stored task definitions on the interface address: *http://middlewarehost[:port]/requests/{request identifier}*. With the HTTP DELETE method the stored request can be deleted. It should be noted that a request can be deleted only, if the request is not currently be-

ing processed (e.g. to prevent subscriptions to events, which never will occur). Another way is to make deleting of requests manual only: this can ensure data consistency (e.g. logging, event's mapping or during request processing).

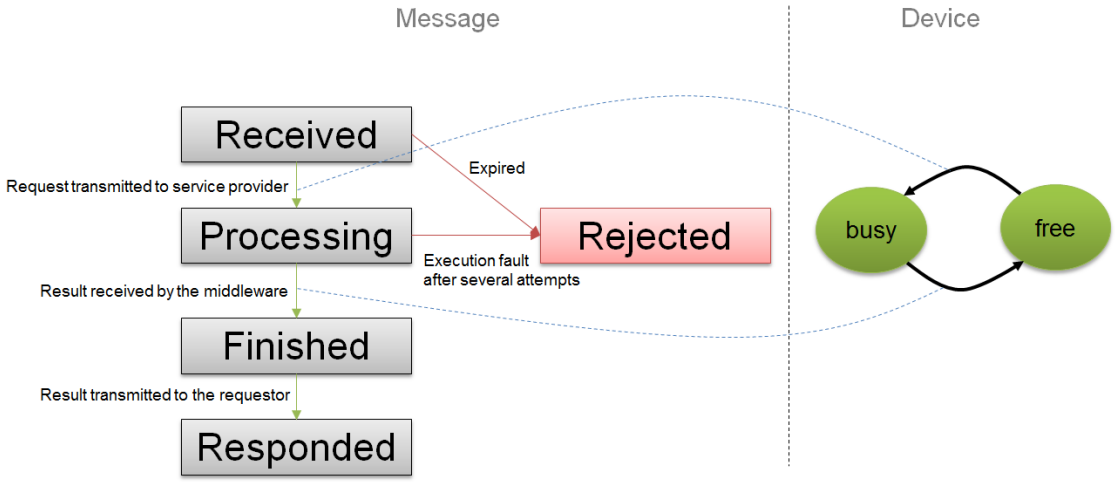


Figure 4.7: Request processing states

Figure 4.7 denotes the possible states of a request: Received, Processing, Finished, Responded, and Rejected. After the request is received by the middleware and stored in the message queue, the state of the message is *received*. If the Scheduler decides that the request is already expired, the state is *rejected*. This can happen if the Executor has begun with the processing of the request and waits until an appropriate service provider is available. In this time the request has expired, and the Scheduler sets the state of the request to *rejected*. In such case, the Executor will break the processing of this request.
When the Executor has begun the processing of the request, and the Distribution Manager has already found an appropriate executor, the state of the request is to *processing*. It can happen that an execution fails. In this case, the executor responds with an error, or the executor provides no feedback[9]. The Executor will try to reassign the request to a new service provider. The number of these attempts should be limited to avoid endless loops.
If the result of an execution is received by the middleware, the request is *finished*. The results are stored in a separate queue. The results' queue allows creation

[9]The maximum waiting time should be defined manually for concrete middleware usage scenario.

and maintenance of stored resources only: results are not processed by the middleware; only create, read, update, and delete operations are available. To distinguish between requests and results, the results management interface will be available by the following URI schema: *http://middlewarehost[:port]/results/{request identifier}*. A result can be created by a provider with the HTTP PUT method, and can be read by a client with the HTTP GET method. The requestor can continuously check the result arrivals through the REST interface. In persistent communication mode, the Executor will check, additionally, if a pointer to a persistent connection channel exists, and, if true, forwards result data to this channel. The state *responded* is optional and shows that the requestor has been notified about the arrival of results of the task execution (only in WebSocket communication mode). In any case, after arrival of the results (*finished* state) the requestor can access the results through the REST interface. By accessing the results through the REST interface, the state of the request will not be changed. In RESTful communication mode, all messages can be distinguished through the interface address and applied HTTP method. In persistent communication mode (using WebSocket protocol), however, this distinction is impossible. Due to the persistent connection is bound to a sole point, a message requires an additional message type parameter. For these purposes, the message schema (see Section 4.2) must be extended with a message type definition (see Listing 4.5). The message type can be defined with an additional attribute “type” within the message node:

```
<message type="create" ...
```

Listing 4.5: XML message type definition

The Communication Adapter will recognize the defined type (see Listing 4.5) and allocate the message within the Message Manager accordingly. Therefore, the “type” attribute is mandatory for the persistent communication mode. The type attribute can have the following values: create, get, update, delete, subscription, publication, check-in/-out, context-update, feedback, data, and recovery. In cases where an identifier is used, this identifier must be defined by the attribute “id”. Since this attribute is used within the outbound messages, it will not conflict with the id attribute defined previously (see Section 4.2.1).

The XML code fragment in Listing 4.6 is equivalent to the HTTP GET request *http://middlewarehost[:port]/data/123*.

```
<message type="data" id="123" ...
```

Listing 4.6: XML message type definition including an identifier

Until now, we have observed the request creation and results' delivery only (cf. Figure 4.5). So, in RESTful communication mode, the client uses the CRUD pattern on RESTful interface with the address schema *http://middlewarehost[:port]/-requests* for request handling, and *http://middlewarehost[:port]/results* for results. In persistent mode, the client requires a persistent connection to the middleware, and can create requests and receive results through this connection. Interaction steps in this case are identical to the RESTful mode (the middleware responses with an identifier of newly created requests). Additionally, in persistent communication mode, a recovery mechanism exists: if a connection was interrupted (a regular scenario for mobile communications (see Chapter 1)), the client should reconnect to the middleware, and this connection should be allocated to previously created requests. This can be done by connecting to the middleware and sending of a recovery message:

```
<message type="request-recovery" id="{request id}"/>
```

Listing 4.7: Recovery message

The recovery message is a simple message of type "request-recovery" with the identifier of the created request. Such recovery messages should be sent for every created request. A scenario where a request is created in RESTful communication mode, and a result is expected in persistent mode, is imaginable. So, a client does not create the polling loop, but rather awaits results through persistent connection (asynchronously). It allows efficient utilization of device resources (in terms of memory use and performance): no separate thread must be created. For this purpose, after the creation is done, the client opens the persistent connection to the middleware, and starts the connection recovery.
Now, we will observe required interactions on middleware-provider side: request delivery, processing feedback, and processing result.
Request delivery takes place when the middleware acts as a server - in order to

notify the client (in this case - provider) about request entries and transmit the request to the provider. In Web mode, a server push mechanism is required on this place. So, the request delivery in RESTful communication mode is realized with “Long polling” technique (see Section 4.3.1). Thus, the provider creates a (long) “poll” to *http://middlewarehost[:port]/providers/{provider identifier}*. The Communication Adapter will create a (temporary) communication channel within the Channel Manager for this poll. The request will be transmitted only when a channel exists for this poll: appropriate service requests and tasks will not be saved provider-specific, they will simply be forwarded to an active channel.
By persistent communication mode, the middleware pushes all tasks and service requests to the corresponding communication channels of a provider. The recovery message to reopen this connection is of type “recovery”. Therefore, the Channel Manager can correlate reopened connection to the specified (by the identifier) device.
Since a provider can host multiple services, and only one communication channel per provider is available, a required service name definition is necessary. This can be done in two ways: first, a provider will receive only the request identifier, and, then request it on the following access interface: *http://middlewarehost[:port]/results/{provider identifier}*. The provider reads the request scope and can find the service name within the context pattern. This implies that the receiver can understand the RDF/XML graphs and pattern schema. Second, the Distribution Manager will add a special data item to the task body. The data item is of type “service/name” and the name attribute represents the name of required service:

```
<item type="service/name" name="Sample gathering"/>
```

Listing 4.8: Service name definition as data item within body part

The second way is preferable, because it requires less interactions and simpler processing logic.
After the request is delivered, the middleware changes the internal state of the provider (provider device) to “busy” (cf. Figure 4.7). This device will not be considered by further distribution until a result (or a fault message) is responded. The state “free” means that the provider is accessible. Because mobile service

providers are stateful (see Chapter 1), an execution of any individual service will block the whole provider (device).

To ensure task is processing, the Executor requires a regular “ping” from the provider, which processes the task. The ping should be repeated in a predefined time intervals (timeout). In the middleware context, this ping is called “Feedback”, because a provider can send current processing state information: e.g. completeness in percent, or a human readable description. This state information will be appended to the request definition within the middleware request queue. The feedback is not required (optional), if the result arrives before the timeout expires.

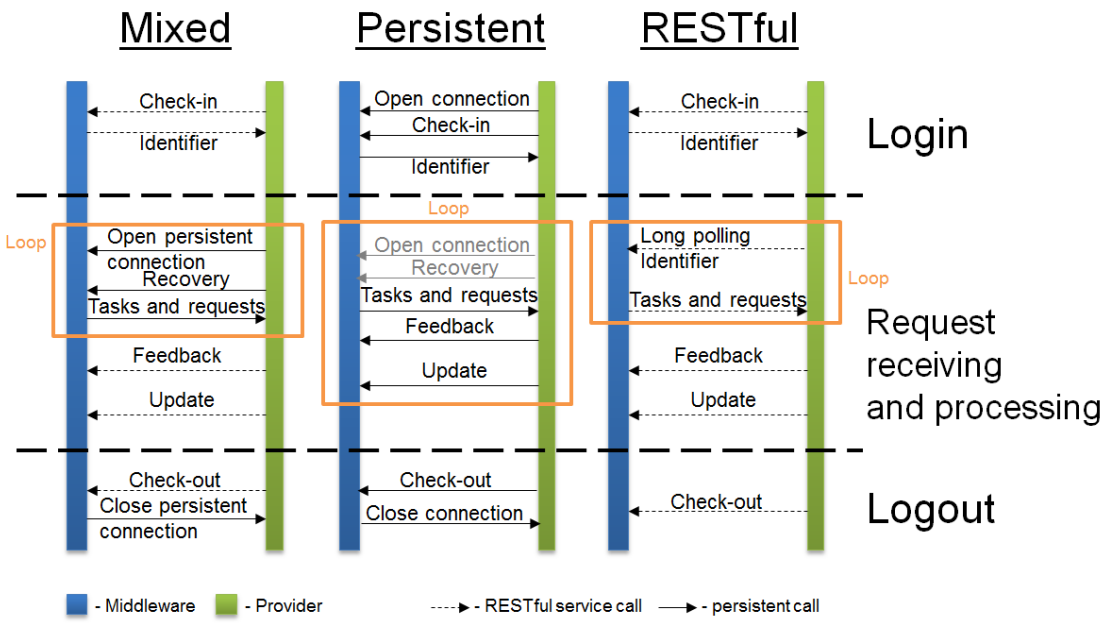


Figure 4.8: Differences between mixed, persistent, and RESTful modes

Figure 4.8 illustrates difference between mixed, persistent, and RESTful communication modes. Main differences are loops for request delivery from the middleware to a provider: while in RESTful mode this loop is mandatory, in mixed and persistent modes loops are optional and required if the connection was interrupted only. Also, the recovery mechanism is shown.

4.4.2 Dealing with context information

In the scope of this thesis, we differentiate between context *information* and context *pattern*. The context pattern means any definition (or description) of required (service) context: e.g. sending of task, context pattern defines required properties of a task executor as a query. The context information means the definition of properties (current state) of this task executor as a record. So, when the context pattern is a filter within a message, the context information requires own components on within the middleware (see Section 3.6.4) for handling the context information, and additional interactions for manipulating the context information. In this subsection, the term “context” means context information.
Context messages are processed in time and require no storage within the Message Manager. If processing of tasks and service requests are transmitted asynchronously, dealing with context data is more a synchronous interaction: it consists of several steps, and each step should wait until previous step is completed (step-by-step).

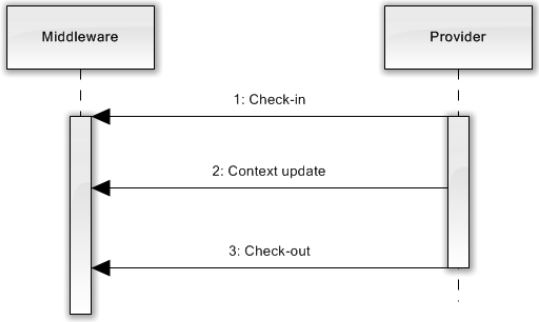


Figure 4.9: Dealing with context information (UML Sequence diagram)

As a first step, a provider sends a check-in message (cf. Figure 4.9). Within the check-in message, the provider specifies supported services and complete context information. This can be done through the REST interface (*http://middlewarehost[:port]/check-in*) or via the persistent connection (WebSocket protocol) (message type “check-in”, see Subsection 4.4.1). In any case, the message will be deserialized and handled by the Message Manager. The latter divides the message into two separated parts: the service description and context information. Firstly (1, cf. Figure 4.10), the Message Manager creates a new device entry within the Device Manager. Because the Device Manager is responsible

for device context data, a device identifier is required. Therefore, the Device Manager generates a new identifier for the created devices. The device identifier is provided as a result of the check-in operation. The provided identifier is used by the provider to receive incoming requests (tasks) (Section 4.4.1), and by the recovery mechanism in persistent communication mode (Section 4.4.1).

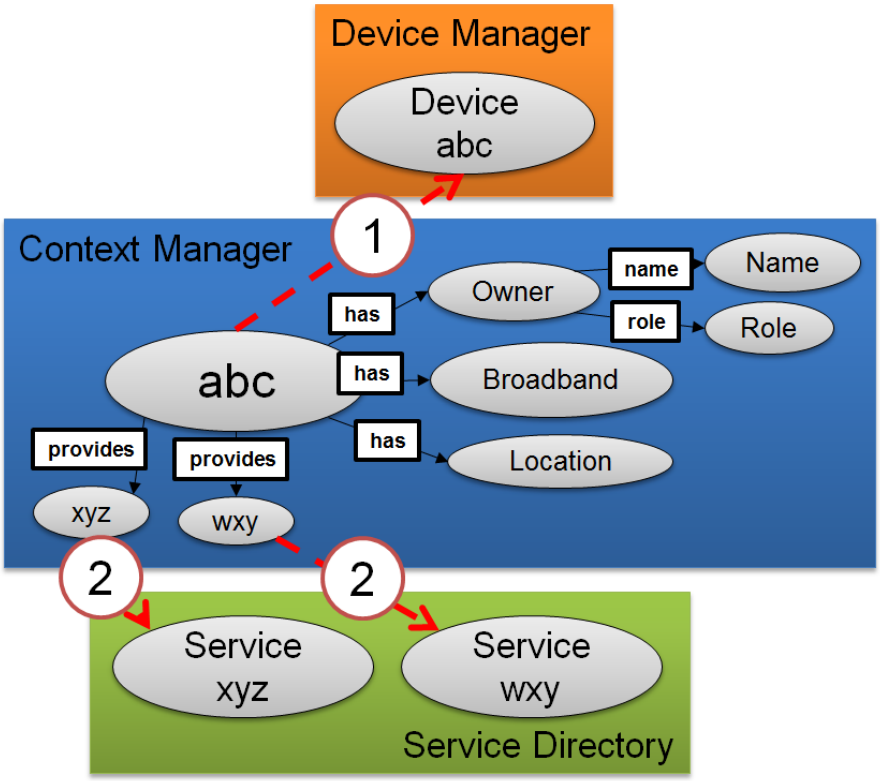


Figure 4.10: Relation between service description, context information, and device

As next (2, cf. Figure 4.10), the Message Manager looks for service descriptions in the Service Directory on the basis of provided service descriptions. If a service (service description) already exists, the Massage Manager takes the identifier of this service. In the other case, the Message Manager creates a new service entry, and assigns a new unique identifier. Finally, a RDF graph is constructed and stored within the Context Manager. All service identifiers are used as links to corresponding service descriptions (stored within the Service Directory), and the device identifier is used as the URI (rdf:about-attribute) for the device node in the graph (the root element for the graph).

Thereafter, the provider updates own (stored) context. This can be done through

a REST context update interface (*http://middlewarehost[:port]/context-update/-{provider identifier}*) for RESTful mode, or a message of type “context-update” (and the provider identifier as id attribute) for persistent communication mode. In both cases, the provider specifies, again, the complete context information within the context update message. The middleware replaces the existing provider context data with the newly one. The context update interface works only for “create” operations. However, a partial context management is possible, e.g. by providing CRUD pattern on the update interface (corresponding: context-replace, context-update, context-delete message types by persistent communication mode):

- Create (context-replace) – provides the replacement of complete context data, as currently implemented.
- Read – not required. The task executor (or the service provider) knows own context information and context changes. Thus, the “read” operation is not meaningful, and not required.
- Update (context-update) – allows addition of context data blocks. For example, an original context does not exist a broadband information entry. The provider generates this entry and sends it to the middleware. The Context Manager will append newly provided entries to the provider context information.
- Delete (context-delete) – deletes a concrete context data block (entry), provided by the provider (or executor) as a pattern.

Context handling possibilities are dependent on the middleware usage scenario: if the middleware is used for scenarios, where all task executors are inside one building, a basic context handling is sufficient. In scenarios, where task executors, or service providers, are continuously moving, an advanced context handling is necessary. Therefore, the service provider can minimize data transfer by sending only changed context data parts, and not replacing the whole context information, as in basic context handling. For example, the provided clinic scenario, basic context handling is sufficient (see Chapter 5).
The last step (cf. Figure 4.9) is when the provider removes its whole data from the middleware (check-out operation, not to confuse with “context-delete”). Af-

ter this operation the provider will not be longer considered by the distribution of tasks (or service requests). This will be done through a REST check-out interface (*http://middlewarehost[:port]/check-out/{provider identifier}*) for RESTful mode, or a message of type “check-out” (and the provider identifier as id attribute) for persistent communication mode. The Message Manager will delete all occurrences of the provider (the device) within the Device Manager and the Context Manager. Therefore, further communication between the middleware and the provider is not possible.
In cases where the check-out operation was not executed, the amount of not properly unregistered service providers can affect the middleware negatively: the Device Manager and the Context Manager will be “wasted” with not existing providers, and the distribution will be slowed down. A manual (or automated) clearing of “cached” context information is required. The clearing means that all devices (and accordingly context information) will be removed, if they have no active communication channels in the middleware. Also, the middleware can store additional parameters, which shows the last access of the middleware by the provider. Therefore, the cleaning mechanism will remove all outdated providers.

4.4.3 Handling events

Event handling provides event support within distributed systems. The used event mechanism is based on the publish-subscribe pattern [14, 13]. It consists of two operations: publish and subscribe (cf. Figure 4.11).
By publishing (Figure 4.11), the task executor (or the service provider) stores some event data within the middleware: e.g. processing state, availability of service providers. This data is stored within the Publications component queue (see Section 3.6.5). The events are correlated to the middleware working items: requests (tasks) and providers (executors). Therefore, an event can describe only properties of these two items. The definition of an event consists of the following parts: *item type*, *item identifier*, *event data* or *event description*, and *time point of publication*. The time point of publication will be defined by the middleware, the rest must be provided by the publisher of the event. Items are defined as data items within the body part of the message. The item type can

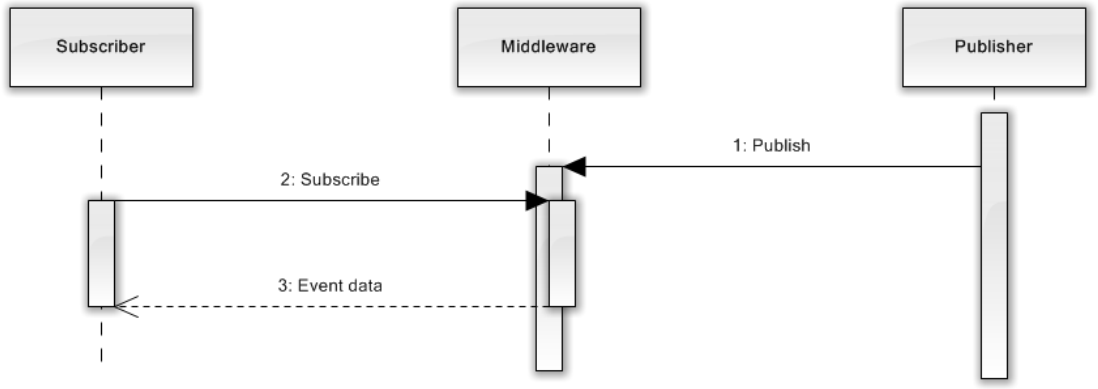


Figure 4.11: Handling events (UML Sequence diagram)

be of type **requestor** or **provider** (data item of type “event/type” and the name attribute “base”). The item identifier is an empty data item of type “base/identifier” and the name attribute provides the identifier of a request, or of a service provider. The event data or the event description are common data items with user defined types, names and encoding.

A subscriber sends the subscription for events to the middleware. In the scope part of the subscription message, the subscriber defines a filter for events-of-interest (EOI). There exist two filters: basic and extended. The basic filter allows only concrete definitions of event parts: event item type (request or provider), event item identifier, and the start and the end of a time range to be considered. These filter properties are read by the Event Manager, which selects all events which match the defined filter criteria from the publications queue. All selected events are delivered to the subscriber as a result. The extended filter allows, additionally, definition of RDF patterns: e.g. all requests to the “Sample gathering” service, or all service providers, which are owned by a doctor. In such cases, the Event Manager will go through the requests’ queue and through the Context Manager queue. Thereafter, the Event Manager selects all “Sample gathering” service providers and corresponding events, and, finally, transmits these to the subscriber.

This event handling mechanism is limited to the request-response method and does not create a relation between a subscriber and a subscription. In order to receive events, the subscriber requires a subscription loop, which continuously “polls” (checks) for new events. In RESTful communication mode, this is realized

by continuous (long) polling. For multiple simultaneous subscriptions, the subscriber requires one polling loop per subscription. In persistent communication mode, a subscription exists as long as the connection is open. But, in this case, all simultaneous subscriptions are handled over this connection: separate connection for each subscription is not required. If the connection was interrupted – all subscriptions get lost. There is no connection recovery mechanism available. This event model is used, also, for logging. By using the filtering a controlling system can evaluate the log. A manual clearing of the log should be implemented within administration features of the middleware.
The following Table 4.2 provides an overview of required and optional message parts for a concrete middleware operation:

Operation	Schedule	Priority	Scope	Multi	Body	Address	Message type
Request creation	o	o	r	o	r	/requests (PUT)	create
Attaching data	-	-	-	-	r	/requests/id (POST)	update
Storing data	-	-	-	-	r	/data (PUT)	data-create
Reading data	-	-	-	-	d	/data/id (GET)	data
Reading request	-	-	-	-	d	/requests/id (GET)	get
Request deleting	-	-	-	-	-	/requests/id (DELETE)	delete
Request delivery	-	-	-	-	r	/providers/id (GET)	(recovery)
Processing feedback	-	-	-	-	o	/feedback/id (POST)	feedback
Processing feedback	-	-	-	-	r	/results/id (PUT)	result
Result delivery	-	-	-	-	r	/results/id (GET)	(request-recovery)
Check-in	-	-	r	-	-	/check-in (POST)	check-in
Context update (basic)	-	-	r	-	-	/context-update/id (POST)	context-update
Context creation (adv.)	-	-	r	-	-	/context-update (PUT)	context-create
Context update (adv.)	-	-	r	-	-	/context-update/id (POST)	context-update
Context delete (adv.)	-	-	r	-	-	/context-update/id (DELETE)	context-delete
Check-out	-	-	-	-	-	/check-out/id (GET)	check-out
Publish	-	-	-	-	r	/publish (PUT)	publication
Check-out	-	-	o	o	-	/subscribe (POST)	subscription

Table 4.2: Operation specifications overview

5 Middleware usage demonstration

In Chapters 3 and 4 theoretical consideration of the researched middleware were provided. This chapter discusses concrete interactions within a (complex) global process from registering of services, through task definition, service request, distribution by the middleware, results' response, and background service context handling. The sample (real-world) scenario is a clinical process that is supported through mobile devices for mobile task execution. The middleware provides tasks coordination and distribution. In following, all required steps and interactions to implement the scenario will be shown.

5.1 Scenario description

The treatment process begins when the patient arrives at the clinic reception and complains about recurring headache (①) (cf. Figure 5.1). A receptionist calls a general practitioner (②) (mobile service named "Analysis"). To make the analysis, the general practitioner requires a free examination room (③), which can be found via a room booking service (wired Web service with the name "RoomBookingService"). So, the practitioner makes a service call of the "RoomBookingService" and reserves a free examination room. The general practitioner sends the reserved room number to the reception (④). The receptionist brings the patient to the room and the analysis starts. The general practitioner creates a case history and starts an anamnesis (⑤). Thereafter, the general practitioner assumes the following diagnosis: head injury, high blood pressure, or a side effect of tablets medication. As next, the general practitioner calls a nurse to delegate necessary sample tests (⑥). Since this action is supported by the middleware, the practitioner can create a task for a nurse. The task coordination and distribution are done by the middleware. To make a sample analysis, the

nurse requires concrete definition of actions: blood substance analysis, blood pressure measuring, basic X-Ray analysis. The general practitioner defines all these parameters and, additionally, defines the patient's case history identifier, as well as the current location. Finally, he sends the task definition via his mobile device to the middleware.

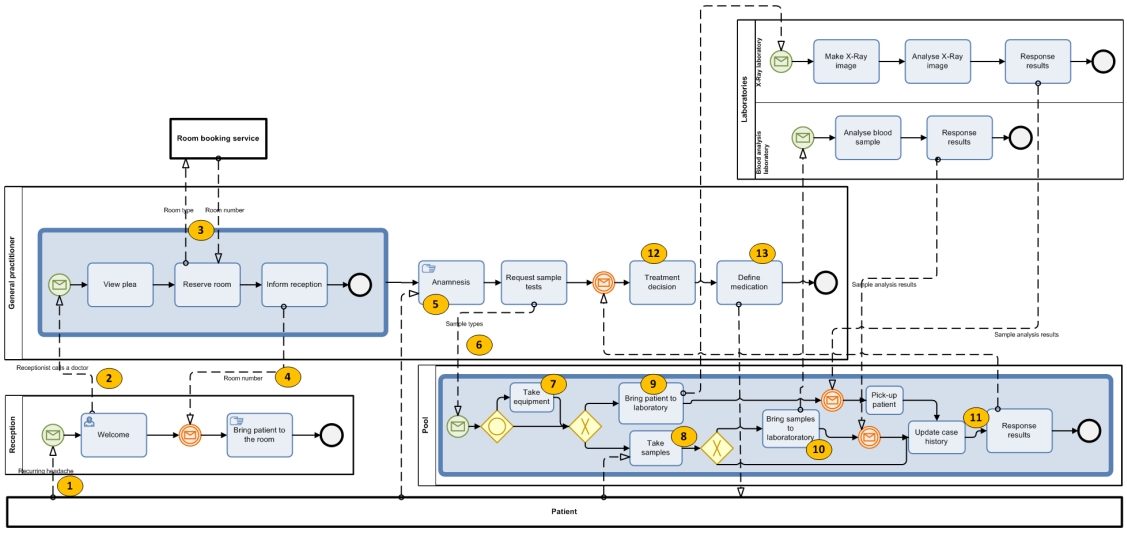


Figure 5.1: Demo scenario process model (in BPMN 2.0) (see B.4 for larger view)

After receiving the task definition, the nurse takes necessary equipment (if possible/required) (⑦) and goes to the mentioned room. Then, the nurse takes a blood sample and measures the patient's blood pressure (⑧). The blood sample analysis and the X-Ray analysis cannot be done in the examination room. Therefore, the nurse leads the patient to the X-Ray laboratory (⑨), and leaves the patient to bring the gathered blood sample to the laboratory (⑩). In the meantime, an X-Ray image of the patient's head will be made. After the X-Ray image was done, the nurse pickups the patient from the X-Ray laboratory, and returns to the examination room.

The X-Ray laboratory and the blood analysis laboratory services are so called "blank services". They require only material input (e.g. patient itself, blood sample) and no electronic parameters. Therefore, the creation of such service calls can be automated by scanning special QR codes or NFC chips (which can, for example, could be placed near the laboratory entrance). The provided request identifier can be used to mark the input materials. So, after analyzing the blood

sample, the laboratory worker will input sample marker data (e.g. a number) and the result of the analysis. The corresponding application will translate these data to a service response.
After receiving results from the laboratory, the nurse updates the patient's case history with the newly received data and informs the general practitioner about it by sending a response of nurse's sample gathering service (11).
After notification, the general practitioner can continue further treatment decisions (12). Accordingly, the general practitioner will access these results in the updated case history, or directly from the response (results are stored within the middleware in this case). The general practitioner sees normal analysis values and decides that the cause of the headache is a migraine, and prescribes painkillers (13).

5.2 Implementation

This section elaborated how the described demo scenario can be implemented with the application of the researched middleware. Required interactions and concrete operations, as well as used messages will be explained.

5.2.1 Prerequisites

We assume that the middleware is running on a host named *mw.modernclinic.de*. All corresponding apps are already installed on employee's mobile devices, and each participant has a mobile device.
To cover as much as possible different communication scenarios, we assume that the software in laboratories use wired connections due to their stationary position. Additionally, we assume that the connection in the X-Ray laboratory is regularly be interrupted by produced rays. This requires periodic recovery of the connection. The connection in the reception is also wired, and the reception software acts as a requestor (client) only.

5.2.2 Establishing of connections

At the beginning of the shift, the general practitioner starts the mobile application on his device. By starting, the application connects to the middleware: first, the application must checking-in, second, it must create a connection in order to receive incoming messages (service requests or tasks' definitions).
Therefore, the application on the device sends the following message to *http://-mw.modernclinic.de/check-in*:

```
<message>
 <scope>
  <rdf:Device>
   <device:has>
    <location:department>General Medicine</location:
       department>
   </device:has>
   <device:provides>
    <rdf:Service>
     <service:name>PatientAnalysis</service:name>
    </rdf:Service>
   </device:provides>
   <device:has>
    <rdf:Owner>
     <owner:name>Who</owner:name>
     <owner:role>Doctor</owner:role>
    </rdf:Owner>
   </device:has>
   <device:has>
    <device:broadband>WLAN</device:broadband>
   </device:has>
  </rdf:Device>
 </scope>
</message>
```

Listing 5.1: Check-in message

As a result, the middleware answers with an identifier (here 67). The identifier will uniquely identify the doctor's device.
Second, the application creates a loop where it continuously sends requests (HTTP GET method) to *http://mw.modernclinic.de/providers/67*. If the application receives an empty answer, it makes the next request, until a service request or a task description is received (long polling).
At the same time, the nurse arrives at work and switches on her mobile device, and starts a corresponding application. So, the application also sends a check-in request to the middleware. As a result of the check-in operation, the nurse application, also, receives an (provider) identifier assigned (32). The nurse makes an internship on the clinic. She finds out later that she needs to work in the general medicine department (before no concrete department was defined in the device context information). So, she should update her context information. For this purpose, she selects the new department in the application. The nurse application sends then the following message to *http://mw.modernclinic.de/context-update/32*:

```
<message>
 <scope>
  <rdf:Device >
   <device:has>
    <location:department>General Medicine</location:
       department>
   </device:has>
   <device:provides>
    <rdf:Service>
     <service:name>SampleGathering</service:name>
    </rdf:Service>
    <rdf:Service>
     <service:name>Medication</service:name>
    </rdf:Service>
   </device:provides>
   <device:has>
    <rdf:Owner>
```

```
    <owner:name>Amy Pond</owner:name>
    <owner:role>Nurse</owner:role>
   </rdf:Owner>
  </device:has>
  <device:has>
   <device:broadband>WLAN</device:broadband>
  </device:has>
 </rdf:Device>
</scope>
</message>
```

Listing 5.2: Context update message

Also, laboratory workers send check-in messages by starting the laboratory software. Because this software uses wired connection, the check-in messages look like following (for blood analysis laboratory):

```
<message type="check-in">
 <scope>
  <rdf:Device >
   <device:provides>
    <rdf:Service>
     <service:name>BloodSubstanceAnalysis</service:name>
    </rdf:Service>
    <rdf:Service>
     <service:name>Medication</service:name>
    </rdf:Service>
   </device:provides>
   <device:has>
    <rdf:Owner>
     <owner:name>Blood analysis</owner:name>
     <owner:role>Laboratory</owner:role>
    </rdf:Owner>
   </device:has>
   <device:has>
```

```
    <device:broadband>LAN</device:broadband>
   </device:has>
  </rdf:Device>
 </scope>
</message>
```

Listing 5.3: Check-in message for persistent connections

To establish a permanent connection in X-Ray laboratory, which can regularly be interrupted, the software in the X-ray laboratory needs to recover interrupted connections. The software connects to *http://mw.moderncliniс.de* (WebSocket Protocol) and sends the recovery message (see Listing 5.4) to assign newly created (opened) connection to existing provider registration (51 is the provider identifier). The middleware will map this connection to the existing device record. All messages will be transmitted through this connection.

```
<message type="recovery" id="51"/>
```

Listing 5.4: Recovery message

Now, all connections are established and processing of service requests can be coordinated by the middleware.

5.2.3 Processing service requests

When a patient comes to the clinic, the receptionist needs to call a doctor. Therefore, the receptionist selects an appropriate department based on first impressions and medical conditions. The reception software connects to *http://mw.moderncliniс.de* (over WebSocket Protocol) and sends the service request message to the middleware:

```
<message type="request" priority="10">
 <scope multi="1">
  <rdf:Service>
```

```
  <service:name>PatientAnalysis</service:name>
 </rdf:Service>
 <rdf:Owner>
  <owner:role>Doctor</owner:role>
 </rdf:Owner>
 <rdf:Location>
  <location:department>General Medicine</location:
     department>
 </rdf:Location>
</scope>
<body>
 <item name="condition description" type="condition"
    encoding="text">
  <![CDATA[Patient complains standing headache]]>
 </item>
 <item name="sender" type="sender" encoding="tex">
  <![CDATA[Reception]]>
 </item>
</body>
</message>
```

Listing 5.5: Service request message for the persistent connection

The semantic of this message is as follows: the request should be delivered as soon as possible with priority 10; a provider that provides the “PatientAnalysis” service is required, the service provider must be a “Doctor”, and belongs to the “General Medicine” department. The response of this request is the request identifier (95). The software uses it to recover the connection:

```
<message type="request-recovery" id="95"/>
```

Listing 5.6: Service request message by provider

The doctor receives the service request from the middleware as follows:

```
<message id="95">
 <body>
```

```
  <item name="condition description" type="condition"
     encoding="text">
   <![CDATA[Patient complains standing headache]]>
  </item>
  <item name="sender" type="sender" encoding="text">
   <![CDATA[Reception]]>
  </item>
 </body>
</message>
```

Listing 5.7: Recovery message for the persistent connection

After the application on the doctor's device has received the message (see Listing 5.7), the task will be presented by a prompt dialog to the doctor: "Patient complains standing headache. Please come to the Reception." The doctor decides to accomplish the anamnesis of the patient in an examination room. The clinic computing center provides a special Web service called "RoomBookingService" in order to reserve a room. The service requires a room type as an input parameter (string). The doctor knows about this service and selects the required room type:

```
<message>
 <scope>
  <rdf:Service>
   <service:name>PatientAnalysis</service:name>
  </rdf:Service>
 </scope>
 <body>
  <item name="roomtype" type="string" encoding="text">
   <![CDATA[Examination room]]>
  </item>
 </body>
</message>
```

Listing 5.8: Web service request message

The middleware replies with a message containing the request identifier “10”. To receive the result from room booking service, the application on the device starts a loop to *http://mw.moderncliniс.de/results/10*. Later, the doctor receives an answer with the room number “ER-456”. Finally, the doctor app generates an answer to the reception (reception software still waits for a response) by sending a response message to *http://mw.moderncliniс.de/results/95*:

```
<message id="95">
 <body>
  <item name="action" type="action/description" encoding="
     text">
   <![CDATA[Bring patient to the room ER-456.]]>
  </item>
 </body>
</message>
```

Listing 5.9: Response message

The rest of communication processes is handled in the same way.

5.2.4 Handling big data

During the description of the middleware in this thesis, we have used two types of data items: required data and optional data. The broadband of the connection between a mobile device and the middleware can be limited. Therefore, it is necessary to minimize message overhead. This can be achieved through optional attachments: the requestor defines mandatory and optional data items. With the HTTP POST method the client can attach additional (optional) data items to the task definition (see Section 4.4.1). For example, after an X-ray examination, a nurse reports with a textual description and an X-Ray image. The textual description is defined as mandatory and the X-Ray image as optional. A doctor will receive, firstly, only the text. In case where the textual description is not sufficient, the doctor can request the attached X-Ray image. Here, the X-Ray image can have a very large size, and transmitting of this image will block the communication for a long time.

Additionally, any provider can store any data objects on the middleware. Storing data items is possible through the HTTP PUT method to http://middlewarehost-[:port]/data/. In principle, attaching of extra items to a response is the same operation, except that by the attaching stored data will be correlated to the response (as optional items). The limitation of the middleware's data storage is that it is impossible to store links (URIs): a provider stores a data item and receives a link to this item. Additional storage of this link within the middleware as a separate item is not possible. The attachment of links to a response is allowed.

In this example, the laboratory application sends the textual description in the initial response, and the X-Ray image will be attached as optional. Because there is no mechanism available to observe response changes, the client should re-read the response to become up-to-date (or actual response state).

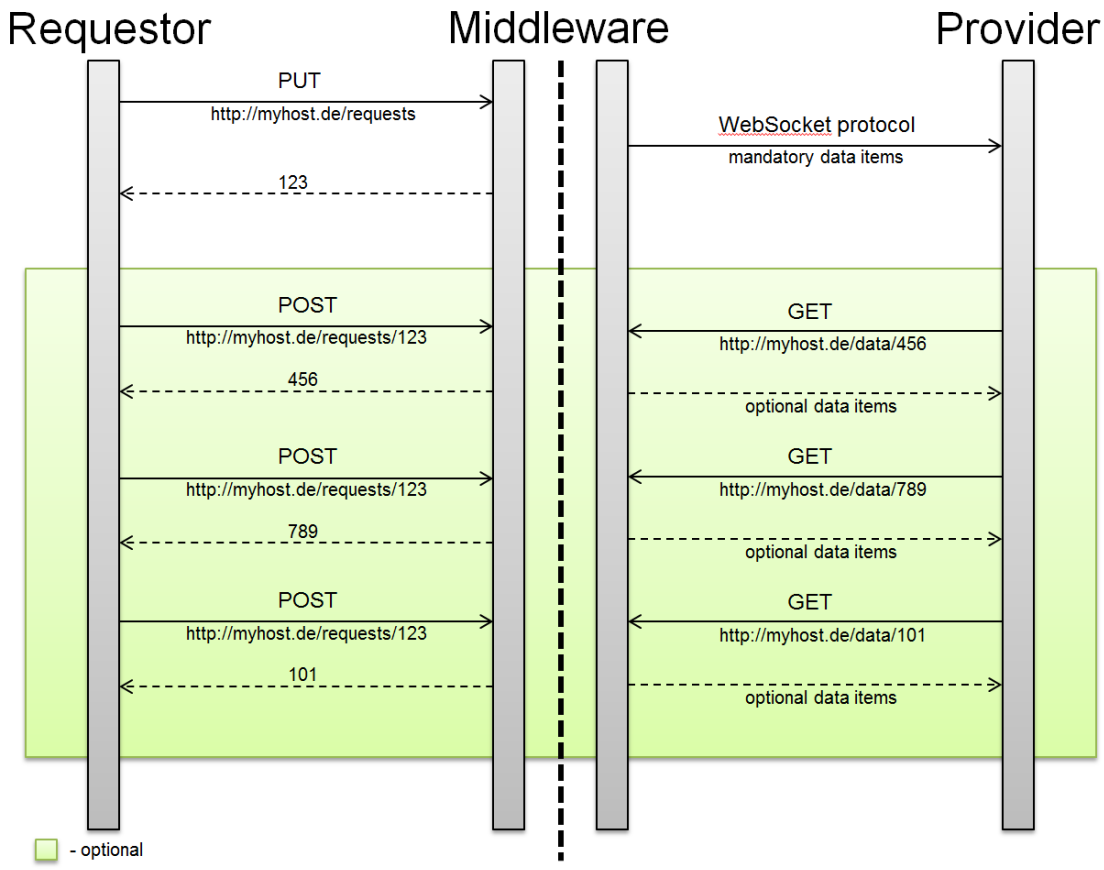


Figure 5.2: Pushing/pulling of optional data

5.2.5 Unregistering service providers and closing connections

At the end of the shift, the doctor, the nurse and laboratories close their applications. This requires a valid unregistering from the middleware.
First of all, applications should save all identifiers of opened requests on both sides (client side and provider side). These identifiers will be used for sending or receiving of responses. Because the communication with the middleware is asynchronous, interruption of the request processing is allowed, and will not result in losing requests.
When all identifiers are saved, applications should check-out from the middleware. This can be done by sending of empty check-out messages to the middleware: for example, the HTTP GET request on *http://mw.modernclinic.de/check-out/67* by the doctor, or following message over the WebProtocol connection for the X-Ray laboratory:

```
<message type="check-out" id="51"/>
```

Listing 5.10: Checkout message for the persistent connection

6 Summary and outlook

The purpose of this thesis is to find a way to integrate mobile devices into business information systems and enterprise computer environments. Realizing this kind of new classes of software solutions and process-aware applications presents a challenging task. Especially, the coordination and communication within the environment, which consists of stationary and mobile entities, is not trivial. Furthermore, the thesis is concentrated on the analysis of appropriate technologies, concepts, design, and architecture decisions for implementing mobile services.

The main goal of this thesis is the conceptualization and design of a middleware, which allows the coordination and the integration of separated (mobile) service providers in a distributed environment. In particular, it observes conceptual and technical requirements for available technologies, communication protocols and implementation techniques. Additionally, it derives an architecture to provide an interoperable solution, which can deal with stateful, context-aware mobile services and their impermanent connections.

The provided result is a middleware, which provides the integration of mobile as well as stationary service providers, and supports the implementation of mobile, Web, and desktop client applications. The focus lies on the communication between client and service provider components, context-based request distribution and, as a result, mobile task coordination.

The introduction of Apple's iPhone brought programmable mobile devices (smartphones) to private users in 2007. Though, mobile devices like PDAs and smartphones were available earlier (e.g. Microsoft Windows Mobile since 2002). However, these devices were mostly used by enterprises and in business area. Nowadays, 34% of all portable communication devices (mobile phones) in Ger-

many are smartphones[1].

Also, the popularity of tablet computers increases constantly. The usage of tablets in the clinical domain is preferable: the display size of a smartphone is too small to edit documents (e.g. case histories), and PCs does not provide the required mobility. In the future, notebooks and netbooks could be replaced by tablets for tasks like browsing and viewing of information, editing small texts, and checking emails. The usage of computing devices could be grouped as follows: servers, desktop computers (including notebooks) for office and development use, tablets for the rest. It is also imaginable that tablets become equipped with even more processing power and could replace desktop computers and notebooks by connecting external periphery devices (e.g. keyboard, mouse and monitors). If we compare modern operating systems as shown in Figure 6.1, the described trends can be observed as follows: Apple and Google follow the idea to support "intelligent" alternatives to normal computers, while Microsoft "mobilizes" them.

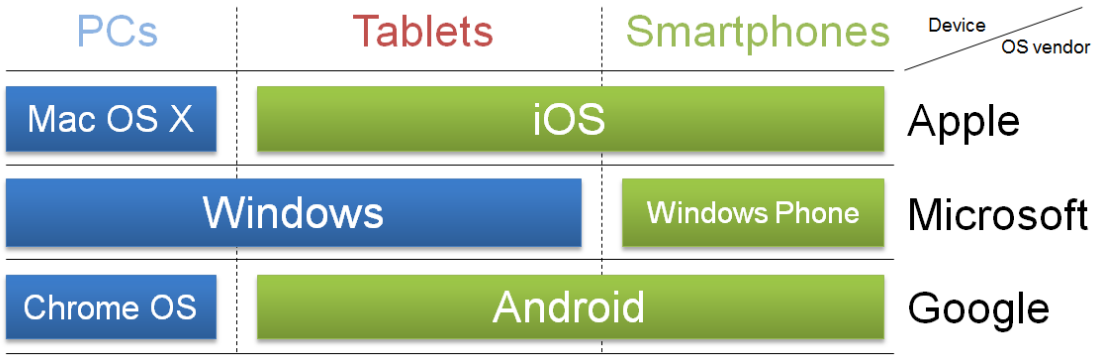


Figure 6.1: Operating system coverage of computing devices

The researched middleware within this thesis provides possibilities to use mobile computing devices like smartphones, PDAs, and tablets to act in an environment combined with stationary computers. Three ways of combining these two worlds (mobile and stationary computers) are imaginable: first way is building of mobile oriented servers ("fat" processing tier – "thin" mobile client). Second way is developing of "intelligent" mobile applications, which can interact with stationary

[1] http://www.gfm-nachrichten.de/news/aktuelles/article/smartphones-in-deutschland-mit-34-prozent-marktanteil.html, viewed 14.10.2012

computers directly (“thin” processing tier – “fat” mobile client). Third way, which is discussed in this thesis, is to provide a bridge between mobile and stationary devices, which acts as a translator and does not require further research and developing of these.

Furthermore, integration of mobile computing devices into computer environments opens new ways for programming “next generation” software solutions and, accordingly, for new business models and processes. Moreover, new models and business processes can be integrated in many areas: logistics, technical support, field services, medicine (e.g. as shows the discussed example) or, even, internal enterprise communication or product testing.

The developed framework of this thesis is concentrated on mobile task coordination. Multiple research approaches exist in this area. In each of them the term “task” is defined differently. In [25] and [20] a task can be defined as part or a fragment of a business process, which is transmitted to a distributed executor. The executor requires in this case a (mobile) process engine to perform the received task. A central process management system, which defines, configures, fragments, deploys and synchronizes the process parts [20], is responsible for the choreography and the orchestration. Such approaches are appropriate in cases where strictly pre-defined processes (process parts) are required and changing of workflow is not allowed: e.g. on patient’s devices. [23] defines an activities as mobile agents: a task is, in this case, a block of code (or stand-alone program). The mobile agent requires an execution environment on task processor side, which performs the execution during runtime. The orchestration is hard-coded within the agent’s processing logic, and the choreography is performed on central server. The task executor has no impact to the entire process (process part) description. Futhermore, performing changes to parent processes is challenging and requires re-implementing of program logic. This approach can be used in cases where durable process definitions are expected: common tasks and data management. However, all presented approaches are based on static processes and require a central control entity, which handles distribution and assignment of several tasks (activities within the process). The figured approach in this thesis allows an equal behavior, where a process management system controls process instances and uses the researched middleware for tasks’ delivery

only, as well as a stand-alone solution for task coordination. The second usage scenario is suitable for role-based task assignment in self-organizing (dynamic) structures. The orchestration is based on the experience and personal decision of the task executor, and the choreography is provided through administrative, management or law regulations. This approach s suitable for to scenarios, where a high level of human-based tasks exists and personal experience-based decisions result in constantly process deviations (e.g. emergencies or incident management).

Naturally, one research thesis cannot cover all aspects and provide an out-of-the-box solution. Supposable further research and development should expose components like automatic context-based formatters and converters; manageable distribution and allocation policies; security and rights management.

The architecture of the researched middleware consists of several components, which provide support for basic scenarios only. Therefore, the architecture can be called “component-based”. By extending or replacing of components and subcomponents, any required behavior of the middleware can be achieved. Thus, components, which are responsible for communication, could be extended to more communication protocols and techniques (e.g. broadcasting possibilities from a mobile device). Additionally, they can allow the usage of the middleware as an edge-server-system (e.g. to support uniform task coordination and distribution within several locations). The distribution subcomponent could be extended with more intelligence to automate activities: e.g. content-based distribution, which can replace anamnesis and can select appropriate department by analyzing provided plea data (as message content). Besides, the distribution can be based on special rules (“distribution policies”), which predefine additional properties of a receiver, or help to make better decisions for distribution: e.g. load balancing of service providers or distribution by level of experience (distinguishing between common and special cases). The data handling within the middleware can be more efficient by implementing a caching mechanism, which recognizes same data items and combines them. The extension of the middleware with optional message formatters and data converters increases the effectiveness of data transmission (e.g. image processing dependent on receiver’s screen resolution and connection bandwidth), and allows integration of variety

of service implementations (e.g. message transformation to a remote procedure call, or transformation to a SOAP, XML-RPC message).
In any case, the thesis provides the fundament for integration of mobile devices into information systems and business processes. The flexibility of the defined message structure allows the integration of the middleware in many purposes, and implementation of mobile, as well as stationary entities. The researched middleware architecture ensures main aspects of this integration: communication between distributed entities and coordination of activity requests. Hence, discussed communication protocols and interaction patterns are useful for further projects and findings of “next generation” information systems.

A Message XML Schema Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
         ns#">

<import namespace="http://www.w3.org/1999/02/22-rdf-syntax
   -ns#"
  schemaLocation="http://www.w3.org/1999/02/22-rdf-syntax-
     ns#" />
<element name="message">
 <complexType>
  <sequence>
   <element name="scope" minOccurs="0" maxOccurs="1">
    <complexType>
     <complexContent>
      <extension base="rdf:RDF">
       <attribute name="multi" type="positiveInteger" use="
          optional" default="1" />
      </extension>
     </complexContent>
    </complexType>
   </element>
   <element name="body" minOccurs="0" maxOccurs="1">
    <complexType>
     <sequence>
      <element name="item" minOccurs="1" maxOccurs="
         unbounded">
```

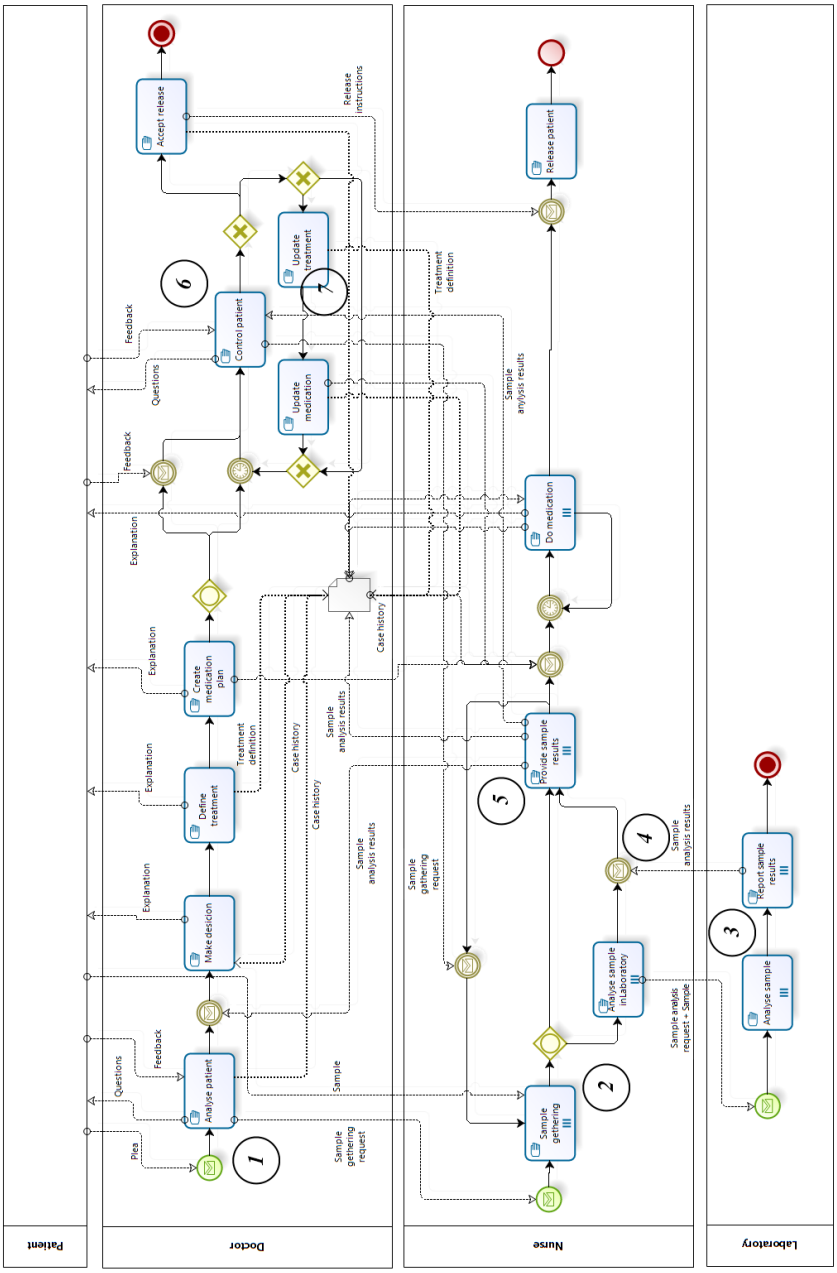
```
    <complexType>
     <attribute name="name" type="string" use="required"
        />
     <attribute name="type" type="string" use="required"
        />
     <attribute name="encoding" use="required">
      <simpleType>
       <restriction base="string">
        <enumeration value="text" />
        <enumeration value="xml" />
        <enumeration value="base64" />
       </restriction>
      </simpleType>
     </attribute>
    </complexType>
   </element>
  </sequence>
 </complexType>
</element>
</sequence>
<attribute name="schedule" type="string" use="optional">
<attribute name="priority" type="integer" use="optional"
    default="0" />
<attribute name="type" use="optional">
 <simpleType>
  <restriction base="string">
   <enumeration value="create" />
   <enumeration value="update" />
   <enumeration value="data-create" />
   <enumeration value="data" />
   <enumeration value="get" />
   <enumeration value="delete" />
   <enumeration value="recovery" />
```

```
      <enumeration value="feedback" />
      <enumeration value="result" />
      <enumeration value="request-recovery" />
      <enumeration value="check-in" />
      <enumeration value="context-update" />
      <enumeration value="context-create" />
      <enumeration value="context-delete" />
      <enumeration value="check-out" />
      <enumeration value="publication" />
      <enumeration value="subscription" />
     </restriction>
    </simpleType>
   </attribute>
   <attribute name="id" type="string" use="optional" />
  </complexType>
 </element>
</schema>
```

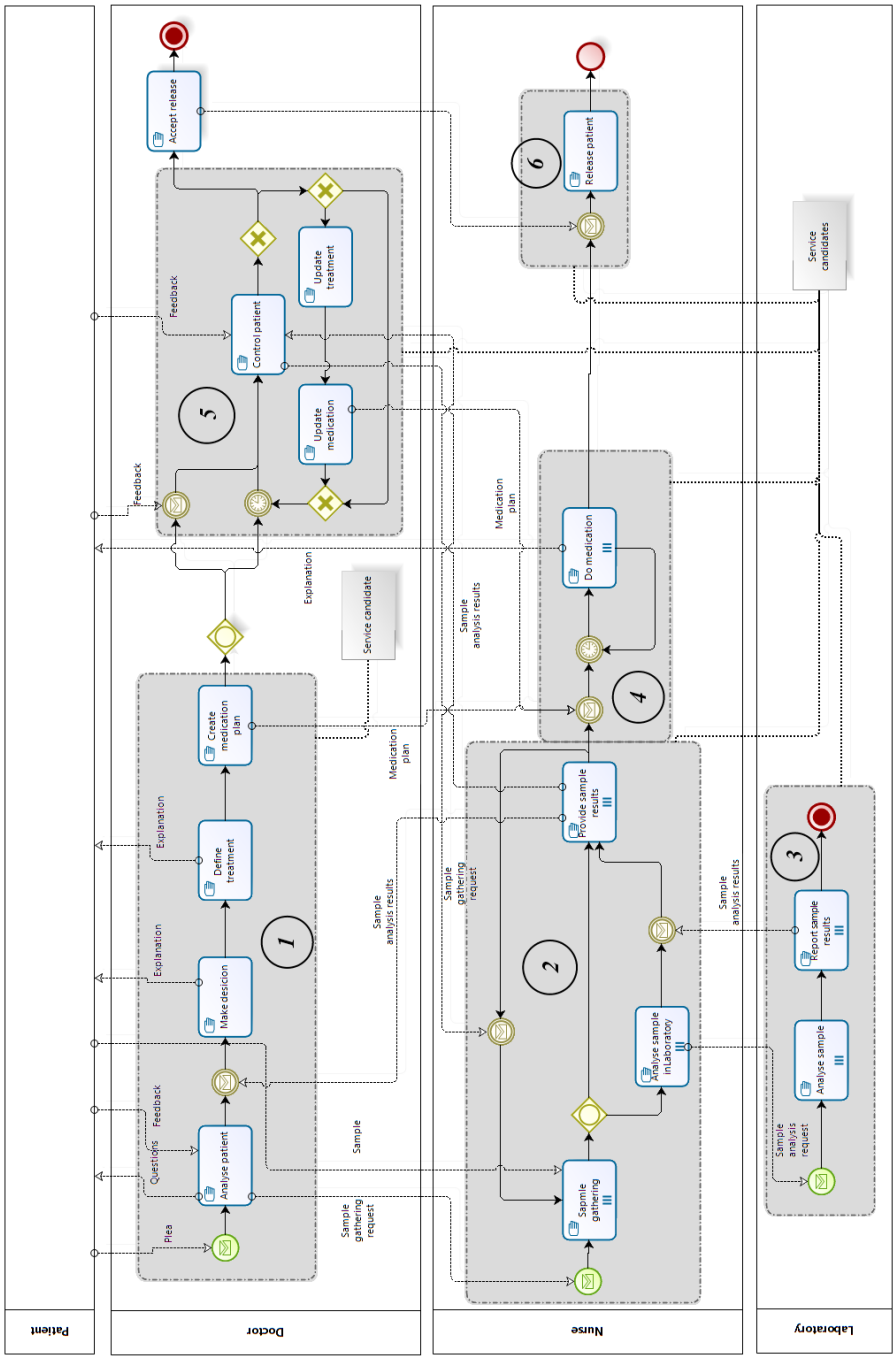
Listing A.1: Message XSD

B Process models

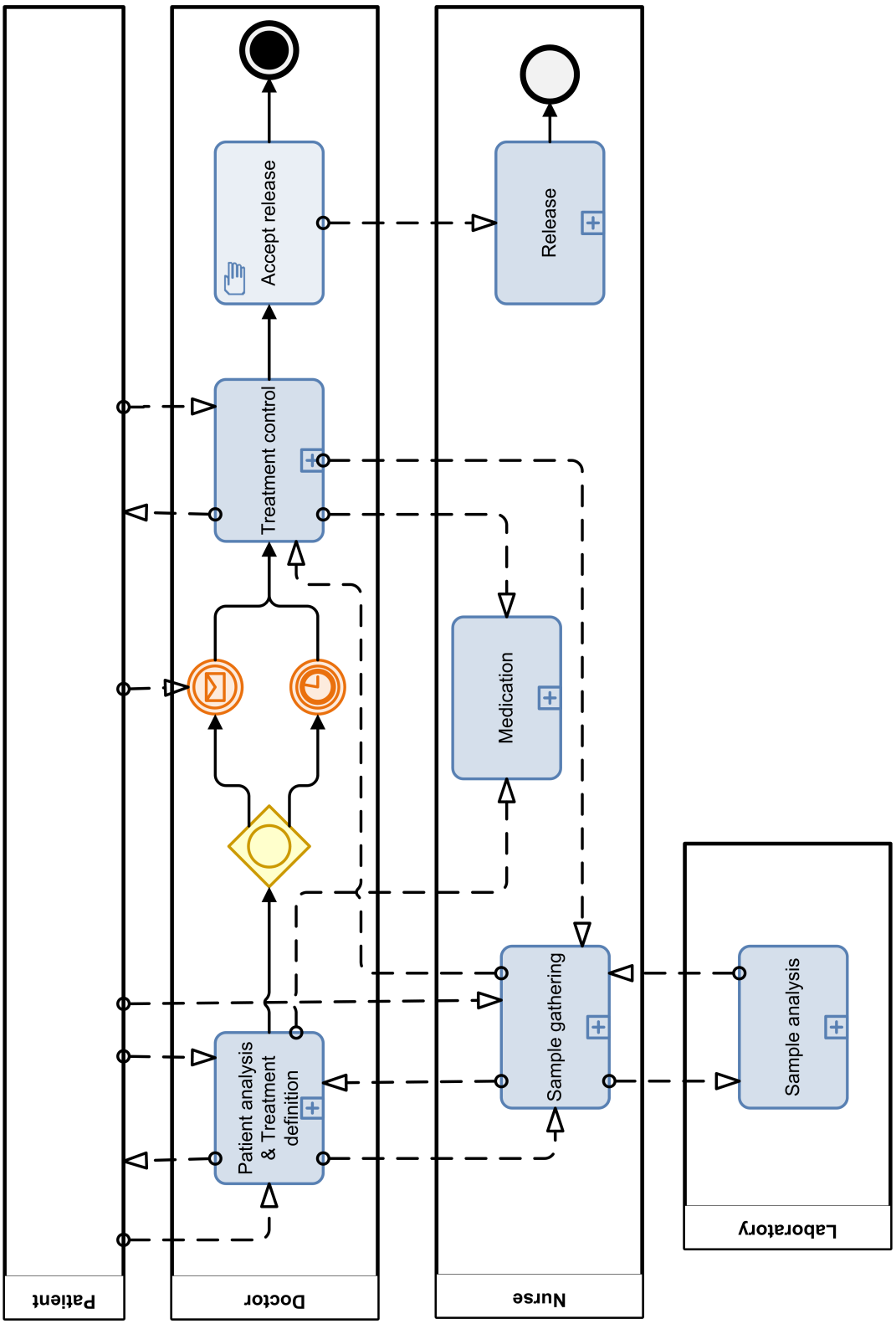
B.1 Stationary treatment process (in BPMN 2.0)



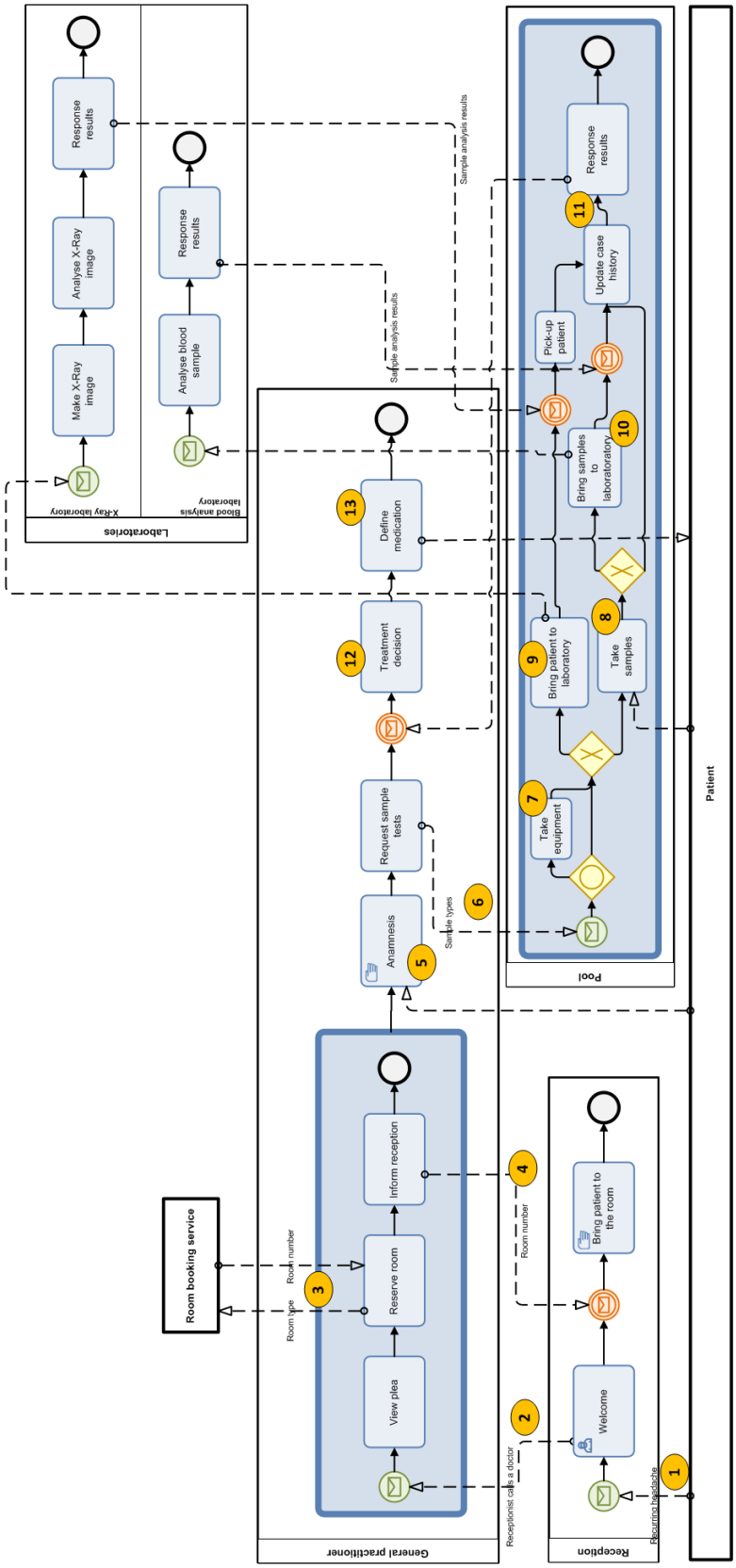
B.2 Stationary treatment process with appropriate service candidates (without data objects)(in BPMN 2.0)



B.3 Treatment process (collapsed) (in BPMN 2.0)



B.4 Demo scenario process model (in BPMN 2.0)



Bibliography

[1] ALONSO, G. ; CASATI, F. ; KUNO, H. ; MACHIRAJU, V.: *Web Services: Concepts, Architectures and Applications*. Springer Publishing Company, Incorporated, 2010

[2] BAHREE, A. ; MULDER, D. ; CICORIA, S. ; PEIRIS, C. ; PATHAK, N.: *Pro WCF: Practical Microsoft SOA implementation*. Apress, 2007

[3] BECKETT, D. ; MCBRIDE, B.: RDF/XML syntax specification (revised). In: *W3C recommendation* 10 (2004)

[4] BORENSTEIN, N.S. ; FREED, N.: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. (1996)

[5] CHRISTENSEN, E. ; CURBERA, F. ; MEREDITH, G. ; WEERAWARANA, S. u. a.: *Web services description language (WSDL) 1.1*. 2001

[6] CONSORTIUM, UDDI u. a.: UDDI executive white paper. In: *OASIS, November* (2001)

[7] ENDREI, M. ; ANG, J. ; ARSANJANI, A. ; CHUA, S. ; COMTE, P. ; KROGDAHL, P. ; LUO, M. ; NEWLING, T.: *Patterns: service-oriented architecture and web services*. IBM Corporation, International Technical Support Organization, 2004

[8] ERL, T.: Service-oriented architecture: concepts, technology, and design. In: *New York* (2005)

[9] FETTE, I. ; MELNIKOV, A.: The websocket protocol. (2011)

[10] FIELDING, R.T.: *Architectural styles and the design of network-based software architectures*, University of California, Diss., 2000

[11] GU, Tao ; PUNG, Hung K. ; ZHANG, Da Q.: A middleware for building context-aware mobile services. In: *Vehicular Technology Conference, 2004.*

VTC 2004-Spring. 2004 IEEE 59th Bd. 5, 2004. – ISSN 1550–2252, S. 2656 – 2660 Vol.5

[12] HALTEREN, A. van ; PAWAR, P.: Mobile Service Platform: A Middleware for Nomadic Mobile Service Provisioning. In: *Wireless and Mobile Computing, Networking and Communications, 2006. (WiMob'2006). IEEE International Conference on*, 2006, S. 292 –299

[13] HAMMERSCHALL, U.: *Verteilte Systeme und Anwendungen: Architekturkonzepte, Standards und Middleware-Technologien*. Pearson Studium, 2005

[14] HUANG, Y. ; GARCIA-MOLINA, H.: Publish/subscribe in a mobile environment. In: *Wireless Networks* 10 (2004), Nr. 6, S. 643–652

[15] KLYNE, G. ; CARROLL, J.J. ; MCBRIDE, B.: Resource description framework (RDF): Concepts and abstract syntax. In: *W3C recommendation* 10 (2004)

[16] LANGER, David ; REICHERT, M. (Hrsg.) ; HALLERBACH, A. (Hrsg.) ; PRYSS, R. (Hrsg.): *MEDo: Mobile Technik und Prozessmanagement zur Optimierung des Aufgabenmanagements im Kontext der klinischen Visite*. `http://dbis.eprints.uni-ulm.de/806/`. Version: April 2012

[17] LASSILA, O. ; SWICK, R.R. u. a.: Resource description framework (RDF) model and syntax specification. (1998)

[18] MCCARTHY, P. ; CRANE, D.: *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, 2008

[19] NATIS, Y.V.: *Service-oriented architecture scenario*. 2003

[20] PRYSS, R. ; TIEDEKEN, J. ; KREHER, U. ; REICHERT, M.: Towards Flexible Process Support on Mobile Devices. In: SOFFER, Pnina (Hrsg.) ; PROPER, Erik (Hrsg.) ; AALST, Wil (Hrsg.) ; MYLOPOULOS, John (Hrsg.) ; ROSEMANN, Michael (Hrsg.) ; SHAW, Michael J. (Hrsg.) ; SZYPERSKI, Clemens (Hrsg.): *Information Systems Evolution* Bd. 72. Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–17722–4, S. 150–165

[21] SCHULTE, R. ; YEFIM, V.: NATIS: SSA Research Note SPA-401-068, Service Oriented Architectures / Part. – Forschungsbericht

[22] TANENBAUM, A.S. ; STEEN, M. van: *Verteilte Systeme: Grundlagen und Paradigmen*. Pearson Studium, 2008

[23] WAKHOLI, P. ; CHEN, W. ; KLUNGSØYR, J.: Workflow Support for Mobile Data Collection. In: *Enterprise, Business-Process and Information Systems Modeling* (2011), S. 299–313

[24] WILKINS, G. ; SALSANO, S. ; LORETO, S. ; SAINT-ANDRE, P.: Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. (2011)

[25] ZAPLATA, S. ; HAMANN, K. ; KOTTKE, K. ; LAMERSDORF, W.: Flexible execution of distributed business processes based on process instance migration. In: *Journal of Systems Integration* 1 (2010), Nr. 3, S. 3–16

Name: Georgy Karpenko Matrikelnummer: 536559

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Georgy Karpenko