JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Integrated Communication Management

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Mag.rer.soc.oec.

im Diplomstudium

WIRTSCHAFTSINFORMATIK

Eingereicht von:
Kamil Sarelo, 0155096

Angefertigt am:
Institut für Telekooperation

Betreuung:
Univ.Prof. Mag. Dr. Gabriele Kotsis

Linz, August 2009

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Kamil Sarelo
Linz, August 2009

Danksagung

Ich möchte mich bei Univ.Prof. Mag. Dr. Gabriele Kotsis für die Berücksichtigung individueller Interessen sowie ihre Anregungen, Unterstützung und Rückmeldungen bedanken. Darüberhinaus möchte ich auch eigens meinen Eltern für ihren Beistand und ihre Geduld herzlichst danken.

Kurzfassung

Diese Diplomarbeit stellt ein Applikationsframework zur integrierten Verwaltung von Kommunikation vor. Das Framework – benannt nach dem Götterboten der griechischen Mythologie HERMES – soll gemäß der Vision das Fundament für Applikationen stellen, die in der Lage sind abhängig von einer Menge von Parametern auf Kommunikationsbedürfnisse von Anwendern zu reagieren und entsprechende Werkzeugunterstützung zu bieten. Darüberhinaus sollen Applikationen dieser Domäne Anwender während der Arbeit beobachten und aus Erfahrung lernen, sich mit Sensoren verbinden und Schnittstellen zur existierenden Kommunikationsinfrastruktur bieten. HERMES unterstützt diese Funktionen in der in dieser Arbeit beschriebenen Art und Weise. Das Framework basiert auf einer serviceorientierten Architektur, was eine hohe Flexibilität und Modularität sichert. Es bietet Dienste und Schnittstellen für die verteilte Verwaltung von Kontakten und Daten generell. Zudem wird der Zugriff auf beliebige Sensordaten und existierende Kommunikationswerkzeuge ermöglicht. Dabei erlaubt die Struktur dieses Frameworks daraus entwickelte Applikationen zugleich verteilt in heterogenen Netzwerken zu betreiben, wobei HERMES die Replikation sämtlicher Daten autonom vornimmt.

Abstract

This thesis presents an application framework for integrated communication management. The framework – called HERMES after the messenger of gods in Greek mythology – is to provide a solid foundation for applications, based on a set of known parameters about users' communication needs, and present appropriate tool support. Furthermore, applications of this domain will be able to observe users during operation, learn from experience, connect to sensors and in conclusion provide interfaces to existing communication infrastructure. HERMES supports these functions in the manner described in this work. The framework is based on the service-oriented architecture, whereof it benefits by great flexibility and modularity. It offers services and interfaces for the general distributed management of contacts and data. In addition, access to any sensory data and any existing communication tools is granted. The framework structure also makes it possible to run HERMES-based application in distributed heterogeneous networks simultaneously, in which case the framework replicates data autonomously.

Table of Contents

List of Abbreviations

API	Application Programming Interface
CORBA	Common Object Requesting Broker Architecture
EDA	Event-Driven Architecture
GPS	Global Positioning System
IO	Input/Output
JDBC	Java Database Connectivity
JID	Jabber Identifier
JPA	Java Persistence Application Programming Interface
JVM	Java Virtual Machine
NTP	Network Time Protocol
ODBMS	Object-Oriented Database Management System
ORM	Object-Relational Mapping
OSGi	Open Services Gateway initiative
P2P	Peer-to-Peer
RDBMS	Relational Database Management System
RFID	Radio-Frequency Identification
PSTN	Public Switched Telephone Network
SOA	Service-Oriented Architecture
SQL	Structured Query Language
UUID	Universally Unique Identifier
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XPath	XML Path Language

List of Figures

List of Tables

List of Listings

1 Introduction

The goal of this thesis was to design and implement a framework for integrated communication management. Integrated communication management, within the scope of this work, can be described as the attempt to support users' communication needs and communication-related behavior patterns by providing appropriate tool support based on precedent data mining and resultant knowledge. The developed framework – called HERMES after the messenger of gods in Greek mythology – supports the data mining process by providing the infrastructure to observe users during operation (e.g. data may be obtained from sensors) and to learn from experience (e.g. by using learning algorithms or artificial intelligence). Furthermore, it provides a solid foundation to connect to sensors and to existing communication infrastructure.

The idea to develop an integrated communication management framework was essentially inspired by the work presented in [1]: *"People are nomadic. They move around to work, to shop, or to play. Increasingly the places they enter are computerized. Their workplaces are filled with computers; the shops they browse are computerized; the toys they buy are computerized. And communications networks reach into every corner: web access continues to soar; cell phones abound, new wireless technologies are emerging. The convergence of these increasingly pervasive computers and communications networks offers new opportunities and challenges for systems designers, which are being addressed in the fields of pervasive …, ubiquitous, nomadic … and context-aware … computing."* Although the authors of [1] focus on how to augment digitial appliances and objects like printers, radios, and automobiles with web technology by implementing an automatic discovery of non-electronic objects, they make a point about a paradigm which can be called device independent communication respectively reachability or ubiquitous connectivity in simple terms. This paradigm can basically be described as nomadic users being able to connect and communicate independent of their constantly changing locations, underlying communication protocols and tools as well as autonomous software assistance during communication tasks based on past behavior as referred to integrated communication management.

In order to visualize the idea of integrated communication management, the use case diagram shown in Figure 1.1 represents numerous ways for the intended use of such a system. As expressed before, applications are able to support the users' communication needs and present appropriate tool support. While working with an integrated communication management application based on the HERMES framework, users are able to directly perform various actions typical for the communication domain. These actions include placing calls, starting conferences, sending messages – including instant

messages and e-mails – and finally setting presence information. All these actions are routed to an appropriate communication tool, which is then responsible for their actual execution. There is also the possibility to input and output data in a generic way. The framework additionally enables users to manage persistent and distributed data and contacts. Moreover, HERMES is also able to run its own autonomous tasks, for example it can interact with sensors, post and handle sensor events or even execute actions autonomously. On top of that, HERMES-based applications are able to replicate their entire databases among themselves without any special assistance by the user.

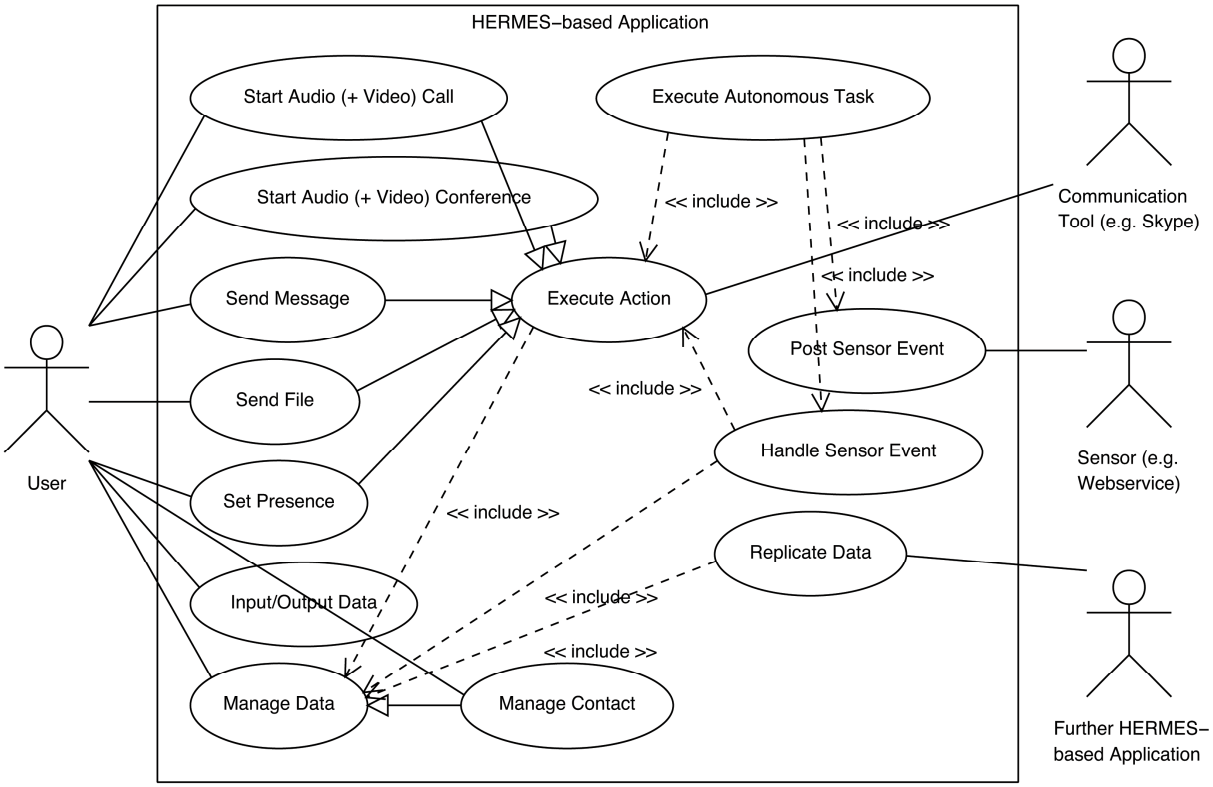

Figure 1.1: Use case diagram of a typical HEMRES-based application.

Figure 1.2 illustrates a sequence diagram with two instances of HERMES-based applications and two actual users: Alice and Bob. Based on the diagram one can assume that Alice possesses two personal devices: alice1 and alice2 (e.g. two desktop computers), each running a HERMES-based integrated communication management application. In the case that Alice adds new contact information on alice1 then this information would also be replicated and made available on alice2. In case Alice intends to call Bob, the execution of this action may depend on the location of the personal device. This information can be retrieved from sensors or be provided as an application setting. For example if alice1 is the home computer and alice2 is the office computer, then the application on alice1 would dial Bob's private number while the application on alice2 would dial Bob's office number. Both numbers are stored in Bob's contact information and are available on both devices. Additionally, the

application could also use different tools like Skype for alice1 and a regular PSTN device for alice2. The decision-making knowledge about proper action handling would be derived from previously gained facts and experience while observing Alice during operation on alice1 and alice2.

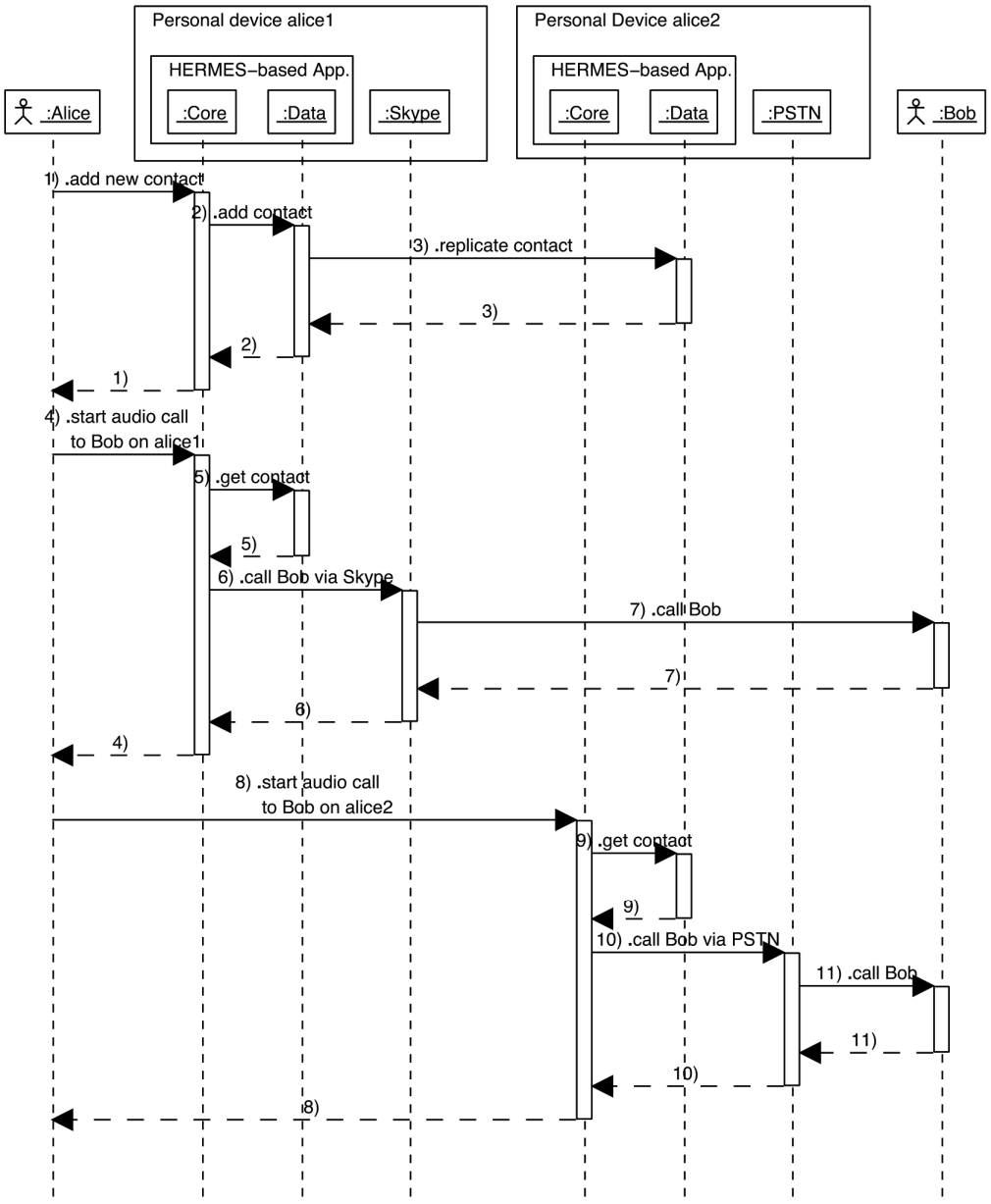

Figure 1.2: Sequence diagram of Alice calling Bob on multiple personal devices.

One can think of ample possible application domains for such systems, e.g. easy and instant file transfers by just dragging and dropping a file on a particular contact without the prior need to select a specific communication tool or protocol. Another example would be an automatic presence-change application when the user is relocating geographically with or even without his personal device. In the first case, an integrated communication management application could determine the user's current

geographical position by using sensory data obtained from an integrated GPS service. In the second case, an application running on several personal devices could determine the user's geographical location by using RFID sensors at each device's location and trying to read a RFID tag the user would carry with him. Yet another possible application for such systems is machine-to-machine communication where one can imagine applications running on multiple devices autonomously performing contiguous communication tasks and negotiating the execution of these tasks amongst each other.

While discussing the mechanism of HERMES-based applications, the framework itself has to be explained. The HERMES framework combines several architectural styles for software development, namely: service-oriented architecture, event-driven architecture, and peer-to-peer architecture. As presented in [2] SOA supports the development of fast, inter-operable and massively distributed software. Services are the major idea behind SOA and represent autonomous, platform-independent objects that are meant to be described, published, discovered, and loosely-coupled while performing a large variety of tasks. These can range from simple duties like answering straightforward requests to much more sophisticated business processes. SOA enables the re-use of existing code and application components and to transform these into services available in the network. The development of service-oriented applications requires an approach that utilizes the discovery and implementation of network-available services in order to execute tasks [2]. Furthermore, in terms of EDA the authors of [3] accurately state that events and messaging have been used throughout software development for years. However, due to the evolution of service-oriented and event-driven architecture the event archetype became a fundamental piece of software. Eventing has been seen as a general architectural paradigm for a while. More recently, this paradigm has broadened the service-oriented and event-driven architectures. SOA does not work just because of the communication over HTTP. Services built according to the SOA paradigm are meant to be autonomous, loosely-coupled, modular, and be able to handle requests, exceptions, and general changes. According to the authors of [3] EDA is versatile enough to handle these issues and additionally manifest the real world's event-driven nature. According to the work in [4] and in regard to peer-to-peer architecture, it can be said that a substantial amount of research on this topic has been conducted in recent times due to the popularity of file sharing systems based on P2P. P2P computing can be described as distributed computing involving a large number of independent peers cooperating and sharing resources and services. P2P networks are basically logical overlay networks formed by peers linking to other peers. As described in [4] querying inside a P2P system works in the following way: A query describing data of interest is published and spread through the network until the query arrives at peers able to provide suitable data and any matching result is returned to the inquirer. However, P2P systems go beyond the probably best known

representatives like file sharing applications. P2P offers a way to build systems that are characterized by increased decentralization and self organization. Vast amounts of computing power, storage and connectivity from around the world may be utilized for a range of applications like network monitoring and routing, large scale event/notification systems and web search [4]. In the opinion of this thesis, P2P perfectly suits the need of the HERMES network backbone, which is to share and replicate distributed data amongst HERMES peer.

Figure 1.3 shows the framework's abstract architecture. In compliance with SOA the framework consists of numerous services and components dedicated to accomplish particular tasks and was designed and built on top of the Java-based OSGi service platform.

According to [5] OSGi follows the SOA concept like Web Services, MS .NET, and CORBA services. OSGi is a framework creating an execution environment for applications and providing functionalities including component management, service registry, and Java class loading. OSGi services are defined by Java interfaces, which describe their functions. Implementations of such Java interfaces are packaged into a so-called bundle. A bundle is a Java jar archive file enhanced by additional OSGi-related manifest headers in the manifest file. These headers provide information about the bundle to the OSGi framework. The framework, in turn, defines an individual life cycle for all bundles where every bundle goes through the stage of null, installed, resolved, starting, stopping, active, and uninstalled. One feature of the OSGi framework that needs to be emphasized at this point is hot deployment, the ability of installing and uninstalling bundles while the framework is running [5]. The most important difference between OSGi and technologies like CORBA is that OSGi is a distinctly Java Virtual Machine technology. It is not directly related to remote invocation and may be referred to as SOA in a JVM. It offers a service-oriented architecture within a single JVM since it runs entirely within a single JVM process. This makes the framework very lightweight, the communication between services is only marginally slowed down, and everything resides on a single piece of hardware interacting with no measurable latency. These were the main reasons to choose OSGi for this thesis.

Figure 1.3 also shows the essential components of the HERMES framework: sensors, events, rule bases, tools, inputs, outputs, protocols, actions, contacts, database, and replication. Sensor services along with rule base services, tool services, and input/output services are easy to inter-change. That means anyone can develop these kinds of services and run them within the framework. All other components are provided by a single framework OSGi bundle and may only be extended if the framework itself is extended concurrently.

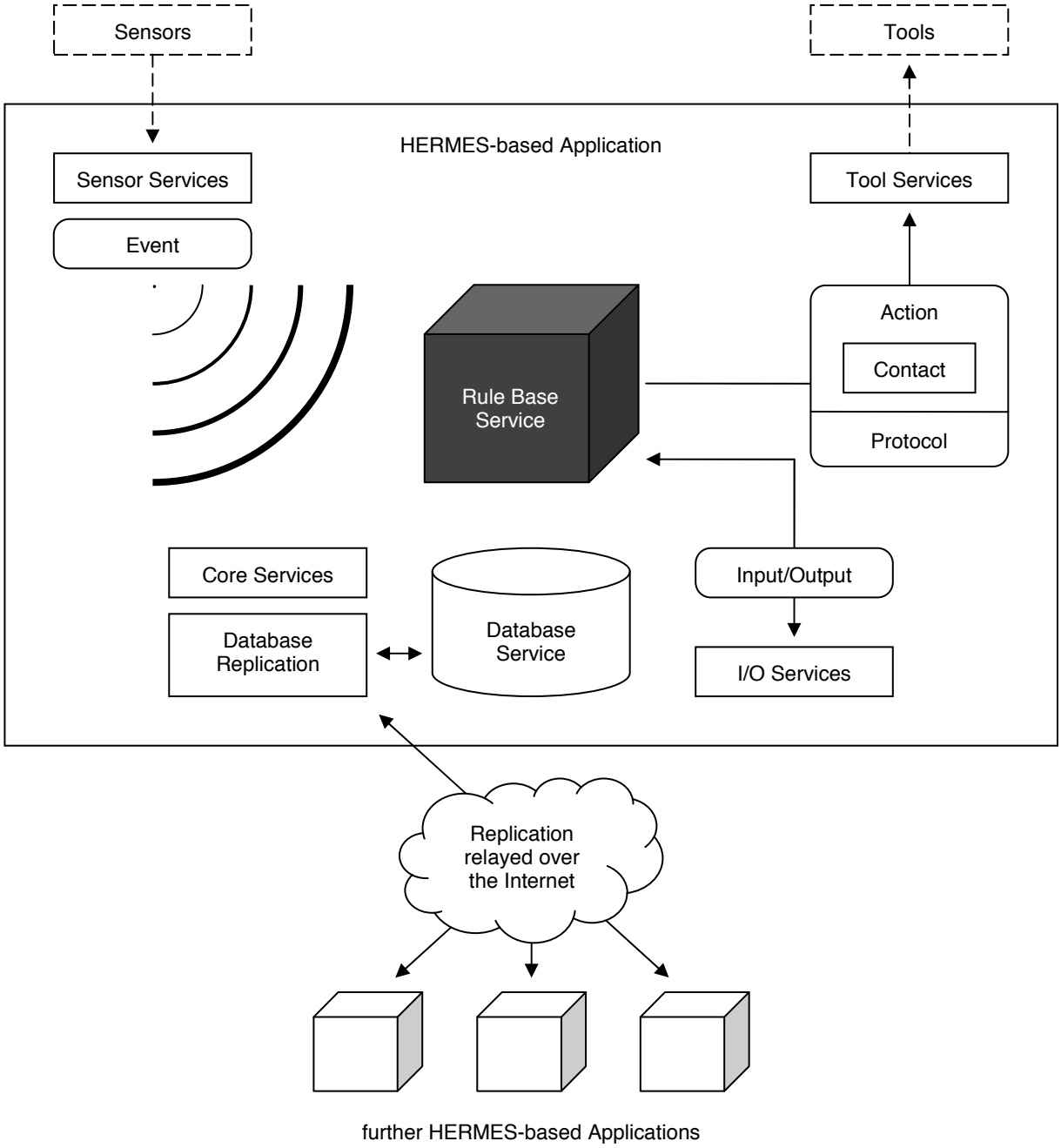

Figure 1.3: Abstract architecture of the HERMES framework.

As implied by Figure 1.3 the fundamental operation of the HERMES framework starts with sensors. These components are responsible for data mining and usually consist of bundles operating directly within the framework. They can also be wrappers for sensors outside of the Java Virtual Machine like for example sensors consuming data from web services. Once a sensor receives new data, its function is to publish this data to all components in the framework via an OSGi event. Depending on the application, a sensor can do this automatically (e.g. timestamp beacon broadcasting the current timestamp every second) or it can be triggered by other services to publish. Once the events are

published, the OSGi platform routes them to interested recipients. Usually only rule base services are interested in these events. HERMES sets very few boundaries for rule base services. These components can be understood as the brain of the entire system, their structure being created by the developer. The vision is to gain knowledge of users' communication needs, recognize behavior patterns, and learn from this collected data. Applications relying on this knowledge should be able to provide appropriate communication tool support and execute appropriate communication-related actions. However, the HERMES framework is just the imperative foundation and the actual advanced rule base services like learning algorithms or artificial intelligence are beyond the scope of this thesis. Yet, once a rule base service has identified the appropriate action to be taken after receiving an event, it has the possibility to send actions to tool services or send inputs/outputs to input/output services. The framework provides a fixed set of actions, inputs, and outputs that can be used for this purpose. These components require additional data, such as strings and contacts before they are forwarded to an appropriate service. Inputs and output can be passes to input/output services, which are then responsible for acquiring user input specified by the passed input object or for outputting the information specified by the passed output object. This generic approach enables to switch between input/output methods and to develop further input/output methods for different devices such as: text, speech, gestures, keyboard, microphone, display, speakers and so on. Actions – e.g. for calling a contact or for sending a message to multiple contacts – can be passes to tool services, which are then responsible for executing the actions. Tools can be bundles operating directly within the framework. They can also be wrappers for tools outside of the Java Virtual Machine like a tool interacting with the Skype communication tool. Actions generally require contact data along with some additional information. Since not every action is supported by every tool, they – like contacts – need some common ground which in this case are protocols. For example, if an application wants to call a user via Skype, a rule base would have to send an appropriate action along with contact information containing a Skype username to the appropriate tool. On the one hand the contact must support the Skype protocol. On the other hand this action needs to be passed to a tool that supports the Skype protocol itself. Protocols represent the interface between contacts, actions, and tools ensuring fault-less operation of the framework. Additionally, HEMRES offers solid data persistency by providing a database management service and a specialized contact management service, which is designed on top of the database service. Generally, all data in HERMES – particularly Java objects – needs to be serialized to XML before being handed over to the database service. The framework drives an embedded relational database management system in the background and is able to store data in common relational tables as well as XML-serialized data. In addition, the structure of the framework makes it possible to run numerous HERMES-based applications for one user or for multiple users in distributed heterogeneous networks at the same time. In this case, the framework deals with all data

replication between distributed peers autonomously and de facto provides a distributed data persistency layer. Besides all these features the framework offers other core services that will be presented in the subsequent chapters along with detailed information about HERMES' operating mode.

The following part of this thesis is structured in three major sections. The first section consisting of Chapter 2 focuses on related work. Firstly, it presents scientific work related to general topics of importance for this thesis. Then it presents technologies related to this work, existing solutions and systems before finally demonstrating how these were incorporated into HERMES. It also illustrates the differences to this work and the contributions of this work.

The second major section consists of two Chapters discussing the inner life of HERMES. In Chapter 3 the architecture and the reference implementation of the framework are explained in-depth. This chapter also explains why certain architectural decisions were made, why specific technologies were chosen, which dead ends were encountered during implementation, what results were achieved and what conclusions were drawn from the entire process. Chapter 4 describes the ways to build new applications based on HERMES. It guides roughly through the process of developing a complete integrated communication management application based on the framework presented in this thesis. Additionally, Chapter 4 introduces the concepts of extending the framework itself. This is very useful in case of necessity for new protocols, actions, inputs or outputs.

Final thoughts, a short critical evaluation as well as the major contributions of this thesis and possible future work form the third and last section of this thesis, consisting of Chapter 5.

2 Related Work

In this chapter, the general topic associated with this thesis, namely device independent connectivity, will be addressed first. The following sub-chapters cover research on further important subjects associated with this thesis: communication and contact management, software architecture, internal communication and data replication. Each individual sub-chapter will present certain solutions, compare these to the HERMES framework and give explanations as to what was integrated in this work and what it contributed.

2.1 Device Independent Connectivity

As stated in the introductory chapter, the idea for an integrated communication management framework was fundamentally inspired by parts of the work described in [1] whereby that work's main contribution is basically a nomadic web. The authors present a project called Cooltown, in which the possibilities of adapting the web infrastructure to support nomadic users and possibilities to embed web technologies in digital appliances and physical objects like printers, radios, and automobiles have been explored. The purpose was to eventually allow users to come into contact with web resources not associated with electronic objects. In addition, the authors linked physically related objects with places on the web and investigated ways for humans to utilize new digital communication devices to interact with these web places. Subsequently, the work describes how they integrated web technologies to improve interaction and offered location-based services between nomadic people and the places they visit and the objects they encounter. The basic idea discussed in [1] is that when people move from place to place and interact with other people and objects at these places, then these places generally become special containers for people and objects. This fragmentation into people, places, and objects helps the authors to introduce systematic entities relevant to the end user. One such entity is an infrastructure for people called WebLink. *"WebLink uses a person's point of web presence as a level of indirection for electronic communications. A user such as Harry, with whom Veronica needs to communicate, places a link to a WebLink redirector service in his conventional personal web page. The WebLink redirector service runs on a globally accessible web server. When Veronica clicks on the link in Harry's page, an invocation is made to the WebLink redirector service to request a resource for communicating with Harry. If the web redirection service possesses a URL for communicating with Harry in the place where he currently resides, it returns that URL to Veronica's PDA. However, if Harry is offline then the redirector returns a web page that includes a service to leave a message for Harry. This information will be delivered to Harry when he comes on-line. The web redirection service is updated with the URLs of suitable communication services as Harry moves around, as long as he wishes to be reachable. There are various ways of implementing the update mechanism,*

reflecting different choices of responsibility between the user's PDA and web servers in the places that host the user. We have implemented an update service that runs in each web-present place. When Harry enters a place, he may identify himself to the PlaceManager through his PDA; if he wishes to be contacted there, his PDA also supplies the URL of his redirection service. The place-level service identifies the URL of a suitable communication service for Harry in the place – unless, as in our example in the previous subsection, Harry nominates his own personal device. It registers this choice of communication channel with Harry's redirection service. The service will then offer that channel via the link in Harry's web page. With WebLink, the communications channel that Harry and Veronica use can depend upon their locations and preferences without the usual trial-and-error used in electronic communications currently." [1] Although the authors primarily describe web presence as a means of connecting real-word objects with the web, they introduce the WebLink infrastructure for connecting people. WebLink points up a particular concept of this thesis: the concept is a paradigm referred to as device independent communication respectively reachability or ubiquitous connectivity. The WebLink redirector service described in [1] provides constantly updated URLs for suitable communication services to the participating communication partners. While a participant moves around, a service identifies URLs of suitable communication services and broadcasts these channels via the user's web page. The communication channels in use depend upon the users' location and preferences. The same demands were to be met by the HERMES framework presented in this thesis. HERMES-based applications provide appropriate communication tool support depending on a set of parameters, observe users during operation, learn from experience, connect to sensors, provide interfaces to existing communication infrastructure and include mechanisms to manage distributed data. As distinguished from the work shown in [1], HERMES as a framework doesn't provide all listed functions out of the box. However, the framework offers services and interfaces to do so. It grants access to any sensory data and any existing communication tool via interfaces and events. Moreover, it is also possible to run HERMES-based application in distributed heterogeneous networks simultaneously, whereby the framework replicates all data autonomously. How HERMES-based application act in particular case scenarios depends on the underlying structure, which may be hot swapped during operation and is the responsibility of the developer.

2.2 Communication Management and Contact Management

In [6], Groth discusses a technological framework supporting knowledge exchange in organizations that feeds the idea of communication management presented in this thesis. Groth states that the framework considers three aspects related to communication management: communication itself, awareness and information management. He adds that a universal application solution satisfying needs of all people is hard to design and build, mainly because usability primarily depends on people's

routines and other applications people use. However, in Groth's opinion it is most likely that the three aforementioned aspects will be a fundamental part of any software involved in communication management. Although the focus of [6] is knowledge exchange in organizations, Groth illustrates the basic idea of communication management, where a communication-related application re-uses existing knowledge while supporting people's needs. In this context Groth poses two questions, also of importance for the HERMES framework: How can communication management support communication between individuals? And how can communication management support awareness of people's activities and availability? The author then presents multiple communication aspects of his framework gained from studies: communication can take place either synchronous or asynchronous; communication can occur between two persons or from one person to another group; a communication tool can be used on one desktop or on multiple platforms. He also refers to ***awareness*** as an important aspect of communication management. During studies, the author observed that people tend to ask a person nearby instead of using the phone to get aware of someone's availability and activities. In his opinion, such information could be presented automatically in communication-related applications as it is or compiled into a single representation. Information about people's activities and availability is usually derived from various inputs like sensors, electronic calendars or keyboard activity so that presenting this information as it is would require the user to interpret it. The author suggests a different approach, in which information would be computed on the basis of the supplied inputs by an algorithm. All these aspects enumerated in [6] were taken under consideration during the conceptual design of the HERMES framework too. On the analogy of Groth's proposal, the communication management in HEMRES supports synchronous and asynchronous communication and awareness information. It also re-uses existing information, combined in a universal framework and can also be used on different platforms.

Further need for communication management has been discussed in [7] and [8]: "*With telecommunication networks beginning to migrate towards Internet technology, users are starting to enjoy the integration of data and communication services … The expectation is that they will be able to access the Internet from anywhere and any time, using any device, and at the same time be reachable by other users. This makes the issue of user mobility an important aspect in the design and development of the next generation of mobile communications and computing systems.*" [7] "*The research of user mobility to date appears to have focused on two main areas: Personal Mobility and Terminal Mobility. Personal Mobility concentrates on the movement of the user, while Terminal Mobility concentrates on the movement of the user's terminal … The first addresses the issues associated with enabling a user to be contacted by another user, regardless of her location and the device she uses – contactability. The second addresses the issues associated with personalising a*

user's operating environment regardless of the terminal or network she is using – personalisation." [8] Consequently, the works in [7] and [8] present the Integrated Personal Mobility Architecture (IPMoA), supporting aspects of personal mobility. The framework contains three mobile agents that perform numerous communication-related tasks. These tasks include executing applications, synchronizing files, and communicating with other instances in the network. Moreover, the IPMoA framework defines an execution environment for these agents that can exist either in the network or on the end user's system. One of the mobile agents presented in [7] is of particular interest for this thesis: the Personal Communication Assistant (PCA). PCA was introduced to handle the personal communication aspect of IPMoA. Its main purpose is to set up calls, whereby it is able to locate the receiver of the call, signal this callee for call initialization, and to install an appropriate plug-in in order to overcome potential incompatibilities. This mobile agent is the representation of the user in the network. Before any call session between two users can be set up, their location must be determined. Based on that information, the PCAs decide if any additional plug-in is required, conduct its installation and finally set up the call session. Similar to IPMoA, the HERMES framework utilizes an execution environment for itself and its components, but unlike IPMoA the environment is always located at the end user's terminal. A comparable functionality to PCA has also been embedded into the design of the HERMES framework. HERMES-based applications are able to support personal mobility of users by operating in distributed heterogeneous networks at the same time. Personalization of operating environments is also ensured due to the Java-based implementation. On the subject of personal communication, HERMES is able to add necessary functions by installing an appropriate plug-in on demand during runtime. The framework's data is available everywhere due to the autonomous database replication. However, in contrast to IPMoA HERMES based-applications are not mobile agents, since HERMES was not designed to wander on the network.

In the domain of contact management, Epp's work reveals a number of difficulties regarding contacts and their management: *"Currently, we exchange documents through email, instant-message our buddies, and remain in cell phone contact with our colleagues. However, the applications that make this possible are developed with an outdated mindset. They are developed on top of devices meant for personal use, not group interaction. Word processors assume that only a single user owns a document, calendars belong to an individual, and each application maintains it own list of buddies. To further complicate this problem, the device we use defines our communication address: cell phone number, email address, laptop URL, or business address."* [9] Epp basically says that contacts may have multiple communication addresses, an aspect also encountered during the design of the HERMES framework. The author tries to solve this problem by introducing so-called Relationship Management. Contacts have a high long-time value and thus relationship management defines a central storage for

contacts, where contact information is persisted. Relationship management supports interoperability by making this contact information available to any application. Contacts themselves are defined by Epps as: *"an abstract representation of a resource that contains all the information necessary to define a relationship with some entity. A contact object captures this information as a set of attributes that specify what contacts are and what they can see, publish, and do in a group. Each contact must be uniquely identifiable and addressable."* [9] The HERMES framework has been limited to contain contact information consisting of multiple addresses (or identities) for different protocols a contact may use. Relationship information was not of particular interest in this thesis. However, due to the modular and flexible HERMES architecture, contacts may support any desired additional information.

In addition to a definition of contacts Epp also discusses a solution for distributed contacts or distributed data management in general. Epp states that in any collaborative system the ownership of contacts must be determined at an early stage. In his proposal, Epp chooses to store all contact information on distributed clients utilizing P2P, because in his opinion this approach does not rely on a connection to a central server. This approach works very well with ad hoc local networks. The focus on P2P interaction eliminates certain security issues associated with a server and moving sensitive knowledge about contacts to the peers. The HERMES framework uses the same approach for contact management. Some characteristics mentioned here already highlight the P2P approach and will be discussed later in this thesis.

In [10] the authors present an empirical study showing that shared activities in teams require a disproportionate amount of individual work, which include finding references to people, finding email attachments or managing different files involved in these activities. The authors state that a great portion of that work involves finding recently accessed files. Their observations led to the creation of Recent Shortcuts, a tool helping users to access recently used objects more easily. This idea mirrors the vision for the HERMES framework, where HERMES-based applications help support communication coordination by automatically enabling quick access to contacts and objects such as files, based on previously observed communication behavior of the user. The authors of [10] add that recently the focus has shifted from the traditional form of organizing around software and objects to research on how to arrange users' interaction according to their tasks and activities. [10] claims that information increasingly becomes fragmentized and general work with computers involves more and more ineffective work like switching among and integrating across software and files. In the opinion of this thesis the same phenomenon can be observed in the domain of communication. Hence, the HERMES framework counteracts the information fragmentation in communication by managing users' communication-related interactions according to their activities and tasks across computers.

2.3 Architecture

When speaking of architectural styles applied in the HERMES framework, sensor networks have to be addressed first. The work presented in [11] addresses a particular hybrid, the agent-based architecture for distributed sensor networks, which shares some similarities with the HERMES framework. *"We suggest an agent model that integrates the software agents of the application with the hardware agents of the physical environment … Issues concerning how to build interoperable, programmable applications on such networks that involve management, information gathering and fusion, querying and tasking of/from these nodes are of practical importance. Agent based systems represent a new software technology that has emerged from merging two technologies, namely, artificial intelligence (AI) and object oriented distributed computing."* [11] The authors define software agents as: *"… intelligent, autonomous, software components' capable of interacting with others within an application, attaining a common goal and thereby contributing to the resolution of some given problem. They are important because they interoperate within modern applications that involve information fusion and mining, retrieval and prediction. Multi Agent Systems (MAS) are composed of a set of agents and are useful for the modeling of distributed information systems with synchronous or asynchronous component interactions."* [11] HERMES-based applications can also be described as hybrid, multi-layered and agent-based applications for distributed management of communication. HERMES enables applications to connect to sensors and to run in distributed heterogeneous networks at the same time. The architectural framework presented in [11] supports three types of communication models, namely inter-node communication, inter-agent communication and agent to node and node to agent communication. The architectural communication model applied in HERMES has been designed accordingly, allowing communication between HERMES-based applications as well as communication between particular components inside the framework, leveraging events and service calls and allowing communication between HERMES-based application and external components like sensors, services or tools.

Agent technology has been deployed in many application domains very successfully and is also proposed for software project management (SPM) in conjunction with communication management in [12]: *"The SPM management areas consist of four objective functions and four facilitator functions. The solution presented by Sauer and Applerath … primarily focuses on the Time Management and certain aspects of the Communication Management functions. Maurer's solution … is applicable to the Scope Management, Time Management and to a certain extent the Communication Management functions. We believe that each of these key processes/functions could successfully be addressed by following a black box approach that is based on agent technology. Each black box consists of collaborative software agents ensuring cooperation, coordination and synergy between the different*

black boxes. Within such a black box a component-based development approach is followed. According to this approach, we use multiple (simple) agents, each with a particular objective, rather than fewer (complex) agents of which each has a long list of tasks to accomplish." According to the work discussed in [11] and [12], software agents serve the task of shared problem solution very well. Since the focus and the main contribution of this work was integrated communication management, the best way to support this complex task seemed to be a black box approach based on agent technology. Consequently, agents or – more precisely – specific components of the HERMES framework attend to different tasks like collecting data from sensors, observing communication behavior or carrying out communication-related tasks via existing communication tools. Moreover, the HERMES framework is based on SOA technology and the components are designed as services resulting in extreme flexibility and modularity. A simple definition of a service in conjunction with SOA is given in [13]: *"A service … is a discreet domain of control that contains a collection of tasks to achieve related goals … This consumer-oriented view of service is central to SOA and differentiates it from object orientation. In OO, an object represents what it is, but in SOA, a service represents how its consumers wish to use it."* Speaking of SOA, a general definition of this architectural style central to the HERMES framework may be found in [14]: *"Service-oriented architectures (SOA) is an emerging approach that addresses the requirements of loosely coupled, standards-based, and protocol independent distributed computing … In an SOA, software resources are packaged as services, which are well defined, self-contained modules that provide standard business functionality and are independent of the state or context of other services. Services are described in a standard definition language, have a published interface, and communicate with each other requesting execution of their operations in order to collectively support a common business task or process …"*

The authors of [14] describe the characteristics of SOA services as follows:

– All functions in a SOA are defined by services.

– All services are autonomous, so external components neither know nor care how services operate. They just expect the services to return a result. The actual implementation and execution is hidden behind service interfaces.

– Service interfaces can be called up, meaning the location of services, the connection protocol, and the infrastructure for the operation are more or less irrelevant and interchangeable.

In this thesis the SOA-based technology OSGi was chosen to power the HERMES framework. *"The Open Service Gateway Initiative, or OSGi (www.osgi.org), is an emergent open architecture that lets us deploy a large array of wide-area-network services to local networks such as smart homes and automobiles. It offers the following benefits:*

– Platform independence. OSGi specifications can be implemented on different types of hardware devices and operating systems because they're based on Java.

– Various levels of system security, allowing digital signing of downloaded services and object access control.

– Hosting of multiple services from different providers on a single gateway platform.

– Support for multiple home-networking technologies.

OSGi defines a lightweight framework for delivering and executing service-oriented applications. Its management functionalities include installing, activating, deactivating, updating, and removing services." [15] Apart from OSGi, the work in [15] primarily focuses on context-aware applications. Context-aware applications use a context provided from an entity to alter their behavior in a way that meets the users' expectations. The authors argue that building context-aware applications is a complex and time-consuming task due to inadequate infrastructure. In the opinion of this thesis, context awareness is a fundamental aspect of communication management. HERMES-based applications are specifically designed to use context information derived from sensory data and previously collected knowledge to best meet users' communication needs analogous to the concept presented in [15]. Moreover, it is also the belief of this thesis that OSGi provides an adequate infrastructure for building complex, flexible, and modular context-aware applications. However, it is arguable whether OSGi is an SOA-based technology or not. Sources such as [5] note that OSGi fails in supporting numerous SOA characteristics: *"We have noticed that the difficulties in providing reliability in OSGi framework are mainly due to its limited support of SOA's loose coupling and late binding. Once an OSGi required service is bound, the service is normally accessed until either the user bundle or the service provider service is unregistered. To be fully loosely coupled, the provider service should be selected at use time, i.e., when the user service calls the service."* However, in the understanding of this thesis OSGi succeeds in following the basic concept of service-orientation and therefore represents a legitimate SOA-based technology. The work in [2] explains that services as utilized in OSGi express a service-oriented approach to programming. This approach is founded on the concept of composing applications to perform tasks by discovering and invoking services that are accessible through the network.

Apart from software agents and service-orientation, another architectural style has been utilized in the HERMES framework, namely the event driven architecture (EDA). More precisely, HERMES uses

OSGi-events to forward incoming data from sensors to interested components. According to the works in [3] and [14] this approach can be described as event-driven SOA. *"Events and messaging have been in use in industry for years. With the advent of service-oriented and event-driven architectures, however, the eventing paradigm becomes a central building block of business IT … for some time, we've been witnessing a shift from specific applications to eventing as a general architecture paradigm. The zero-latency enterprise was propagated a while ago only to be superseded more recently by service-oriented and event-driven architectures (EDAs) … Service-oriented architectures don't prosper just because we communicate over HTTP. The components must be designed to serve by making them autonomous, composable, and responsive to input, exceptions, and changes. Event-driven design offers the flexibility to address these issues and reflects the real world's event-driven nature."* [3] *"In the enterprise context, business events, e.g., a customer order, the arrival of a shipment at a loading dock, or the payment of a bill, may affect the normal course of a business process at any point in time … This implies that business processes cannot be designed a priori assuming that events are predetermined following a particular flow, but must be defined dynamically, driven by incoming, parallel and a synchronous event flows. Supporting enterprise applications then must communicate using an event-driven SOA …"* [14] Another precedent of SOA joined with EDA is represented by the Event-Driven Response Architecture (EDRA) discussed in [16]. EDRA facilitates preservation of the application's context and subscribes to appropriate services based on this information. The subscribed services then forward notifications about any changes that could potentially influence the application's context to EDRA. EDRA semi-autonomously decides upon suitable actions after receiving notifications. The system either executes predefined policies or offers a proposal of services that may help the application to comply with the occurred changes. In correspondence to EDRA, HERMES-based applications are to observe users during operation and autonomously evaluate that information to provide better communication support. This information is distributed amongst the services and components in the framework by using events (notifications in EDRA). Upon receiving an event, a HERMES component is to automatically respond with the appropriate action to adapt to changes in the scope of communications.

2.4 Framework Communications

This sub-chapter refers to the framework's internal communication between components and also between HERMES-based applications themselves. The internal communication represents another critical aspect of the overall design of the HERMES framework. The framework integrates two types of internal communication: inter-component communication and inter-application communication.

The internal communication between components has already been addressed at the end of the previous sub-chapter. While being based on the OSGi framework, HERMES utilizes the OSGi Event Admin system service, which provides a universal mechanism to publish and subscribe to events within the OSGi framework [17]. This generic, officially specified approach ensures that all bundles within the OSGi framework are able to deal with events. This service does not build on top of the popular service interface mechanism, which assigns listeners to specific events because of scalability issues and the dynamic nature of the OSGi environment letting bundles appear and disappear at any time. Instead this service allows bundles to publish topic-based events regardless of their recipients and allows bundles to subscribe to specific events. Events are generally published under a topic along with numerous event properties. Subscribers can set up a fine-grained filter controlling the events they receive [18].

The internal communication between numerous HERMES-based applications was a much greater concern during the design of the framework. From the beginning, HERMES was intended to operate in distributed heterogeneous networks simultaneously. By utilizing this ability, HERMES-based applications are able to monitor and assist a user in different environments. For example, during the day an application may observe and aid a user while at work and also provide that same assistance while the user is at home or en route. It is obvious that communication differs from location to location. Applications at various locations have to be aware of this fact and process data in an appropriate way. However, the idea of integrated communication management heavily relies on data and applications being able to collect – also from miscellaneous locations – and share as much relevant data as possible in order to serve user's communication needs best. For this reason the internal communication between HERMES-based applications is primarily intended for data exchange, or more precisely data replication.

In order to efficiently replicate data in HERMES-based applications that operate in distributed heterogeneous networks simultaneously and are extremely nondeterministic in terms of uptime, a correspondingly efficient architecture had to be created. Peer-to-peer technology was considered as the preeminent pick from the very beginning. P2P architectures are considered by the work in [19] as architectures designed to share computer resources like content, storage or CPU cycles by direct exchange instead of utilizing centralized servers or authorities. The authors continue by mentioning the main advantages, namely P2P's ability to adapt to malfunctions and host constantly changing amounts of nodes while ensuring satisfactory connectivity and performance. According to the work in [19] creating an application on top of the P2P architecture may be primarily motivated by the ability to function, scale, and self-organize in environments with fluctuating number of nodes, transient

networks, and computer failures without the need to utilize a central server. The authors of [19] provide the following formal definition of P2P computing: *"The strictest definitions of pure peer-to-peer refer to totally distributed systems, in which all nodes are completely equivalent in terms of functionality and tasks they perform. These definitions fail to encompass, for example, systems that employ the notion of supernodes (nodes that function as dynamically assigned localized mini-servers) such as Kazaa …, which are, however, widely accepted as peer-to-peer, or systems that rely on some centralized server infrastructure for a subset of noncore tasks (e.g. bootstrapping, maintaining reputation ratings, etc). According to a broader and widely accepted definition …, peer-to-peer is a class of applications that take advantage of resources – storage, cycles, content, human presence – available at the edges of the internet. This definition, however, encompasses systems that completely rely upon centralized servers for their operation (such as seti@home, various instant messaging systems, or even the notorious Napster), as well as various applications from the field of Grid computing."* [19] This thesis understands P2P under the terms of the latter definition where even a centralized structure is credited as P2P technology as long as peers share their resources directly and require marginal intermediation of a centralized authority. In the early stages of designing HERMES, JXTA – an open source P2P protocol specification – was considered for handling the internal communication between HERMES-based applications. The authors of [20] introduced JXTA as a set of protocols defining overlay networks on top of physical networks. The protocol specification was created by Sun Microsystems and is now developed by the JXTA Community. The authors add that JXTA may be used to build P2P networks in Java in which resources can be published and searched for. The goal of the JXTA specification is to enable computers to collaborate and converse in a decentralized manner. However, despite the initial excitement about JXTA, it became clear to be the wrong choice for the HERMES framework for several reasons, also discovered by the work covered in [20] and [21]. The authors of [20] explain that a JXTA setup requires knowledge of the network topology. The infrastructure is unable to traverse NATs – a key requirement for their work – without additional intermediate peers, which is not straightforward at all. The authors further complain that while being a complex architecture, JXTA lacks up-to-date documentation and programming. Furthermore, JXTA proved to be extremely inefficient and error prone even when using JXTA within the same network. The authors of [21] add that in their case JXTA failed completely at delivering a robust, general purpose framework to build P2P application on top of. More precisely, JXTA turned out to be too low-level and complex and its messaging framework proved inappropriate even to develop an application using a many-to-many pure P2P communication. Consequently, an alternative to enable the HERMES framework to operate in distributed heterogeneous networks and to avoid the difficulties with configuration and maintenance in JXTA was needed. Both the authors of [20] and

[21] suggested using the XMPP communication platform instead. Their preference for XMPP was based on their experience of XMPP as more stable, easy-to-use, and reliable than JXTA.

At this point XMPP needs to be addressed further since it represents a fundamental technology utilized in the HERMES framework. *"It was, until Jeremie Miller developed the idea of XML streams in 1998 and built a working implementation in the jabberd instant messaging and presence server that he released in early 1999. Since then, many more implementations have been written, the core XML streaming protocol has been formalized by the IETF under the name Extensible Messaging and Presence Protocol (XMPP), and Jabber/XMPP technologies have been extended far beyond the realm of instant messaging to encompass everything from network management systems to online gaming networks to financial trading applications."* [22] Instant messaging is defined by the authors of [20] as follows: *"Instant messaging describes a form of real-time communication between two or more (usually two) persons. In contrast to emails, messages are delivered immediately. Also the availability of the parties is usually visible. Received messages are cached in a local cache and can be read later if the receiving party is not currently near the receiving computer. Instant messaging is usually offered by a service provider as messages are usually sent to its server infrastructure and then forwarded to the respective receiver client."* [20] Instant messaging proved to be not only an extremely effective way of communicating for humans, but also for machines, which is why instant messaging, utilizing the XMPP communication platform, was chosen to power the communication between HERMES-based applications. The author of [22] state that although P2P implementations of XMPP exist, most XMPP implementations follow the client-server paradigm that is familiar from email. Upon connection, both sides open a stream in each direction resulting in two streams. The connection process includes a negotiation on numerous stream parameters like encryption and authentication. Afterwards, each entity is able to send XML stanzas over the stream. Stanzas addressed to entities on a different server require the server to negotiate a server-to-server stream with the foreign server, enabling clients to send stanzas to any addressable entity.

According to [22] the XMPP specification defines the three basic stanza types <message/>, <presence/>, and <iq/>:

- The <message/> stanza represents messages published from one entity to another similar to email.
- The <presence/> stanza represents information about an entity's network ability published to entities that subscribed to that information.
- The <iq/> stanza represents a request made from one entity to another whereby the other entity returns a response being similar to HTTP in some ways.

Based on this information, it is arguable if XMPP – especially the XMPP implementation chosen for this thesis – is a P2P technology. It is the belief of this thesis that this technology is no pure server-client technology because clients logically communicate via addresses directly with clients and XMPP also allows servers to be aggregated into federations. Figure 2.1 illustrates a sample federation of XMPP servers and clients as well as the direct communication via addresses between clients. For example if the client ***alice*** connected to the server ***xmppserverone.com*** wants to send a message to the client ***bob*** also connected to the server ***xmppserverone.com***, ***alice*** addresses the message with ***bob@xmppserverone.com***. In case of clients within the federation, ***alice*** addresses messages respectively with ***carol@xmppservertwo.com*** and ***dave@xmppservertwo.com*** for the clients connected to the server ***xmppservertwo.com***.

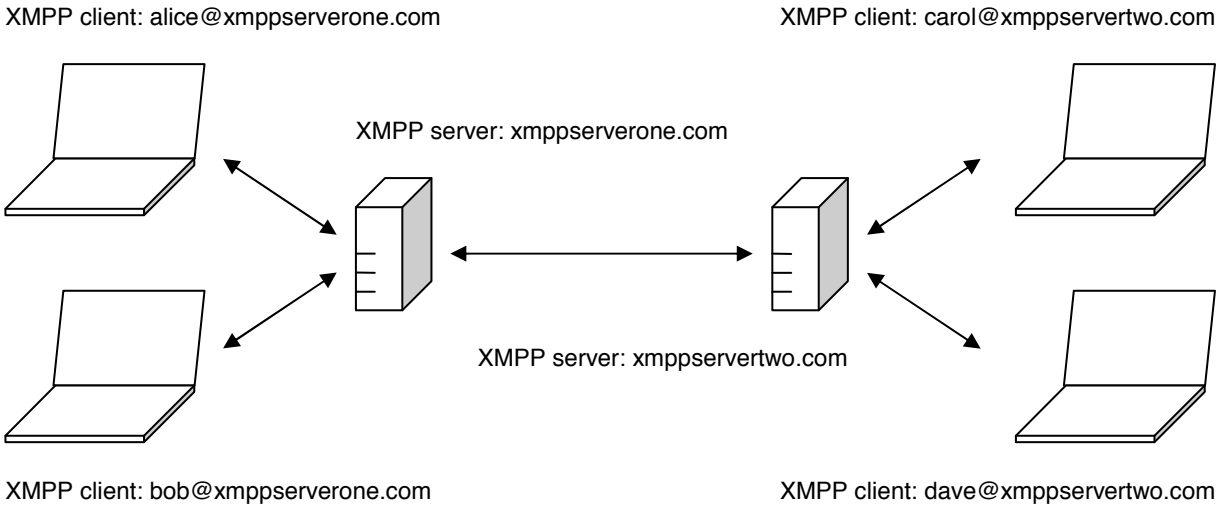

Figure 2.1: Federation of XMMP servers and clients.

However, the most suitable definition for XMPP may be ***hybrid decentralized unstructured P2P technology*** analogous to the following definition given in [19]: *"Each client computer stores content (files) shared with the rest of the network. All clients connect to a central directory server that maintains:*

– A table of registered user connection information (IP address, connection bandwidth etc.)

– A table listing the files that each user holds and shares in the network, along with metadata descriptions of the files (e.g. filename, time of creation, etc.)" [19]

According to the work in [19] computers willing to join a hybrid decentralized unstructured P2P network have to contact the central server first and communicate the files they store. Computers, who already are members of such a network and intending to search for files have to send requests to the server. The server will comb through its indexes and return a list of peers in possession of the matching files. To download, the requested files computers have to open a direct connection to the

peers and download the files. The authors of [19] mention that hybrid decentralized unstructured P2P systems are simple to implement and provide quick search but are vulnerable to censorship, legal action surveillance, malicious attack, and technical failure because of the central server. In the opinion of the authors another disadvantage of these systems is the inability to scale due to limited capacities of the server. However, the XMPP communication platform utilized in the HERMES framework allows servers to be aggregated into federations eliminating this disadvantage. In contrast to the definition in [19] the XMPP servers don't index the data provided by the clients but route search requests and responses between clients instead. Regardless of the exact taxonomy, XMPP proved to be a stable and reliable technology for inter-application communication between HERMES-based peers, easy to set up and use,

2.5 Data Replication

The previous sub-chapter presented technologies for the internal communication between numerous HERMES-based applications, which replicate the knowledge collected by HERMES instances in distributed heterogeneous networks. This sub-chapter focuses on the actual data replication necessary to enable HERMES to operate in distributed environments. Although the final technology chosen for the internal communication between HERMES-based applications is XMPP and deviates from typical P2P technologies, it can be said that due to the distributed nature of HERMES-based applications, XMPP, in general, makes similar demands on data replication as P2P technologies in this particular case. As mentioned before, another special feature of XMPP is the fact that any amount of servers can be merged into federations, which is very uncommon for typical pure client-server architectures. Furthermore, the last chapter has shown that XMPP may also be described as hybrid decentralized unstructured P2P technology. This sub-chapter will therefore focus on the details of data replication in P2P systems and apply the collected findings to XMPP.

The work in [23] gives an overview of the most important ideas in the context of data replication in P2P systems. The authors state that P2P systems rely on using data replication because data storage and processing are distributed and high data availability has to be provided despite peers that can dynamically join and leave the network at any time. The work mentions that P2P systems dealing with semantically rich data and advanced query languages require more sophisticated replication approaches, such as multi-master replication and suitable conflict resolutions. According to [23] these issues are addressed by optimistic replication, whereby replicas are updated asynchronously and applications can continue to operate even when some peers fail to respond properly or simply disconnect. In these terms, HERMES can be described as an advanced framework dealing with rich data and requiring capabilities such as multi-master replication and update conflict resolution. The

work in [23] also presents a classification of replica control mechanisms – place and time of updates – and gives definitions of the basic replication concepts: single-master and multi-master replication, full replication and partial replication as well as synchronous versus asynchronous propagation.

The work in [23] defines replicas as follows: *"A replica of an object can be classified as primary copy or secondary copy according to its updating capabilities. A primary copy accepts read and write operations and is held by a master site. A secondary copy accepts only read operations and is held by a slave site."*

The work in [23] defines single-master replication and multi-master-replication as follows: *"In the single-master approach, there is only a single primary copy for each replicated object. In this case, every update is first applied to the primary copy at the master site, and then it is propagated towards the secondary copies held by the slave sites. Due to the interaction between master and slave sites, this approach is also known as master/slave replication. Centralizing updates at a single copy avoids concurrent updates on different sites, thereby simplifying the concurrency control. In addition, it assures that one site has the up-to-date values for an object. However, this centralization introduces a potential bottleneck and a single point of failure. Therefore, a failure in a master site blocks update operations, and thus limits data availability … In the multi-master approach, multiple sites hold primary copies of the same object. All these copies can be concurrently updated, wherefrom the multi-master technique is also known as update anywhere. Distributing updates avoids bottlenecks and single points of failures, thereby improving data availability. However, in order to assure data consistency, the concurrent updates to different copies must be coordinated or a reconciliation algorithm must be applied to solve replica divergences. On the one hand, coordinating distributed updates can lead to expensive communication, and on the other hand reconciliation solutions can be complex."*

The work in [23] also defines full replication and partial replication – the two basic approaches for replica placement: *"Full replication consists of storing a copy of every shared object at all participating sites. This approach provides simple load balancing since all sites have the same capacities, and maximal availability as any site can replace any other site in case of failure. With partial replication, each site holds a copy of a subset of shared objects, so that the objects replicated at one site may be different of the objects replicated at another site … This approach expends less storage space and reduces the number of messages needed to update replicas since updates are only propagated towards the affected sites … Thus, updates produce reduced load for the network and sites. However, if related objects are stored at different sites, the propagation protocol becomes more*

complex as the replica placement must be taken into account. In addition, this approach limits load balance possibilities since certain sites are not able to execute a particular set of transactions."

And finally the work in [23] differentiates between synchronous and asynchronous update propagation: *"The synchronous update propagation approaches (a.k.a. eager) apply the changes to all replicas within the context of the transaction that initiates the updates … As a result, when the transaction commits, all replicas have the same state. This is achieved by using concurrency control mechanisms like two-phase-locking … or timestamp based algorithms … The asynchronous update propagation approaches (a.k.a. lazy) do not change all replicas within the context of the transaction that initiates the updates. Indeed, the initial transaction commits as soon as possible, and afterwards the updates are propagated to all replicas … An advantage of the asynchronous propagation is that the update does not block due to unavailable replicas, which improves data availability. In addition, communication is not needed to coordinate concurrent updates, thereby reducing the transaction response times and improving the system scalability."*

Additionally, the authors of [23] divide asynchronous replication in optimistic and non-optimistic replication: *"In general, optimistic asynchronous replication relies on the optimistic assumption that conflicting updates will occur only rarely, if at all. Updates are therefore propagated in the background, and occasional conflicts are fixed after they happen. In contrast, non-optimistic asynchronous replication assumes that update conflicts are likely to occur and implements propagation mechanisms that prevent conflicting updates"*

HERMES applies the fusion of multi-master replication, full replica placement and non-optimistic asynchronous update propagation. All participating peers store full prime copies of the same data and all these copies can be updated simultaneously. While this particular blend of techniques and strategies exhibits many advantages, there are of course some shortcomings. On the one hand, multi-master replication improves data availability simultaneously, avoiding bottlenecks and single points of failure. Since data is fully stored at different locations, the replication protocol usually remains simple as the replication placement can be ignored. The flexible asynchronous replication strategy enables applications to autonomously choose the proper time to propagate updates and consequently to function within over dynamic heterogeneous networks, from which peers can appear and disappear at any time. Moreover, lazy replication does not block due to unavailable replicas and also does not require any extra communication in order to coordinate concurrent updates improving the scalability and response times. All these requirements are crucial for the data distribution and replication within a distributed integrated communication management solution like HERMES. On the other hand, multi-

master replication may require applications to coordinate the update process of diverse copies or to apply a complex reconciliation algorithm. Because updates are spread to all peers, full replication has the disadvantage of occupying more storage space and a higher number of messages to update replicas resulting in higher network load. The asynchronous approach implicates possible divergences amongst replicas. This means that replicas may diverge at different locations and local reads cannot promise to return the most recent values. However, it is the belief of this thesis that full lazy multi-master replication is the best choice in terms of simplicity, efficiency and quickly achievable results for data replication in distributed HERMES-based applications.

As stated before, the full lazy multi-master replication presents several disadvantages, such as the need for a moderately complex reconciliation algorithm to solve replica discrepancies when simultaneously updating different copies. On the contrary, this thesis required a simple solution for this particular problem. Thus, from the very beginning, the idea to develop a reconciliation algorithm integrating logging and timestamping was pursued. Scientific research on this specific topic has been already undertaken, for example in [24]. The work states that concurrent updates result in replica divergence and conflicts that require reconciliation. Furthermore, [24] states that most solutions are not well tuned for peers that may join or leave the network at any time. The authors of [24] propose a solution providing P2P logging and timestamping for P2P reconciliation over a distributed hash table, where every update is timestamped at every peer, stored in a P2P log and can be retrieved during reconciliation to enforce consistency. This approach is, in fact, very similar to what was intended and finally implemented for the HERMES framework. Although P2P logging and timestamping sounds straightforward, the work in [24] presents a rather complex solution due to the difficulties of P2P systems and especially the use of a distributed hash table. Fortunately, the HERMES framework relies on the XMPP communication platform instead of typical P2P technology and consequently does not have to deal with difficulties like distributed hash tables. This is a considerable benefit when designing an appropriate reconciliation algorithm, a task that proved to be quite simple: The HERMES framework provides a service that is responsible for persisting data in an embedded relational database at each peer. In order to enable data replication, this service keeps an operation log. Whenever data is stored the framework logs this operation. Each log entry consists of an identification number, a timestamp, a reference to the affected data and the performed operation type: ***create***, ***update*** or ***delete***. Once an associated peer triggers the replication process, a log entry with a specific identification number, which has to be memorized and incremented, is requested. In case there are no entries with this particular number, replication does not take place and the log entry including the actual updated data is returned to the requesting peer. This peer then has to compare the timestamp of the received log entry and the timestamp of the same local data. If the received data contains a more recent timestamp

then it has to be replicated, because the responder made the most recent amendment. Otherwise it will be skipped. One important rule applying to this algorithm is that whenever data is deleted, this operation is final, which means that even local data with a more recent timestamp will be overridden and deleted by received data that has been deleted on any other associated peer. Since the fundamental aspect of this idea is timestamping, the whole peer network has to be synchronized in time to facilitate proper operation. The HERMES framework accomplishes this by using time synchronization, utilizing NTP.

3 Architecture and Implementation

This chapter addresses the architecture and the reference implementation of the HERMES framework. Firstly, the framework is addressed as a whole, whereby the thesis focuses on providing general understanding for the framework's design from an abstract top-down-oriented point of view. Subsequently, the sub-chapters cover the framework's details, more precisely its components and services and will provide a detailed illustration of the framework's supplementary cornerstones namely contacts, database and database replication. The framework's ability to replicate its knowledge over distributed HERMES instances in heterogeneous networks is identified as a further major contribution of this thesis besides the framework itself and therefore receives an in-depth treatment. All sub-chapters examine their respective topics from an abstract and more comprehensible point of view as well as from a technical, program-related standpoint, where a summary of the framework API specification together with Java packages, interfaces, classes and methods is offered. Conclusively, the next chapter will present and discuss how to utilize the framework in order to build applications upon on the basis of an implemented sample application.

3.1 Overview

The HERMES framework is designed in compliance with the service-oriented architecture and built on top of the Java-based OSGi service platform. The keystone behind this architectural decision was to maximize the extensibility and modularity of the anticipated integrated communication management solution. As illustrated in the previous chapter, the OSGi framework is basically an execution environment for applications and services. Service functions are described via Java interfaces. So-called bundles – actual jar archive files with OSGi extended manifest information – contain packaged service specifications and implementations.

The entire HERMES framework is specified as services described via Java interfaces. The framework's reference implementation was realized in the single OSGi bundle ***at.jku.tk.hermes***. All components and services were packed into this bundle. In order to run the framework, an implementation of the OSGi R4 core framework specification is required. In such an implementation, the HERMES framework bundle has to be installed and activated. Figure 3.1 illustrates a typical OSGi-compliant software architecture, whereby the layers above the ***Hardware/OS*** tier represent an OSGi framework implementation and the area labeled ***Bundles*** is where the HERMES framework bundle and any other custom bundles have to be placed. At time of development of the framework, various open source implementations of the OSGi R4 core framework specification were available, for example Apache Felix or Eclipse Equinox.

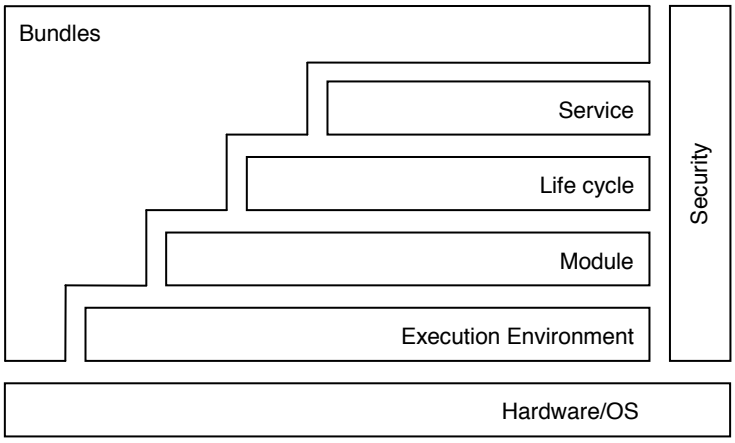

Figure 3.1: OSGi-compliant software architecture [25].

As stated in the introductory chapter and shown in Figure 1.3, the HERMES framework was split into numerous components during the design phase. Each of these components was expected to accomplish particular limited tasks, roughly fulfilling the divide-and-conquer paradigm. The framework consists of key components: sensors, events, rule bases, inputs, outputs, actions, protocols and tools. Clearly these units are sufficient to provide the necessary functionality for communication management. Nevertheless, in order to operate smoothly, the framework requires the ability to learn and to store its knowledge as data. In order to collect and preserve data in a distributed manner, the framework was equipped with an embedded database which can be replicated over distributed HERMES instances. Finally, to route communication-related actions to proper recipients, contact information is required. Contact information or contacts generally are treated by the framework as specialized data. Thus, contact management was provided by a set of interfaces and contacts, which are persisted in the framework's embedded database.

The breakdown of the HERMES framework into numerous components reveals its predetermined organization, which also seems eligible for the framework API. Therefore and due to the fact that most of the framework components are represented by multiple proxies, the presented components have been chosen to picture and form the Java packages of the framework API:

at.jku.tk.hermes.action

This package contains classes representing a set of predefined communication-related actions.

at.jku.tk.hermes.contact

This package encompasses classes representing contact information and services aimed to support contact management.

at.jku.tk.hermes.core

This package consists of classes and services indispensable for the very basic operability of the framework. These are classes required for metadata retrieval and for the internal event-based communication as well as services aimed to support database management and access to system preferences.

The framework's reference implementation additionally includes classes implementing the aforementioned services as well as the implementation of the framework's database replication mechanism.

In order to distinguish between the core components and the reference implementation's internals, this package has been amended by the sub-packages ***at.jku.tk.hermes.core.internal*** and ***at.jku.tk.hermes.core.internal.replication***.

at.jku.tk.hermes.io

This package holds classes representing a set of predefined communication-related inputs and outputs as well as corresponding support services

In order to distinguish between inputs, outputs and supplementary classes this package has been amended by the sub-packages ***at.jku.tk.hermes.io.input*** and ***at.jku.tk.hermes.io.output***.

at.jku.tk.hermes.protocol

This package includes classes representing a set of predefined communication-related protocols.

at.jku.tk.hermes.rulebase

This package contains a service specification aimed to be implemented by rule bases developed by third parties.

at.jku.tk.hermes.sensor

This package consists of a service specification aimed to be implemented by sensors developed by third parties as well as supplementary sensor-related classes.

at.jku.tk.hermes.tool

This package incorporates a service specification aimed to be implemented by tools developed by third parties as well as supplementary tool-related classes.

The presented breakdown was chosen in order to best support the complex task of integrated communication management, by distributing small task fragments to capillary components. These distinct components are able to carry out specific task fragments and in the first instance provide great modularity and flexibility, complying with the propagated concept of SOA. Together, they manage to complete more sophisticated communication management tasks than could have been expected from an integrated communication management framework. It is the opinion of this thesis that the most

efficient way to support communication management in a distributed and adaptive manner are the presented architectural considerations.

3.2 Sensors

The role of sensors has already been emphasized by several related works and, as noted before, they are a fundamental part of the HERMES framework. The framework's sensors are deemed services in terms of OSGi and have been included in order to collect data required for the framework's operation. These specialized data mining components are responsible for gathering data that is of interest for any communication management-related task. Such data could be appointments in calendars, communication tool usage statistics, patterns in communication behavior or even trivial pieces of information like the current date and time.

Sensors generally represent a bundle operating promptly within the HERMES framework or more precisely a JVM. However, this is not a mandatory limitation. Sensors may also be simple bundles within a JVM wrapping, further external sensors or services. An example would be a sensor consuming and processing calendar data from an external web service feed like Google Calendar.

Once sensors pick up communication management-relevant data, they usually are obliged to inform the rest of the framework about this. To accomplish this task, sensors utilize the OSGi Event Admin system service [18]. This service is a publish/subscribe-based inter-bundle communication mechanism allowing sensors to publish any type of data to the entire framework in the shape of events consisting of an event topic and event properties. Components interested in handling these events have to subscribe to the relevant event topic and may also define further fine-grained filters on top of that. Depending on the implementation, sensors will publish events either autonomously or require triggering by other components to do so. An example for an autonomously publishing sensor would be a timestamp beacon broadcasting the system's current timestamp every second.

The HERMES framework sets very few limits to the actual implementation of sensors. As a matter of fact, these components can be almost entirely shaped by their developer. Sensors have been designed as OSGi services so that they may be developed and customized by third parties to not only allow hot swapping during runtime, but also to emphasize the flexibility of the entire framework. The framework exposes sensors in the shape of the ***SensorService*** Java interface. Classes implementing this interface must be also registered as a ***SensorService*** service along with the OSGi service registry.

The HERMES framework specifies the ***SensorService*** Java interface API as follows:

package at.jku.tk.hermes.sensor

public interface SensorService

A ***SensorService*** instance allows the gathering of communication-related data and publishing this data in the shape of events to the entire framework by leveraging the OSGi Event Admin system service.

public MetaData getMetaData()

Returns the metadata characterizing this ***SensorService*** instance.

public boolean supportsEventTopic(String topic)

Returns ***true*** when this ***SensorService*** instance publishes events containing the specified event topic or ***false*** otherwise. The topic is specified by the parameter ***topic***.

public java.util.List<String> getSupportedEventTopics()

Returns a list of strings representing the event topics enclosed in events published by this ***SensorService*** instance.

public void executePublishing()

In case of a ***SensorService*** implementation that publishes events only on demand, this method triggers such an event publication.

As mentioned earlier, events published by sensors consist of an event topic and event properties. The event properties provide information about the actual event that occurred and are specified by the OSGi Event Admin system service [18]. The specification enumerates several standard property names for event properties that are additionally wrapped by a HERMES framework utility. The ***EventProperties*** Java class offers a quick and convenient way to create and set event properties.

The HERMES framework specifies the ***EventProperties*** Java class API as follows:

package at.jku.tk.hermes.core

public class EventProperties

An ***EventProperties*** instance allows convenient management of OSGi Event Admin system service standard event properties which are sent along with events published by a ***SensorService*** instance.

public EventProperties(String bundleSymbolicName)

Creates a new ***EventProperties*** instance with the specified bundle symbolic name of the bundle relevant to the event. The name is specified by the parameter ***bundleSymbolicName***.

public java.util.Properties toProperties()

Returns all this instance's properties in the shape of a ***java.util.Properties*** object. Such an object is required to properly instantiate an event.

public void setBundleSigner(String bundleSigner)

Sets the distinguished name of the bundle relevant to the event. The name is specified by the parameter ***bundleSigner***.

public void setBundleSymbolicname(String bundleSymbolicname)

Sets the bundle symbolic name of the bundle relevant to the event. The name is specified by the parameter ***bundleSymbolicname***.

public void setEvent(Object event)

Sets the actual event. The event is specified by the parameter ***event***.

public void setException(Throwable exception)

Sets the exception of the event. The exception is specified by the parameter ***exception***.

public void setExceptionClass(String exceptionClass)

Sets the exception class of the event. The class is specified by the parameter ***exceptionClass***.

public void setExceptionMessage(String exceptionMessage)

Sets the exception message of the event. The message is specified by the parameter ***exceptionMessage***.

public void setMessage(String message)

Sets the human-readable message of the event. The message is specified by the parameter ***message***.

public void setService(org.osgi.framework.ServiceReference service)

Sets the service reference of the service relevant to the event. The reference is specified by the parameter ***service***.

public void setServiceId(Long serviceId)

Sets the service ID of the service relevant to the event. The ID is specified by the parameter ***serviceId***.

public void setServiceObjectclass(String[] serviceObjectClass)

Sets the object classes of the service relevant to the event. The classes are specified by the parameter ***serviceObjectClass***.

public void setServicePid(String[] servicePid)

Sets the sustained identities of the service relevant to the event. The identities are specified by the parameter ***servicePid***.

public void setTimestamp(Long timestamp)

Sets the timestamp when the event occurred. The timestamp is specified by the parameter ***timestamp***.

Another essential part of any sensor and of most of the framework's components is metadata. As already mentioned, metadata provides a way of characterizing many of the HERMES framework's components. The attributes constituting such a metadata unit are the component's name and description since these two are the lowest common denominators.

The HERMES framework specifies the ***MetaData*** Java class API as follows:

package at.jku.tk.hermes.core

public final class MetaData

A ***MetaData*** instance allows obtaining of the metadata ***name*** and ***description*** for a component it is characterizing.

public String getName()

Returns the name of the component this ***MetaData*** instance is characterizing.

public void setName(String name)

Sets the specified name for the component this ***MetaData*** instance is characterizing. The name is specified by the parameter ***name***.

public String getDescription()

Returns the description of the component this ***MetaData*** instance is characterizing.

public void setDescription(String description)

Sets the specified description for the component this ***MetaData*** instance is characterizing. The description is specified by the parameter ***description***.

3.3 Rule Bases

Whenever events are published by sensors, they can be picked up by any interested constituent using the OSGi's own Event Admin system service, which routes these events to interested recipients. The HERMES framework provides special rule base components for this particular task. The framework's rule bases are considered as services in terms of OSGi and may be regarded as the brains of a HERMES-based application. Rule bases represent the key idea of an integrated communication management system, namely the system's ability to learn from previous experience in the domain of communications. Once implemented and properly set up, rule bases intercept events published by sensors, analyze the data contained in these events and take appropriate communication-related actions

based on the analyzed data and the previously collected knowledge about users' communication needs and habits.

Similar to sensors, the HERMES framework also sets very few limits to the actual implementation of rule bases. In actual fact, it is the developer's decision what a rule base will eventually be able to accomplish. The general idea is to derive knowledge about users' communication needs and study behavior patterns by employing these components. Actual applications based on the HERMES framework have to envelop highly advanced rule bases, if they wish to fully utilize the framework's capabilities and provide added value to users' communication experience. Such a rule bases could, for example, employ learning from experience by making use of specialized learning algorithms or advanced technologies like artificial intelligence. However, as previously mentioned, the HERMES framework is just the imperative foundation for such components and exploring the described advanced techniques and technologies to fully exploit the framework's potential is an overwhelming task, beyond the scope of this thesis.

Supposing a HERMES-based application would be equipped with an advanced rule base that actively monitors users' communication activities, it appears to be safe to assume that essentially every rule base has to interact with its environment somehow. For this purpose, the HERMES framework offers the possibility to send communication-related actions to tools and to input or output data. The framework includes a set of predefined actions, inputs and outputs that can be used exactly for this purpose.

Rule bases have been designed as OSGi services because, analogous to sensors, they are supposed to be developed and customized by third parties. The framework exposes rule bases in the shape of the ***RuleBaseService*** Java interface. Classes implementing this interface must also be registered as a ***RuleBaseService*** service along with the OSGi service registry.

The HERMES framework specifies the ***RuleBaseService*** Java interface API as follows:

package at.jku.tk.hermes.rulebase

public interface RuleBaseService extends org.osgi.service.event.EventHandler

A ***RuleBaseService*** instance allows intercepting events published by sensors by leveraging the OSGi Event Admin system service, analyzing the events and taking appropriate communication-related actions regarding users' communication needs.

public MetaData getMetaData()

Returns the metadata characterizing this ***RuleBaseService*** instance.

public boolean supportsEventTopic(String topic)

Returns ***true*** when this ***RuleBaseService*** instance subscribes events containing the specified event topic or ***false*** otherwise. The event topic is specified by the parameter ***topic***.

public java.util.List<String> getSupportedEventTopics()

Returns a list of strings representing the event topics subscribed by this ***RuleBaseService*** instance.

3.4 Inputs and Outputs

In the previous chapters, rule bases have been described as magic black boxes handling communication-related events from the environment by applying rules derived from the system's knowledge and interacting with further components. Inputs, outputs and their corresponding services constitute such components.

Inputs and outputs can be described as wrappers or carriers in the shape of Java objects. They do not only specify actual inputs or outputs by type but also carry the actual input or output value and additional metadata. The presented input and output functionality is but a generic scope provided by the HERMES framework. The framework offers numerous predefined inputs and outputs to use out of the box and can be enhanced with new ones on demand. The framework exposes inputs and outputs in the shape of the ***Input*** and the ***Output*** Java interfaces.

The HERMES framework specifies the ***Input*** Java interface API as follows:

package at.jku.tk.hermes.io.input

public interface Input<T>

An ***Input*** instance represents a wrapper for an input value. The actual type of the input is set on instantiation specified by the ***T*** Java generic type.

public MetaData getMetaData()

Returns the metadata characterizing this ***Input*** instance.

public void setMetaData(MetaData metaData)

Sets the metadata characterizing this ***Input*** instance. The metadata is specified by the parameter ***metaData***.

public T getInput()

Returns the actual input value of this ***Input*** instance.

public void setInput(T input)

Sets the actual input value of this ***Input*** instance. The value is specified by the parameter ***input***.

The HERMES framework specifies the ***Output*** Java interface API as follows:

package at.jku.tk.hermes.io.output

public interface Output<T>

An ***Output*** instance represents a wrapper for output values. The actual type of the output is set on instantiation specified by the ***T*** Java generic type.

public MetaData getMetaData()

Returns the metadata characterizing this ***Output*** instance.

public void setMetaData(MetaData metaData)

Sets the metadata characterizing this ***Output*** instance. The metadata is specified by the parameter ***metaData***.

public T getOutput()

Returns the actual output value of this ***Output*** instance.

public void setOutput(T output)

Sets the actual output value of this ***Output*** instance. The value is specified by the parameter ***output***.

The HERMES framework specifies a number of predefined input and output implementations for basic Java data types to use out of the box:

Predefined ***Input*** implementations:

BooleanInput, ***ByteInput***, ***CharacterInput***, ***DoubleInput***, ***FloatInput***, ***IntegerInput***, ***LongInput***, ***ShortInput***, ***StringInput***.

Predefined ***Output*** implementations:

BooleanOutput, ***ByteOutput***, ***CharacterOutput***, ***DoubleOutput***, ***FloatOutput***, ***IntegerOutput***, ***LongOutput***, ***ShortOutput***, ***StringOutput***.

The framework defines that instances of inputs or outputs have to be passed to IO services. These services are responsible for executing the actual input or output and also returning results. Moreover they are designed to use any IO method and any IO device respectively such as text, speech, gesture, keyboard, display, microphone or speaker. Using this approach, the framework is not only able to support input and output via simple graphical user interfaces but also via speech and even tactile interfaces.

The IO services are considered as services in terms of OSGi and were introduced in order to provide framework-wide accessibility to inputs and outputs through common interfaces. In addition, IO services facilitate hot-swapping during runtime by design and are aimed to be customized by third

party implementations. Again, the HERMES framework sets few limits for the actual implementation, so these components may be fashioned by the developer as needed. The framework exposes IO services in the shape of the ***IOService*** Java interface. Classes implementing this interface must also be registered as an ***IOService*** service along with the OSGi service registry.

The HERMES framework specifies the ***IOService*** Java interface API as follows:

package at.jku.tk.hermes.io

public interface IOService

An ***IOService*** instance allows executing actual inputs and outputs independent of the IO method and the IO device and returning results.

public MetaData getMetaData()

Returns the metadata characterizing this ***IOService*** instance.

public boolean supportsInput(Class<? extends Input<?>> clazz)

Returns ***true*** when this ***IOService*** instance supports executing inputs for the specified Java class extending ***Input*** or ***false*** otherwise. The class is specified by the parameter ***clazz***.

public List<Class<? extends Input<?>>> getSupportedInputs()

Returns a list of Java classes extending ***Input*** and representing the inputs supported in execution by this ***IOService*** instance.

public void executeInput(Input<?> input, IOCallback callback)
throws InputNotSupportedException

Executes the specified input and returns the result to the specified callback or throws an exception. The input is specified by the parameter ***input***, the callback is specified by the parameter ***callback***.

public boolean supportsOutput(Class<? extends Output<?>> clazz)

Returns ***true*** when this ***IOService*** instance supports executing outputs for the specified Java class extending ***Output*** or ***false*** otherwise. The class is specified by the parameter ***clazz***.

public List<Class<? extends Output<?>>> getSupportedOutputs()

Returns a list of Java classes extending ***Output*** and representing the outputs supported in execution by this ***IOService*** instance.

public void executeOutput(Output<?> output, IOCallback callback)
throws OutputNotSupportedException

Executes the specified output and returns the result to the specified callback or throws an exception. The output is specified by the parameter ***output***, the callback is specified by the parameter ***callback***.

The ***IOService*** API specification additionally refers to the following Java exception classes, which are also part of the HERMES framework specification:

InputNotSupportedException

An ***InputNotSupportedException*** instance is thrown when commanding an ***IOService*** to execute an input with an unsupported input Java class.

OutputNotSupportedException

An ***OutputNotSupportedException*** instance is thrown when commanding an ***IOService*** to execute an output with an unsupported output Java class.

In addition, the HERMES framework specifies the following Java exception classes, which should be utilized by ***IOService*** implementations:

InputExecutionFailedException

An ***InputExecutionFailedException*** instance is thrown when an ***IOService*** failed to execute an input.

OutputExecutionFailedException

An ***OutputExecutionFailedException*** instance is thrown when an ***IOService*** failed to execute an output.

IO services are obliged to collect inputs described by input Java objects they receive during operation. The method of collecting inputs depends on the actual implementation of the service and is the responsibility of the developer. A simple IO service, for example, using a command line interface and assigned to collect a character input would have to ask the user to input a character in the command line. Upon user interaction, the service would return the entered character to a callback provided by the inquirer. In this scenario, the input intent is an input Java object asking for character input, carrying a character placeholder and potential metadata passed from the inquirer to the IO service. A more sophisticated version of this service would be the use of a speech interface that would ask the user to verbalize the mandatory character instead.

In contrast to executing inputs, IO services also support outputs. They are responsible for signaling certain outputs to the environment. The outputs are described by output Java objects the services obtain during operation. The signaling of outputs again depends on the actual implementation of the service and is the responsibility of the developer. For example, a simple IO service using a command line interface and assigned to signal a character output would output the character to the command line. The service would not expect the user to interact in any way. Furthermore, the service would immediately notify the callback provided by the inquirer about the output's success. In this scenario,

the output intent is an output Java object asking for character output, carrying a character to output and potential metadata passed from the inquirer to the IO service. A more sophisticated version of this service would involve the use of a speech interface that would verbalize the character instead.

The term ***callback*** has been mentioned numerous times in conjunction with IO services. IO callbacks have been introduced to separate the execution of inputs and outputs from the rest of the framework. The reason for this decision can be found in the unpredictable amount of time consumed by inputs or outputs, eventually blocking the framework's main Java thread. In order to preventively eliminate such problems, inputs and outputs have been supplemented with callbacks. By passing callbacks from inquirers to IO services the actual execution of inputs and outputs is transferred to a separate Java thread running asynchronously, so the framework's main Java thread will not be blocked. The executions' results are then relayed by IO callbacks from the IO services back to the inquirers. The framework exposes IO callbacks in the shape of the ***IOCallback*** Java interface.

The HERMES framework specifies the ***IOCallback*** Java interface API as follows:

package at.jku.tk.hermes.io

public interface IOCallback

An ***IOCallback*** instance allows relaying input and output results from an asynchronously running IO service back to the actual inquirer.

public void onInput(Input<?> input)

This method is called by IO services when intending to return the result of a successful input execution to the inquirer providing this ***IOCallback*** instance. The input is specified by the parameter ***input***.

public void onInputExecutionFailed()

This method is called by IO services when intending to notify the inquirer providing this ***IOCallback*** instance about a failed input execution.

public void onOutput()

This method is called by IO services when intending to notify the inquirer providing this ***IOCallback*** instance about a successful output execution.

public void onOutputExecutionFailed()

This method is called by IO services when intending to notify the inquirer providing this ***IOCallback*** instance about a failed output execution.

The IO services were designed in the presented manner with the ulterior motive to maximize flexibility when developing these kinds of services for HERMES-based applications and make the utilization of any imaginable IO method or IO device possible.

3.5 Actions

Beyond inputs and outputs, actions present yet another way for rule bases to interact with other components. Actions can be described as communication-related intents usually passed from rule bases to tools and executed in existing communication tools. An action may intend to send a text message to a contact, to video-call a contact or to transfer a file to a group of contacts. These are just a few examples of conceivable communication-related actions that constitute a ***must-have*** for any integrated communication management framework. The HERMES framework has been equipped with a series of predefined actions out of the box that can be extended on demand. The framework exposes actions in the shape of the ***Action*** Java interface.

The HERMES framework specifies the ***Action*** Java interface API as follows:

package at.jku.tk.hermes.action

public interface Action

An ***Action*** instance represents a wrapper for a communication-related action.

The HERMES framework specifies a number of predefined communication-related action implementations to use out of the box:

AudioCallStartAction, ***AudioConferenceStartAction***, ***AudioVideoCallStartAction***, ***AudioVideoConferenceStartAction***, ***EmailSendAction***, ***FileTransferToManyAction***, ***FileTransferToOneAction***, ***MessageSendToManyAction***, ***MessageSendToOneAction***, ***PresenceSetAction***.

Actions are usually required to carry contact information of the receivers along with additional data such as the message content. However, some actions may exist without any further data except the contact information. These include actions initiating a call, generally only requiring the contact information to dial but no further data. Yet, some actions may exist without any additional information at all, not even contact information, which include actions changing the presence status of a communication tool, generally only requiring the presence type represented by the action itself but no further information.

It is obvious that not every possibly existing communication-related action can be handled by every existing communication tool. In order to resolve this divergence, the HERMES framework enables tools to reject unsupported actions with proper Java exceptions and provides a supplementary mechanism called protocols. Protocols can be identified as the common ground for actions, contact information and tools. Actually, it is the link that makes these three components work together smoothly. The following scenario should aid understanding of the concept of protocols: Once a rule base deposits an action, the framework will route this action to a communication tool. This action, for example, intends to call a contact and therefore has to piggyback the actual contact to call. As stated before, the precondition here is that the tool has to support this particular action in order to execute it. Moreover, the tool has to retrieve the proper contact address to use from the piggybacked contact. It is clearly evident that in an integrated communication management system each contact will usually possess more than one contact address. Given this circumstance, every tool in a HERMES-based application would require each contact to provide a contact address for this particular tool. Obviously this would lead to very inefficient systems, since the amount of tools is never final and some tools may also refer to the same contact address, offering space for optimization. Likewise, contacts cannot be expected to provide a contact address for every possibly existing communication tool. In order to solve these problems, the framework was equipped with protocols. With this addition, contacts generally provide a contact address for a particular protocol. Tools, on the other hand, may support multiple protocols and thus might be able to retrieve the proper contact address for the desired protocol. Similarly, actions cannot be executed by tools if the desired protocol is not supported by the contact and/or the tool. The HERMES framework contains a number of predefined protocols that can be extended when necessary.

The HERMES framework specifies the ***Protocol*** Java interface API as follows:

package at.jku.tk.hermes.protocol

public interface Protocol

A ***Protocol*** instance represents a wrapper for an existing communication protocol and enables smooth integration of communication-related actions, contact information and communication tools within the HERMES framework.

public boolean supportsAction(Class<? extends Action> clazz)

Returns ***true*** when this ***Protocol*** instance supports execution of the specified Java class extending ***Action*** or ***false*** otherwise. The class is specified by the parameter ***clazz***.

public List<Class<? extends Action>> getSupportedActions()

Returns a list of Java classes extending ***Action*** representing the actions supported in execution by this ***Protocol*** instance.

The HERMES framework specifies a number of predefined protocol implementations to use out of the box:

EmailClientProtocol, ***EmailSmtpProtocol***, ***PhoneFixedProtocol***, ***PhoneMobileProtocol***, ***SkypeProtocol***, ***SkypePstnProtocol***, ***SMSGatewayProtocol***.

3.6 Tools

Tools, or more specifically, communication tools denote a further component of the HERMES framework. The framework's tools are considered as services in terms of OSGi and have been introduced for the following reasons. On the one hand, tools stand for further enforcement of the SOA concept throughout the entire HERMES framework. The role intended for these components is, first and foremost, the execution of communication-related actions. On the other hand, the design of an integrated communication management framework requires such a framework to be inter-operable with existing communication tools. In short, tools are responsible for executing received communication-related actions but may also be responsible for forwarding these actions to existing communication tools and interacting with these.

Tools usually represent a bundle operating promptly within the HERMES framework or more precisely the JVM. However, similar to sensors this is not a mandatory limitation. Tools may also just be simple bundles within the JVM wrapping, further external tools or services. An example would be a tool that receives actions but forwards them to the Skype communication tool and returns the actions' execution status to the inquirer.

Tools share the following characteristic with sensors, rule bases and IO services: HERMES-based applications operate with awareness of users' communication needs and present appropriate tool support with great flexibility and modularity. Sensors, rule bases, IO services and tools are the key components of every HERMES-based application. For all these components the HERMES framework provides rudimentary interfaces, thus setting few limits to the actual implementation. These components may be fashioned very freely providing a large scope for the developer, not to mention that these services may be hot-swapped during runtime due to the elastic nature of the underlying Java-based OSGi service platform. The framework exposes tools in the shape of the ***ToolService*** Java interface. Classes implementing this interface must be also registered as a ***ToolService*** service along with the OSGi service registry.

The HERMES framework specifies the ***ToolService*** Java interface API as follows:

package at.jku.tk.hermes.tool

public interface ToolService

A ***ToolService*** instance allows executing communication-related actions and inter-operates with existing communication tools.

public MetaData getMetaData()

Returns the metadata characterizing this ***ToolService*** instance.

public boolean supportsProtocol(Class<? extends Protocol> clazz)

Returns ***true*** when this ***ToolService*** instance supports executing actions for the specified Java class extending ***Protocol*** or ***false*** otherwise. The class is specified by the parameter ***clazz***.

public List<Class<? extends Protocol>> getSupportedProtocols()

Returns a list of Java classes extending ***Protocol*** and representing the protocols, whose actions are supported in execution by this ***ToolService*** instance.

public boolean supportsAction(Class<? extends Action> clazz)

Returns ***true*** when this ***ToolService*** instance supports executing actions for the specified Java class extending ***Action*** or ***false*** otherwise. The class is specified by the parameter ***clazz***.

public List<Class<? extends Action>> getSupportedActions()

Returns a list of Java classes extending ***Action*** and representing the actions supported in execution by this ***ToolService*** instance.

public void executeAction(Class<? extends Protocol> clazz, Action action)
throws ProtocolNotSupportedByToolException,
ActionNotSupportedException, ActionExecutionFailedException;

Executes the specified action for the specified Java class extending ***Protocol*** or throws an exception. The class is specified by the parameter ***clazz***, the action is specified by the parameter ***action***.

The ***ToolService*** API specification additionally refers to the following Java exception classes, which are also part of the HERMES framework specification:

ProtocolNotSupportedByToolException

A ***ProtocolNotSupportedByToolException*** instance is thrown when commanding a ***ToolService*** to execute an action with an unsupported protocol Java class.

ActionNotSupportedException

An ***ActionNotSupportedException*** instance is thrown when commanding a ***ToolService*** to execute an action with an unsupported action.

ActionExecutionFailedException

An ***ActionExecutionFailedException*** instance is thrown when the action execution failed.

3.7 Contacts

So far this chapter has presented components of the HERMES framework that provide necessary functions for the fundamental operation of an integrated communication management framework. In this and the upcoming sub-chapters, this thesis will focus on detailing the remaining framework components and aspects such as contacts, data, data persistency, data replication and finally supplementary, previously unnamed services.

When speaking of communication management frameworks, in general, it is the belief of this thesis that such a framework cannot exist without a proper management of contacts. Contacts constitute the elementary element of any communication-related task and especially of any communications-related software. A lack of contact management in a communication management framework like HERMES would be unacceptable. The framework presented in this thesis provides contact management as a matter of course and offers a set of basic instruments to administrate contacts.

As mentioned above, contacts are integral parts of the HERMES framework or more specifically of almost every communication-related action. Most of these actions require recipients and the framework represents recipients by contact Java objects. The first issue to address in this context is the constitution of contacts. The HERMES framework supports basic contact Java objects. These Java objects consist of a name, a mapping of protocols and the actual contact addresses, specified as ***identities***. This mapping provides a straightforward way to manage numerous identities for a single contact. This work assumes that this is the regular case since most individuals possess multiple identities, at least one for every communication tool or generally speaking every protocol they use. For example, owning at least one email identity, numerous phone numbers and numerous identities on diverse instant messaging networks is considered absolutely common nowadays. Thus every contact Java object is able to carry all identities for every possible protocol an individual may use. Every entry in the mapping can be visualized as a simple key and value pair. Each key represents a protocol and each assigned value is the actual identity or, more precisely, the actual address for the specific protocol. Listing 3.1 shows the EBNF definition for contacts as implemented in the HERMES framework and just described. The framework exposes contacts in the shape of the ***Contact*** Java class.

```
contacts = { contact } ;
contact = name, mapping ;
name = ? all visible characters ? ;
mapping = { ( protocol , identity ) } ;
protocol = ? all visible characters ? ;
identity = ? all visible characters ? ;
```

Listing 3.1: EBNF description for contacts.

The HERMES framework specifies the ***Contact*** Java class API as follows:

package at.jku.tk.hermes.contact

public final class Contact

A ***Contact*** instance allows maintaining a communication contact's name and numerous identities for each communication protocol the contact consumes.

public Contact(String name)

Creates a new ***Contact*** instance with the specified name. The name is specified by the parameter ***name***.

public String getName()

Returns the name of this ***Contact*** instance.

public void setName(String name)

Sets the name of this ***Contact*** instance. The name is specified by the parameter ***name***.

public String getIdentity(Class<? extends Protocol> clazz)

throws ProtocolNotSupportedByContactException {

Returns the identity for the specified Java class extending ***Protocol*** or throws an exception. The class is specified by the parameter ***clazz***.

public boolean putIdentity(Class<? extends Protocol> clazz, String identity)

Sets the specified identity for the specified Java class extending ***Protocol*** for this ***Contact*** instance and returns ***true*** on success or ***false*** otherwise. The class is specified by the parameter ***clazz***, the identity is specified by the parameter ***identity***.

public boolean removeIdentity(Class<? extends Protocol> clazz)

Removes the identity for the specified Java class extending ***Protocol*** for this ***Contact*** instance and returns ***true*** on success or ***false*** otherwise. The class is specified by the parameter ***clazz***.

public Map<Class<? extends Protocol>, String> getIdentities()

Returns a mapping of all protocols and their corresponding identities for this ***Contact*** instance. Each key represents a Java class extending ***Protocol*** and each assigned value is the actual identity for the specific protocol.

public boolean supportsProtocol(Class<? extends Protocol> clazz)

Returns ***true*** when this ***Contact*** instance supports maintaining an identity for the specified protocol Java class or ***false*** otherwise. The class is specified by the parameter ***clazz***.

public List<Class<? extends Protocol>> getSupportedProtocols()

Returns a list of Java classes extending ***Protocol*** representing the protocols this ***Contact*** instance can maintain identities for.

The ***Contact*** API specification additionally refers to the following Java exception class, which is also part of the HERMES framework specification:

ProtocolNotSupportedByContactException

A ***ProtocolNotSupportedByContactException*** instance is thrown when requesting a ***Contact*** to return an identity for an unsupported protocol Java class.

Referring to the previously mentioned ownership of multiple communication identities, an example contact Java object could consist of the name ***John Doe*** plus the mapping, keeping multiple protocol and identity pairs this individual may possess, for example the identity ***john.doe@domain.tld*** corresponding to the email protocol, the identity ***+0123456789*** corresponding to the PSTN protocol or the identity ***john.doe*** corresponding to the Skype protocol.

Internally, the HERMES framework treats contacts as specialized type of data. This means that the framework keeps contacts inside its database in the same manner as any other ordinary data. For the purpose of contact management simplification, the framework was equipped with the additional ***ContactService*** and ***ContactObserverService*** OSGi service interfaces. The ***ContactService*** is the service other components make use of when contact-handling is required. This service provides numerous methods to read, write and query contacts. The framework reference implementation contains a fully functional ***ContactService*** implementation registered along with the OSGi service registry.

The HERMES framework specifies the ***ContactService*** Java interface API as follows:

package at.jku.tk.hermes.contact

public interface ContactService

A ***ContactService*** instance allows performing basic operations on the framework's contacts.

public boolean containsContact(String key)

Returns ***true*** when the framework's database contains a contact identified by the specified key or ***false*** otherwise. The key is specified by the parameter ***key***.

public boolean isContactDeleted(String key)

Returns ***true*** when a contact identified by specified key was deleted or ***false*** otherwise. The key is specified by the parameter ***key***.

public boolean putContact(String key, Contact contact)

Creates or updates a contact identified by the specified key. Returns ***true*** when this operation was successful or ***false*** otherwise. The key is specified by the parameter ***key***, the contact is specified by the parameter ***contact***.

public Contact getContact(String key)

Returns a contact identified by the specified key or returns ***null*** when the contact could not be retrieved. The key is specified by the parameter ***key***.

public boolean removeContact(String key)

Deletes a contact identified by the specified key. Returns ***true*** when this operation was successful or ***false*** otherwise. The key is specified by the parameter ***key***.

public Map<String, Contact> queryContacts(String xpath)

Queries all contacts by applying the specified XPath query and returns the result as a mapping. The return value is a mapping of contact keys and contacts. Every entry in the mapping is a key and value pair. Each key represents a contact key and each assigned value is the corresponding contact. Returns an empty map when the query did not return any result. The XPath query is specified by the parameter ***xpath***.

The framework additionally provides the ***ContactObserverService*** service interface. It may be implemented by any class interested in receiving notifications about contacts' modifications. In such a case, the implementing class must be registered as a ***ContactObserverService*** service along with the OSGi service registry. The HERMES framework, more precisely the ***DatabaseServcice***, is compelled to notify all registered ***ContactObserverService*** implementations about all contacts' modifications.

The HERMES framework specifies the ***ContactObserverService*** Java interface API as follows:

package at.jku.tk.hermes.contact

public interface ContactObserverService

An ***ContactObserverService*** instance allows receiving notifications about all contacts' modifications.

public void contactPut(String key)

This method is called after a contact identified by the specified key was created or updated. The key is specified by the parameter ***key***.

public void contactRemoved(String key)

This method is called after a contact identified by the specified key was deleted. The key is specified by the parameter ***key***.

public void contactPutOnReplication(String key)

This method is called in the course of replication after a contact identified by the specified key was created or updated. The key is specified by the parameter ***key***.

public void contactRemovedOnReplication(String key)

This method is called in the course of replication after a contact identified by the specified key was deleted. The key is specified by the parameter ***key***.

During the initial design of the framework, an alternative method to represent contacts was tested. As in the final specification, contact Java objects consisted of a name and a mapping. However, the mapping was much more elaborate in this former design. Tool references were used as keys and the values were further mappings. These mappings were formed by protocol references as keys and the actual addresses as values. This composition implicated numerous shortcomings: Firstly, all addresses were de facto bound to tools, meaning that whenever a new tool was added to the framework it was necessary to extend contacts with further addresses for this new tool. Because the role of protocols was neglected, the possibility of a number of tools enclosing the same protocol to address mappings but unable to share them leading to overheads existed. Consequently not only these mappings had to be input over and over again but they all also had to be kept persistently. The outcome would have high redundancy, overhead and inefficiency, all traits which are undesirable within an integrated communication management system. Hence this design was dismissed and substituted with the aforementioned simplified and more efficient solution.

3.8 Data Persistence

As already attested, the HERMES framework requires the ability to learn and to store its knowledge about users' communication needs and habits as data in order to operate properly. Contacts have already been defined as a particular specialized kind of data. However, to take full advantage of the HERMES framework or any integrated communication management system, applications must be provided with the power to read and write any kind of data. In case of the HERMES framework, it is not far to seek that data will be composed entirely of Java objects.

The first draft of the framework considered making use of the Java Persistence API [26]. JPA is a Java framework allowing maintaining Java runtime objects beyond a single session of a Java application. Objects are serialized by applying the object-relational mapping technique and written to a relational

database. Though JPA is a mature technology provided in several implementations, its primary scope is enterprise software. This characteristic was the principal argument against this technology. During evaluation of numerous JPA implementations it became clear that from a developer's point of view, the effort to persist an object was relatively high and all evaluated JPA implementations had the drawback of consuming comparatively large amounts of resources. These are two attributes that are contrary to the objective induced by the HERMES framework's distributed nature, namely being lightweight. Although the idea of exploiting the simplicity of a traditional RDBMS along with JPA was very tempting, it was the JPA's overhead that made it eventually unsuitable for the framework presented in this thesis.

After abandoning JPA and the landscape of relational databases along with it, the consequent thought was to move to a modern object-oriented database where Java objects could be directly read from and written to the database without engaging any ORM. Despite the availability of numerous ODBMS for Java, a particular product, db4o, was selected for further investigation because it seemed most promising. Besides being a native object database for Java and .NET, db4o stood out due to low memory requirements, thus making it perfectly suitable for embedded environments. Initial tests with db4o were very encouraging. This ODBMS provided an extremely simple and efficient way of persisting Java objects. Moreover, db4o even offered a fully grown replication system based on client-server architecture that could be deployed immediately out of the box. All these qualities fulfilled the HERMES framework's demands for a storage system. Only db4o replication presented a problem. During test runs the db4o replication mechanism turned out to be virtually ineffective and unsuitable for the P2P setup used for the HERMES framework. Since this drawback turned out to be the only reason preventing db4o from becoming the database of choice for the HERMES framework, several attempts were made to replace the db4o's native replication system. However, the implementation of a custom replication system for an object-oriented database unfortunately proved to be a hard nut to crack. The first problem encountered was the circumstance that in order to be replicated, every object had to be somehow universally and globally identified. For this purpose the simplest solution seemed to be the assignation of a universally unique identifier to every object intended to be persisted. But within an object-oriented database, this would have meant that every object had to carry such an UUID even before it entered the database. From a developer's point of view, this would have resulted in equipping all data with proper UUIDs, which would have greatly and unnecessarily increased the complexity of the HERMES framework and would have made the persistence process fault-prone. Another argument against db4o and the use of an ODBMS arose from the use of XMPP – the P2P technology integrated in the framework. Using the XML-based XMPP would require all Java objects stored in db4o first to be serialized into a string representation before being transported to further peers

over XMPP. Although the serialization was possible, it turned out to be extremely delicate, particularly the task where transported objects had to be restored and merged with another peers' databases. These difficulties cancelled out the advantages of an object-oriented database and increased the effort to achieve a satisfying result. The idea of including an ODBMS in the HERMES framework finally proved ineligible. Nevertheless, the considered use of UUIDs to cross-identify objects on distributed peers and the indispensable serialization of data to XML conditioned by the use of XMPP as P2P backbone, sparked off the idea of still using a traditional RBDMS - but in a different manner.

The conclusive and finally implemented idea makes use of the simplicity of traditional RDBMS avoiding the complexity of JPA and turning the drawbacks experienced during the db4o evaluation to its advantage at the same time.

As described above, the goal was to write and read any Java object in a simple way and replicate the objects stored in an embedded database over a P2P network. Using the XML-based XMPP as the underlying P2P technology implicates serializing data to XML. This awards two significant advantages: On the one hand, XML-serialized data can be easily transported over XMPP from peer to peer and on the other hand, the use of XML does not limit the spectrum of readable and writeable data. This means that not only Java objects but any kind of data may theoretically be contained in the database as long as it is serializable to XML. Also, XML had proven itself to be an effective language to describe data in heterogeneous environments and ensure open access to this data. Furthermore, the replication process itself was simplified because peers only had to replace simple strings containing XML representations of data instead of genuine Java objects. And lastly, the utilization of XML enables applications built on top of HERMES to query and select certain XML nodes from serialized data using the XPath query language. The actual serialization of Java objects to XML and de-serialization of Java objects from XML may be outsourced to third party libraries as it was done in the reference implementation of the HEMRES framework. It is certainly possible to do this using Java's native XML encoder and decoder. However, HERMES uses the third party open source library XStream [27] to serialize contact Java objects to XML and back again in an extremely simple way.

At this point, a short introduction to XStream is provided to get a general idea of this library. The first requirement is a Java class whose instances will be persisted. Most suitable for this scenario is the ***at.jku.tk.hermes.contact.Contact*** Java class whose outline is shown in Listing 3.2.

```
// package statement and imports
public class Contact {
  private String name;
  private final Map<Class<? extends Protocol>, String>;
  // further implementation
}
```

Listing 3.2: Outline of the at.jku.tk.hermes.contact.Contact Java class.

It is notable that all ***Contact*** class' fields are private. This does no harm to XStream since the library neither cares about the visibility nor requires any special getters, setters or even a default constructor. In order to use the XStream library it has to be instantiated first as shown in Listing 3.3.

```
XStream xstream = new XStream();
```

Listing 3.3: Instantiating XStream.

In the next step a ***Contact*** instance has to be created and populated with data as shown in Listing 3.4.

```
Contact contact = new Contact("John Doe");
contact.putIdentity(
  at.jku.tk.hermes.protocol.EmailSmtpProtocol.class,
  "john.doe@domain.tld"
);
contact.putIdentity(
  at.jku.tk.hermes.protocol.PhoneFixedProtocol.class,
  "+0123456789"
);
contact.putIdentity(
  at.jku.tk.hermes.protocol.SkypeProtocol.class,
  "john.doe"
);
```

Listing 3.4: Creating a Contact instance and populating it with data.

Now, in order to serialize the ***Contact*** instance to XML all that is required is make a simple call to XStream as shown in Listing 3.5.

```
String xml = xstream.toXML(contact);
```

Listing 3.5: Serializing a Contact instance to XML.

The created Java object contains the resulting XML that looks like Listing 3.6.

```
<at.jku.tk.hermes.contact.Contact>
  <name>John Doe</name>
  <java.util.Map>
    <at.jku.tk.hermes.protocol.EmailSmtpProtocol.class>
      john.doe@domain.tld
    </at.jku.tk.hermes.protocol.EmailSmtpProtocol.class>
    <at.jku.tk.hermes.protocol.PhoneFixedProtocol.class>
      +0123456789
    </at.jku.tk.hermes.protocol.PhoneFixedProtocol.class>
    <at.jku.tk.hermes.protocol.SkypeProtocol.class>
      john.doe
    </at.jku.tk.hermes.protocol.SkypeProtocol.class>
  </java.util.Map>
</at.jku.tk.hermes.contact.Contact>
```

Listing 3.6: XML of a serialized Contact instance.

To de-serialize and restore an object from XML, another simple call to XStream has to be made as shown in Listing 3.7.

```
Contact newContact = (Contact) xstream.fromXML(xml);
```

Listing 3.7: Deserializing a Contact instance from XML.

The presented XML serialization using XStream allows skipping annoyances experienced with JPA like special annotations or schema descriptions simultaneously, saving a vast quantity of time during development. The XStream library is just a sample XML serialization library chosen for the reference implementation of the HERMES framework. The developer is not bound by the framework to use XStream, in fact he may freely choose the way to XML-serialize his data. The only requirement of the framework is that the data passed to the framework has to be stored in the database as XML. One could now argue that the whole process of serializing and finally de-serializing Java objects is a drawback to the entire system. It is the belief of this thesis that this is not the case. The boundaries set by the framework are marginal, so that the developer may choose his custom way of XML serialization (e.g. by choosing a freely available open source library like XStream). More importantly, during the evaluation this method, reading, persistently writing and replicating data has proven to be much more simple and efficient than with JPA or db4o. When applied, the introduced design reveals the simplicity of the replication, where a string containing XML has to be fetched from the sending

peer's database, is then transferred via XMPP to the receiving peer and finally replaces the receiver's database entry.

Another drawback experienced during the db4o evaluation was the need to cross-identify data of various peers. In the case of an ODBMS, the developer would have to supply every Java object with a proper UUID himself. As mentioned before, the consequences would be increased complexity of the entire framework and a fault-prone persistence process. Using a RDBMS instead offers the possibility to separate the identifier from the actual Java object. The developer still needs to provide an UUID in order to globally pinpoint and replicate objects, but he does not have to imprint that identifier upon the actual Java object. This approach shows a significant improvement and simplification over the design utilizing an ODBMS.

During the preliminary advances to implement the idea outlined above, another database system evaluation was carried out. This time, Apache Xindice [28], a native XML database for Java was chosen to operate as the framework's data store. It soon became apparent that Xindice, not unlike db4o, would not be able to meet the needs of the final design. For example, trivial functionality present in any RDBMS, like attaching auto-incremented integer values to the database entries or returning a sorted query result, were not supported by Xindice and thus disqualified this database from further consideration. However, despite the encountered flaws during the assessment phase, Xindice unveiled a notable aspect helping organizing stored XML by introducing a top cataloging layer called ***collections***. Collections in Xindice are the equivalent for tables in RDBMS. Introducing this concept to a native XML database seemed very reasonable and thus was also incorporated into the final design of HERMES' embedded database. The reference implementation reserves several collection names, namely ***at_jku_tk_hermes_contact***, ***at_jku_tk_hermes_drs_ids*** and ***at_jku_tk_hermes_drs_log***. The first one is used to store contacts and the latter ones are utilized for the replication process.

Since all other attempts considering advanced databases or advanced database technologies to simplify the data persistence in HERMES failed, it was deemed prudent to return to simple RDBMS. The requirements for an actual product were the ability to run in embedded mode, small footprint and high performance as well as the database's support of the JDBC API version 4.0. In addition to numerous improvements, JDBC 4.0 introduces several SQL 2003 facets, including a Java-specific mapping for the SQL's native XML data type [29]. At time of development of the HERMES framework's reference implementation, only Apache Derby [30] sufficiently satisfied all prerequisites, especially the support for the XML data type, and was chosen to backbone the persistence in the framework's reference implementation. In the case of Apache Derby, the framework requires additional information about

where to store the actual database's data in the file system. This information is acquired from the custom Java system property ***at.jku.tk.hermes.storage,*** which has to be provided by the developer. It should be noted that numerous further HERMES-related Java system properties are required by the framework's reference implementation but these will be discussed in detail in the following sub-chapters.

Listing 3.8 shows a generalized SQL statement for creating every collection holding data objects in the framework's reference implementation. This statement demonstrates the overall achieved simplicity by persisting all data objects in XML. Each entry in such a collection consists of a unique key, globally identifying the data object and the XML-serialized data objects itself.

```
CREATE TABLE <collection_name> (
  obj_key VARCHAR(1024) NOT NULL PRIMARY KEY,
  obj_xmls11n XML NOT NULL
)
```

Listing 3.8: Generalized SQL statement for creating every data object collection.

In order to meet the goal of maximizing the HERMES' data persistence simplicity, the framework was equipped with additional supplementary service interfaces. The ***DatabaseService*** is one of these service interfaces and provides numerous methods to read, write and query the persisted data.

The HERMES framework specifies the ***DatabaseService*** Java interface API as follows:

package at.jku.tk.hermes.core

public interface DatabaseService

A ***DatabaseService*** instance allows performing basic operations on the framework's embedded database.

public boolean containsCollection(String collection)

Returns ***true*** when the framework's database contains a collection identified by the specified collection name or ***false*** otherwise. The name is specified by the parameter ***collection***.

public boolean isCollectionDeleted(String collection)

Returns ***true*** when a collection identified by the specified collection name was deleted in the framework's database or ***false*** otherwise. The name is specified by the parameter ***collection***.

public boolean createCollection(String collection)

Creates a collection identified by the specified collection name in the framework's database and returns ***true*** when this operation was successful or ***false*** otherwise. The name is specified by the parameter ***collection***.

public boolean dropCollection(String collection)

Deletes a collection identified by the specified collection name in the framework's database and returns ***true*** when this operation was successful or ***false*** otherwise. The name is specified by the parameter ***collection***.

public boolean containsObject(String collection, String key)

Returns ***true*** when the framework's database contains a data object identified by the specified key in the specified collection or ***false*** otherwise. The collection name is specified by the parameter ***collection***, the key is specified by the parameter ***key***.

public boolean isObjectDeleted(String collection, String key)

Returns ***true*** when a data object identified by the specified key in the specified collection was deleted in the framework's database or ***false*** otherwise. The collection name is specified by the parameter ***collection***, the key is specified by the parameter ***key***.

public boolean putObject(String collection, String key, String xml)

Creates or updates a data object identified by the specified key in the specified collection with the specified XML serialization. Returns ***true*** when this operation was successful or ***false*** otherwise. The collection name is specified by the parameter ***collection***, the key is specified by the parameter ***key***, the XML serialization is specified by the parameter ***xml***.

public String getObject(String collection, String key)

Returns a XML-serialized data object identified by the specified key in the specified collection. Returns ***null*** when the data object could not be retrieved. The collection name is specified by the parameter ***collection***, the key is specified by the parameter ***key***.

public boolean removeObject(String collection, String key)

Deletes a data object identified by the specified key in the specified collection. Returns ***true*** when this operation was successful or ***false*** otherwise. The collection name is specified by the parameter ***collection***, the key is specified by the parameter ***key***.

public Map<String, String> queryCollection(String collection, String xpath)

Queries the specified collection by applying the specified XPath query and returns the result as a mapping. The return value is a mapping of data object keys and data objects' XML serializations. Every entry in the mapping is a key and value pair. Each key represents a data object key and each assigned value is the corresponding XML serialization of this data object. Returns an empty map when the query did not return any result. The collection name is specified by the parameter ***collection***, the XPath query is specified by the parameter ***xpath***.

In addition, the framework provides another service interface called ***DatabaseObserverService***. It may be implemented by any class interested in receiving notifications about database modifications. In such a case, the implementing class must be registered as a ***DatabaseObserverService*** service along with the OSGi service registry. The HERMES framework, more precisely the ***DatabaseServcice***, is compelled to notify all registered ***DatabaseObserverService*** implementations about database modifications. It is worth mentioning that no observer is notified about modification in the framework reference implementation's internal replication-related collections ***at_jku_tk_hermes_drs_ids*** and ***at_jku_tk_hermes_drs_log***.

The HERMES framework specifies the ***DatabaseObserverService*** Java interface API as follows:

package at.jku.tk.hermes.core

public interface DatabaseObserverService

A ***DatabaseObserverService*** instance allows receiving notifications about the framework's embedded database's modifications.

public void collectionCreated(String collection)

This method is called after a collection identified by the specified collection name was created. The name is specified by the parameter ***collection***.

public void collectionDropped(String collection)

This method is called after a collection identified by the specified collection name was deleted. The name is specified by the parameter ***collection***.

public void collectionCreatedOnReplication(String collection)

This method is called in the course of replication after a collection identified by the specified collection name was created. The name is specified by the parameter ***collection***.

public void collectionDroppedOnReplication(String collection)

This method is called in the course of replication after a collection identified by the specified collection name was deleted. The name is specified by the parameter ***collection***.

public void objectPut(String collection, String key)

This method is called after a data object identified by the specified key in the specified collection was created or updated. The collection name is specified by the parameter ***collection***, the key is specified by the parameter ***key***.

public void objectRemoved(String collection, String key)

This method is called after a data object identified by the specified key in the specified collection was deleted. The collection name is specified by the parameter ***collection***, the key is specified by the parameter ***key***.

public void objectPutOnReplication(String collection, String key)

This method is called in the course of replication after a data object identified by the specified key in the specified collection was created or updated. The collection name is specified by the parameter ***collection***, the key is specified by the parameter ***key***.

public void objectRemovedOnReplication(String collection, String key)

This method is called in the course of replication after a data object identified by the specified key in the specified collection was deleted. The collection name is specified by the parameter ***collection***, the key is specified by the parameter ***key***.

To sum up, the HERMES' persistence mechanism consists of multiple units. Firstly, not only Java objects may be persisted but any kind of data as long as it is serializable to XML. The framework's persistence mechanism expects XML data to persist and returns XML on queries. The actual serialization and de-serialization does not fall within the framework's duties, although the framework's reference implementation is using the third party open source library XStream to serialize Java objects to XML and back again. To store the XML, a conventional embedded RDBMS has been proposed. The product Apache Derby was chosen for the reference implementation, primarily because of its support for the XML data type and the ability to query data with XPath queries. Data stored in the embedded database is generally identified by a universally unique

identifier, whereby the use of UUIDs is recommended. The identifier has to be extraneous provided and is stored along with the data but data is not required to be bundled with identifiers. Additionally and in order to improve the overall structure, data is stored in so-called collections which correspond to tables in RDBMS. Finally, the framework provides a service to manage data in a simple manner as well as an applicable interface for services interested in being notified about database modifications.

3.9 Data Replication

This sub-chapter examines the HERMES' database replication mechanism. Apart from the design and the implementation of an integrated communication framework, a further goal of this thesis was to enable multiple HERMES-based applications to operate and interact with one another in distributed heterogeneous networks simultaneously. A common use case scenario for this requirement would be any average user utilizing a HERMES-based application at multiple locations, for example at home and at work. In a situation like this, applications need to communicate with each other to ensure the same level of knowledge required to properly support the user in his communication. Since this knowledge is represented in HERMES by XML-serialized data stored in the embedded database, the only issue the applications have to be aware of is the replication of their databases from peer to peer, using the framework's XMPP-based communication channel. Due to the fact that Apache Derby, the embedded database utilized in the framework's reference implementation, does not support such kind of replication out of the box, a custom mechanism had to be designed and implemented. Because the RDBMS replication over XMPP plays a significant role in the overall HERMES framework, it also represents another major contribution of this thesis besides the framework itself.

XMPP was chosen primarily to back the inter-peer communication and replication process, because its stability and way of setting up and utilizing technology. Previously described as hybrid decentralized unstructured P2P technology, XMPP enables peers to logically communicate directly with other peers via addresses assembled similar to e-mail addresses and usually referred to as Jabber ID. Once again it has to be said that due to this characteristic and the fact that XMPP servers may be aggregated into federations, XMPP is not a pure server-client technology in the understanding of this thesis. For the HERMES reference implementation, the third party library Smack API [31] was chosen to provide the necessary XMPP functionality. For testing purposes and to demonstrate successful operation the XMMP server Openfire [32] was used. The Smack API and the Openfire server are both open source and are developed and maintained by Jive Software, a company specializing in open source messaging solutions.

HERMES utilizes full lazy non-optimistic multi-master replication, whereby all peers store full copies of the same data and all copies can be updated simultaneously. On the one hand, HEMRES benefits from this replication strategy by improved data availability, freely selectable time for updates propagation and avoidance of bottlenecks, single points of failure and blockers due to unavailable replicas. On the other hand, multi-master replication requires HERMES to apply an individual reconciliation algorithm based on logging and timestamping. Because of the distributed nature of updates, full replication requires more storage space and a higher number of message throughputs in XMPP, resulting in higher network load. Furthermore, this strategy implicates possible divergences amongst replicas, which is why local reads cannot promise to return the most recent values. Nevertheless, it is the belief of this work that full lazy multi-master replication is the best choice for HERMES, considering the framework's requirements.

The basic idea of the entire replication process is pull-based, meaning that a peer has to initiate requests to other peers in order to receive the most recent values from other replicas. But foremost, the ability to properly respond to such requests is tied with each peer's need to keep account of all its database modifications. Modifications are creations and deletions of collections as well as creations, updates and deletions of actual data objects. For this purpose the HERMES reference implementation stores metadata about all database transaction in a special collection, mentioned above and labeled ***at_jku_tk_hermes_drs_log***. The metadata for each transaction consists of the timestamp of the operation, the operation's type, the affected collection's name and the affected data's key. In such metadata, there may be no key since modifications on collections are logged as well. Each entry in this particular collection is additionally tagged with a unique integer value as primary key. Listing 3.9 shows the SQL statement of the ***at_jku_tk_hermes_drs_log*** collection as described.

```
CREATE TABLE at_jku_tk_hermes_drs_log (
  id BIGINT NOT NULL PRIMARY KEY GENERATED ALWAYS AS IDENTITY
    (START WITH 1, INCREMENT BY 1),
  obj_timestamp BIGINT NOT NULL,
  obj_operation CHAR(1) NOT NULL,
  obj_collection VARCHAR(1024) NOT NULL,
  obj_key VARCHAR(1024)
)
```

Listing 3.9: SQL statement for creating the at_jku_tk_hermes_drs_log collection.

The operation's timestamp in milliseconds is retrieved from the HERMES core ***TimestampService*** service. The reference implementation of this service uses Java's native ***java.lang.System.currentTimeMillis()*** method for this purpose. However, the employment of

currentTimeMillis() involves two major problems. Firstly, according to Java API specification [33] this method returns the *"…current time in milliseconds. Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger. For example, many operating systems measure time in units of tens of milliseconds."* Because of different operating system granularities, the uncertainty about a platform's accurate clock resolution and potential further delays, the implemented ***TimestampService*** is incapable of returning values accurate to the millisecond and therefore returns instead values accurate to 100 milliseconds to compensate all these discrepancies. This is performed by stripping off the last two digits of a timestamp value. The second problem caused by the use of ***currentTimeMillis()*** is the need to synchronize all peers' clocks involved in the replication process. It is obvious that for a replication strategy relying on timestamping, synchronizing the clocks is the only way to determine which replica holds the most recent version of a specific data object. In order to perform the clock synchronization HERMES – concretely the ***TimestampService*** – uses NTP. The values returned by the ***TimestampService*** are already corrected by the current NTP offset. The only information that HERMES requires in this case is the custom Java system property ***at.jku.tk.hermes.time*** pointing to a public NTP time server, whereby the reference implementation is imparted with ***europe.pool.ntp.org***.

The HERMES framework specifies the ***TimestampService*** Java interface API as follows:

package at.jku.tk.hermes.core

public interface TimestampService

A ***TimestampService*** instance allows obtaining an accurate current timestamp value of the system regardless of the system's granularities, clock resolution and potential further delays.

public long getAccurateTimestamp()

Returns the system's current timestamp value accurate to 100 milliseconds and corrected by the system's NTP offset.

Apart from the timestamps, the unique integer values mentioned earlier and serving as primary keys are crucial to the entire replication process. Once a peer initiates a request to another peer asking for modified data to replicate, the responding peer should only send modifications that the requester has not acquired yet. To accomplish this, the requester must inform the responder about modifications he had already acquired. To do so, the requester remembers the unique integer values identifying all already received modifications for each peer he contacts. The requester sends the last responded identifier from the corresponding peer along with every request. Listing 3.10 shows the SQL statement of the ***at_jku_tk_hermes_drs_ids*** collection, storing the last received modification identification for each peer identified by its JID.

```
CREATE TABLE at_jku_tk_hermes_drs_ids (
  peer_jid VARCHAR(3072) NOT NULL PRIMARY KEY,
  peer_log_id BIGINT NOT NULL DEFAULT 0
)
```

Listing 3.10: SQL statement for creating the at_jku_tk_hermes_drs_ids collection.

It is also worth noting that ***at_jku_tk_hermes_drs_ids*** and ***at_jku_tk_hermes_drs_log*** are the sole specific database fractions that are never replicated but are just meant to enable and support replication. Another characteristic of the HERMES' database replication mechanism is the fact that the entire replication is carried out by the framework in the background and is fully autonomous, which means that users and even developers are virtually relieved of this task – a further user-friendly advantage of the framework. In the background, the framework's ***DatabaseService*** implementation feeds both replication-relevant collections with appropriate data. It does so whenever a collection or a data object is locally created, updated or deleted. In these cases, a new entry describing the modification is automatically added by the framework to the ***at_jku_tk_hermes_drs_log*** collection. In order to keep the size of this collection slim, the framework automatically removes entries that are overridden by new entries. Additionally, whenever a requester receives a collection or a data object to replicate from a responder, the entry memorizing the responder's new modification identification in the ***at_jku_tk_hermes_drs_ids*** collection is automatically updated or created if not already existent. If no database modification has taken place since the last request, the responder simply does not answer.

So far the framework's internal requirements for a functional replication mechanism have been covered. Now it is time to examine the actual replication process in-depth. Once a HERMES-based application establishes a connection to an XMPP server, the framework's reference implementation activates a special daemon process. This process is responsible for repeatedly polling other peers in order to receive the most recent data objects from other replicas. It is evident that not every peer is able to poll every other peer connected to the XMPP network. Such behavior would result in extreme overhead and inefficiency leading to the opposite of what is the objective of an integrated communication management framework. Thus, the HERMES framework proposes the concept of ***replication tribes***. A replication tribe is nothing other than a group of peers belonging together in terms of replication. This means that only related peers belonging to the same tribe may be interested in each others' replicas. A common use scenario for this would be a user running HERMES-based applications at numerous locations like at home and at work. This user would not be interested in receiving modified data objects from any other replicas but his own. Therefore, apart from addressing each peer in the XMPP network with a unique name, the framework's replication mechanism expects an additional name prefix to be provided. This prefix represents the name of the replication tribe and is

concatenated with an underscore character and the actual peer name, forming the actual unique JID. The JID uniquely identifies peers in the entire XMPP network and the replication name prefix allows discovering peers belonging to the same replication tribe. A replication tribe name is embedded in the JID, which identifies a peer uniquely once connected to the XMPP network. This approach implicates that replication tribe names are static once the network connection is established. Furthermore, currently they may only exist in a one to one relationship with peers.

In order to obtain a list of its tribe fellows, a peer utilizes the XMPP extension XEP-0055 [34], which allows searching information repositories on the XMPP network. The basic function of this extension is to query an information repository regarding the possible search field, to send a search query and finally to receive search results. Since in HERMES every peer's JID is the concatenation of its replication tribe name and its actual name, it is safe to assume that a search query requesting all peers whose JID starts with an identical string will return all peers from the same replication tribe. Accordingly, a HERMES peer can send a search query to an XMPP server or multiple XMPP servers, asking for a list of usernames starting with its replication tribe's name. The returned list contains usernames of peers from the same replication tribe as the sender. All this takes place in the background without the need of user interaction. However, the framework's reference implementation requires information in the shape of the Java system properties in order perform replication properly, as shown in Table 3.1.

While carrying out replication, the HERMES reference implementation sets a daemon process in motion, which keeps an account of all database modifications, requests a list of its replication tribe fellows from XMPP servers and repeatedly polls its replication tribe fellows in order to receive the most recent data objects. Furthermore, the replication process or rather the replication daemon has to replace local data objects with the received remote data objects. This requires a differentiated behavior depending on the state of the local data object and the received remote data object as shown in Table 3.2. Firstly, it has to discriminate if the local object does or does not exist or was marked as deleted. The further execution depends on the local data object's state and the received data object's operation. HERMES distinguishes three operations that are also logged in the ***at_jku_tk_hermes_drs_log*** collection: either a data object was created, updated or deleted. It is worth noting that the replication mechanism does not perform any merging but only replaces local copies of data objects with received remote data object when mandatory.

at.jku.tk.hermes_xmpp.server	The IP address of the XMPP server to log on (e.g.: ***140.78.92.143***).
at.jku.tk.hermes_xmpp.search	The search service name of the XMPP server (e.g.: ***search.140.78.92.143***).
at.jku.tk.hermes_xmpp.username	The name of this peer (e.g.: ***alice***). The peer's name prefixed by the replication tribe name specified by ***at.jku.tk.hermes_xmpp.tribe.name*** and an underline character together from the username to log on the XMPP server.
at.jku.tk.hermes_xmpp.password	The password to log on the XMPP server (e.g.: ***password***).
at.jku.tk.hermes_xmpp.tribe.servers	An optional list of further XMPP server addresses separated by commas (e.g.: ***140.78.92.140,140.78.92.141,140.78.92.142***). These servers are expected to form a federation with the server specified by ***at.jku.tk.hermes_xmpp.server*** and will be queried for lists of peers belonging to the replication tribe specified by ***at.jku.tk.hermes_xmpp.tribe.name***.
at.jku.tk.hermes_xmpp.tribe.name	The replication tribe name of this peer (e.g.: ***8c90ab08-a5d5-4633-9c70-f671456e72ae***).

Table 3.1: Required Java system properties for proper replication execution.

		received remote data object		
		created	***updated***	***deleted***
local data object	***does not exist yet***	create	create	log as deleted
	exists	illegal	compare timestamps and replace or skip	delete
	deleted	skip	skip	skip

Table 3.2: Synchronization and merge strategy during replication.

The precondition for the replication mechanism to take any action is that the local data object and the remote data object come from the same collection and have the same key. As stated in Table 3.2 the following situations may occur during reconciliation: If the local data object does not exist yet and the remote data object was created or updated, then the replication mechanism creates this data object in the local database. If the local data object does not exist and the remote data object was deleted, then the replication mechanism marks the object as deleted in the local ***at_jku_tk_hermes_drs_log*** collection. Once a data object is marked as deleted, the framework forbids recreating it. That means that every deletion is final. If the local data object exists, the handling differs. In this case, the framework specifies that creating identical objects on different peers is illegal. Identical objects have the same collection and the same key. However, if the remote data object was updated, the replication mechanism has to compare the timestamp of the local data object with the timestamp of the remote data object. If the remote timestamp is higher compared to the local one, it is obvious that the remote data object is newer than its local copy and that the replication mechanism has to replace the local copy. Otherwise the local data object has the higher timestamp and does not need to be replaced, so the remote data object is simply ignored. In case that the remote data object was deleted and the local data object still exists, the local copy needs to be deleted too, since the replication mechanism handles all deletions as final. Lastly, any remote data object received is skipped if the local data object was deleted already, again due to the final nature of delete operations.

Conclusively, the replication mechanism possesses certain traits worth noting. Firstly, an important adjustment was made on the employed open source XMMP server Openfire. It was critical to disable offline messages so the server would never store offline messages in case a recipient was unavailable. In this case the server would just drop the messages and neither the sender nor the receiver would be notified. Considering the intensive messaging that occurs between HERMES peers, this setting was significant, because messages from peers to other peers who are offline would be stored by the server and forwarded to the offline peers once coming online by default. This would not only produce a considerable overhead on the server but also cause confusion at peer level, who would possibly have to deal with a massive number of outdated messages when logging on again. Another aspect to keep in mind is that the framework's reference implementation always responds with only one modified data object per request.

The presented replication mechanism has been implemented mostly as an extension to the aforementioned Smack API library. The ***ReplicationDaemon*** Java class sets two daemon processes in motion. One daemon is responsible for connecting to the XMPP network and maintaining the established connection. The second one is in control of updating the local embedded database with

modified data objects received from other replicas. The class initiates requests to other peers for modified data objects by obtaining a list of its replication tribe fellows from the XMPP network first and also responds to requests for modified data objects. Furthermore, the ***ReplicationDaemon*** registers the ***ReplicationManager*** Java class as an XMPP extension. The ***ReplicationManager*** registers a packet listener to the XMPP connection, processes incoming packages and notifies its own ***ReplicationRequestListener*** Java class listeners about received requests for modified data objects and its own ***ReplicationResponseListener*** Java class listeners about responses containing modified data objects. The ***ReplicationManager*** is also responsible for the actual sending and receiving of XMPP messages on packet level. The ***Replication*** Java class implements the Smack API's ***PacketExtension*** Java interface and serves as the XMPP extension intended to be sent over the XMPP network, holding modified data objects and providing basic getter and setter methods. As the final piece of the replication mechanism, the ***ReplicationProvider*** Java class implements the Smack API's ***PacketExtensionProvider*** Java interface and is in charge of properly parsing the XML of received XMPP messages in order to restore the transported ***Replication*** Java objects.

Regarding the Java classes involved in the replication process the framework's reference implementation defines the following six classes and interfaces as being primarily responsible for the entire replication:

at.jku.tk.hermes.core.internal.replication.ReplicationDaemon

Connects to the XMPP network; registers the ***ReplicationProvider*** as XMPP extension provider; initiates requests to other peers for modified data objects and responds to such requests; receives modified data objects and updates these in the local embedded database.

at.jku.tk.hermes.core.internal.replication.smackx.ReplicationManager

Processes incoming XMPP packets and notifies appropriate listeners about incoming requests and responses; responsible for communication on packet level with other peers by means of messages.

at.jku.tk.hermes.core.internal.replication.smackx.ReplicationRequestListener

Interface for listeners interested in being notified about received modified data object requests.

at.jku.tk.hermes.core.internal.replication.smackx.ReplicationResponseListener

Interface for listeners interested in being notified about received modified data objects responses.

at.jku.tk.hermes.core.internal.replication.smackx.packet.Replication

XMPP packet representation implementing the Smack API's ***PacketExtension*** interface; holds modified data objects; provides serialization of itself to an XMPP messages.

at.jku.tk.hermes.core.internal.replication.smackx.provider.ReplicationProvider

XMPP extension provider implementing the Smack API's ***PacketExtensionProvider***; restores instances of ***Replication*** out of received XMPP messages.

For the sake of completeness, Listing 3.11 shows a typical request in the shape of an XML-based XMPP message sent from a peer to another replica requesting a modified data object. By contrast, Listing 3.12 shows a typical response in the shape of an XML-based XMPP message sent from a replica peer returning a modified data object.

The XMPP message presented in Listing 3.11 is sent as a request from the peer ***8c90ab08-a5d5-4633-9c70-f671456e72ae_alice*** on the XMPP server ***140.78.92.143*** to the peer ***8c90ab08-a5d5-4633-9c70-f671456e72ae_bob*** on the same server. The replication tribe name is ***8c90ab08-a5d5-4633-9c70-f671456e72ae***. The actual request is surrounded by the ***x*** tag that generally expresses an XMPP extension. In this case, the XML namespace ***jabber:x:at:jku:tk:hermes:replication*** is identifying HERMES' custom extension. The following tags ***mode*** and ***id*** specify the details of the request. The ***mode*** tag describes the request whereby the content may equal ***Q*** for requests or ***A*** for responses. In case of a request the ***id*** tag contains the unique integer value identifying an entry in the ***at_jku_tk_hermes_drs_log*** collection the requester is interested in.

```
<message
    id="8XcfI-1"
    to="8c90ab08-a5d5-4633-9c70-f671456e72ae_bob@140.78.92.143">
  <x xmlns="jabber:x:at:jku:tk:hermes:replication">
    <mode>Q</mode>
    <id>1</id>
  </x>
</message>
```

Listing 3.11: XMPP message requesting a data object to replicate.

The XMPP message presented in Listing 3.12 is sent as a response from the peer ***8c90ab08-a5d5-4633-9c70-f671456e72ae_bob*** on the XMPP server ***140.78.92.143*** to the peer ***8c90ab08-a5d5-4633-9c70-f671456e72ae_alice*** on the same server. The replication tribe name is ***8c90ab08-a5d5-4633-9c70-f671456e72ae***. The actual response is also surrounded by the ***x*** tag using the same XML namespace. Contrary to the requests, Listing 3.12 shows different content of the ***mode*** tag indicating a response. In case of a response the ***id*** tag contains the unique integer value identifying the requested entry in the ***at_jku_tk_hermes_drs_log*** collection. Additionally the response message carries further information to properly perform reconciliation. Most importantly, the ***timestamp*** tag's content stands

for the operation's timestamp in milliseconds. Then the ***operation*** tag's content describes the logged operation carried out at the replica, whereby the content may equal ***C*** for creation, ***U*** for update or ***D*** for deletion. The collection name and the key of the modified data objects required to identify it in the requester's local embedded database are appended within the ***collection*** and the ***key*** tags. Finally the modified and XML-serialized remote data object itself is included inside of the ***xmlS11N*** tag.

```
<message
    id="Cwp2G-1"
    to="8c90ab08-a5d5-4633-9c70-f671456e72ae_alice@140.78.92.143">
  <x xmlns="jabber:x:at:jku:tk:hermes:replication">
    <mode>A</mode>
    <id>2</id>
    <timestamp>1238245413200</timestamp>
    <operation>C</operation>
    <collection>at_jku_tk_hermes_contact</collection>
    <key>070cfbba-f1df-466d-980f-58ab7fa34b06</key>
    <xmlS11N>
      <at.jku.tk.hermes.contact.Contact>
        <name>John Doe</name>
        <identities>
          <entry>
            <java-class>at.jku.tk.hermes.protocol.EmailSmtpProtocol</java-class>
            <string>john.doe@domain.tld</string>
          </entry>
          <entry>
            <java-class>at.jku.tk.hermes.protocol.PhoneFixedProtocol</java-class>
            <string>+0123456789</string>
          </entry>
          <entry>
            <java-class>at.jku.tk.hermes.protocol.SkypeProtocol</java-class>
            <string>john.doe</string>
          </entry>
        </identities>
      </at.jku.tk.hermes.contact.Contact>
    </xmlS11N>
  </x>
</message>
```

Listing 3.12: XMMP message responding with a data object to replicate.

To sum up, the HERMES' database replication mechanism is defined by multiple characteristics. The framework utilizes full lazy non-optimistic multi-master replication allowing all peers to store full copies of the same data and also allowing simultaneous updates. The framework's knowledge is generally represented by XML-serialized data stored in an embedded database that is not aware of

replication. Therefore the framework's duty is the replication of databases from peer to peer utilizing the framework's XMPP-based communication channel. The entire replication process is fully autonomous and pull-based, requiring peers to contact other peers in order to receive the most recent copies of data objects. The framework keeps account of all its database modification and based on these records, is able to properly respond to other peer's requests. Apart from logging all of the database's transactions, the framework's accounting deploys timestamping in order to enable identification of the most recent copy of a data object globally amongst all replicas. Timestamps accurate to 100 milliseconds and corrected by the system's NTP offset are obtainable from a built-in framework service. Furthermore, the HERMES framework introduces so called replication tribes, representing groups of related peers interested in each others' replicas. Addresses of replication tribe members can be obtained by querying the XMPP network. Finally, the multi-master replication requires the HERMES framework to apply an individual synchronization and merge strategy during replication of data objects depending on the status of the local data objects and the received remote data objects.

3.10 Supplements

The final sub-chapter in the series focusing on the architecture and the reference implementation of the HERMES framework dedicates itself to the so far unmentioned framework components ***ParameterService*** and ***ParameterException***.

The ***ParameterService*** serves as a central resource for acquiring special Java system properties demanded by the HERMES framework to operate smoothly. In particular, HERMES requires knowledge about where to store its embedded database, which public NTP time server to use for its ***TimestampService*** and finally numerous properties for its replication mechanism to operate properly. Basically, the service facilitates the access to the aforementioned HERMES-related Java system properties.

The HERMES framework specifies the ***ParameterService*** Java interface API as follows:

package at.jku.tk.hermes.core

public interface ParameterService

A ***ParameterService*** instance allows obtaining a custom HERMES-related Java system property.

public File getStorage() throws ParameterException

Returns the directory where HERMES will deposit its data or throws an exception. The directory is specified by the system property ***at.jku.tk.hermes.storage***.

public String getNtpTimeServer() throws ParameterException

Returns the public NTP time server used by the ***TimestampService*** to correct timestamp values by the system's current NTP offset or throws an exception. The server is specified by the system property ***at.jku.tk.hermes.time***.

public String getXmppServer() throws ParameterException

Returns the IP address of the XMPP server to log on or throws an exception. The address is specified by the system property ***at.jku.tk.hermes.xmpp.server***.

public String getXmppUsername() throws ParameterException

Returns the peer's username (also introduced as JID) to log on the XMPP server with or throws an exception. The username consists of the replication tribe name concatenated with an underscore character and the peer name. The name is specified by the system properties ***at.jku.tk.hermes.xmpp.tribe.name*** and ***at.jku.tk.hermes.xmpp.username***.

public String getXmppBareUsername() throws ParameterException

Returns the peer's name or throws an exception. Every whitespace and underline character is automatically removed from the name. Furthermore the name may be 36 characters long at most. The name is specified by the system property ***at.jku.tk.hermes.xmpp.username***.

public String getXmppPassword() throws ParameterException

Returns the peer's password to log on the XMPP server with or throws an exception. The password is specified by the system property ***at.jku.tk.hermes.xmpp.password***.

public String[] getXmppTribeServers() throws ParameterException

Returns the list of further optional XMPP server IP addresses or throws an exception. These servers are expected to form a federation with the server specified by the system property ***at.jku.tk.hermes.xmpp.server***. The addresses are specified by the system property ***at.jku.tk.hermes.xmpp.tribe.servers***.

public String getXmppTribeName() throws ParameterException

Returns the peer's replication tribe name or throws an exception. Every whitespace and underline character is automatically removed from the name. Furthermore the name may be 36 characters long at most. The name is specified by the system property ***at.jku.tk.hermes.xmpp.tribe.name***.

The ***ParameterService*** API specification additionally refers to the following Java exception class, which is also part of the HERMES framework specification:

ParameterException

A ***ParameterException*** instance is thrown when requesting a ***ParameterService*** instance to return a HEMRES-related Java system property that was not set.

4 Application Development and Extensions

This chapter illustrates the development of HERMES-based applications on the basis of a sample application implemented for this thesis and sketches the possibilities to extend the framework's current capabilities. The following sub-chapters discuss the sample application leveraging the framework's reference implementation by narrating its custom components and demonstrating the fundamental approach to develop HERMES-based application at the same time. Furthermore, this chapter shows two available methods to extend the framework's current capabilities, either by providing an individual implementation of the various services specified by the framework or by adapting the actual framework specification itself. However, this chapter and this work in general assume that the reader is familiar with OSGi and therefore no further explanatory material on this topic will be provided. More precisely, the reader is expected to recognize the OSGi key concept of being an execution environment for applications and services. In addition the reader should be able to develop OSGi bundles represented by jar archive files with OSGi extended manifest information and also be able to maintain these bundles within the OSGi lifecycle including installation, start, stop and un-installation. This chapter solely focuses on giving an essential outline of the development of HERMES-based application and the possibilities to expand the framework's current abilities.

4.1 Sample Application

Apart from the reference implementation of the HERMENS framework, an additional sample application was developed within the scope of this thesis. The main purpose behind this was to demonstrate the framework's capabilities. However, the sample application ideally serves the goal of illustrating the development of HERMES-based applications too.

Figure 4.1 shows the sample application's graphical user interface, built with Java Swing and containing various controls, wrapping the application's sample functionality. The controls have been bound to some key concepts of the HERMES framework, namely contact management, inputs and outputs, communication-related actions, communication tools and finally database management. The user interface allows creation and deletion of contacts with one or two identities for different protocols by interacting with the framework's ***ContactService*** reference implementation. The sample application provides several communication-related actions for each contact, such as calling a contact via Skype, composing an email in the host system's default email client and sending a text message via gateway. Each of these actions is routed to the corresponding custom tool implementation. The sample application showcases simple utilization of inputs and outputs using a graphical user interface as IO method by implementing the framework's ***IOService*** Java interface. Additionally, the user interface

contains a text area that is automatically notified and displays all contacts' and all HERMES embedded database's modifications by implementing the framework's ***ContactObserverService*** and ***DatabaseObserverService*** Java interfaces. Finally, the user interface also includes a further text area that monitors the host environment's standard output and standard error output streams and displays messages and error messages sent to these streams. The sample application may be executed at multiple points in distributed heterogeneous networks at the same time and the HERMES framework will autonomously take care of the entire data replication. In this case, the framework will replicate all created and deleted contacts at all peers involved.

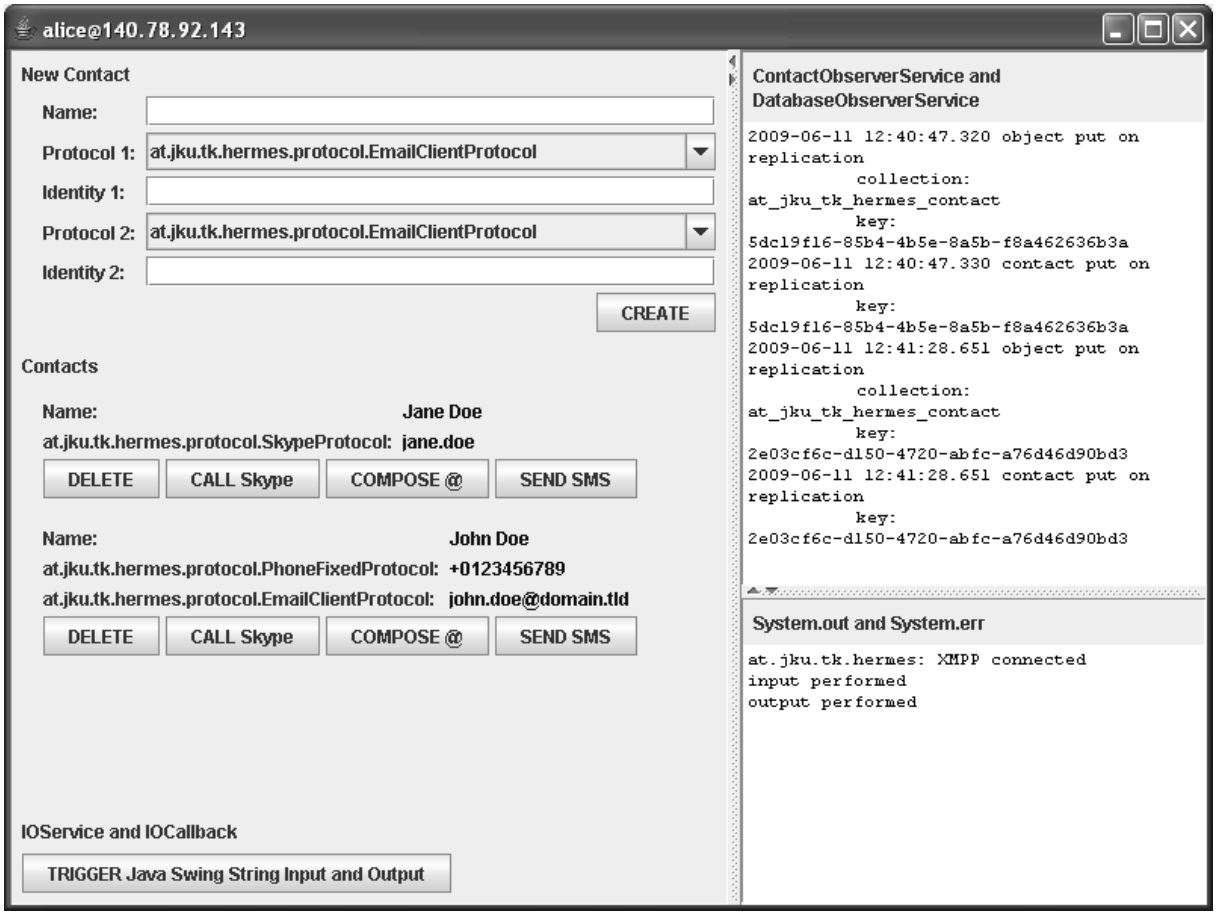

Figure 4.1: Sample application's graphical user interface.

The sample application's graphical user interface has been implemented in the ***at.jku.tk.hermes.sample.ui.swing*** OSGi bundle. The sample application's inputs and outputs utilization has been implemented in the ***at.jku.tk.hermes.sample.io.swing*** OSGi bundle. However, the screenshot does not show the additional components that have been developed for the sample application like custom sensors, a custom rule base and custom tools.

Firstly, the sample application contains two custom implementations of the framework's ***SensorService*** Java interface. The first sample sensor simply works as a timestamp beacon broadcasting the system's current timestamp every second, without the need to be triggered. Events generated by this sensor are published with the topic ***at/jku/tk/hermes/sample/sensor/time/event*** and the actual timestamp is set as an event property. This sensor was implemented in the ***at.jku.tk.hermes.sample.sensor.time*** OSGi bundle. The second sample service fetches the Google Calendar service for all currently active calendar entries in a consistent interval. By contrast, this sensor does not publish events to the HERMES framework autonomously but only on demand and must be manually triggered to do so. The events generated by this Google Calendar sensor are published with the topic ***at/jku/tk/hermes/sample/sensor/google/calendar/event*** and the calendar entries are set as an event property. This sensor was implemented in the ***at.jku.tk.hermes.sample.sensor.google.calendar*** OSGi bundle. In order to interact with the Google Calendar service's API, the sample application is required to incorporate the Google Data APIs, which provide a standard protocol for reading and writing data on the internet. The entire Google Data API library was included in the ***com.google.gdata*** OSGi bundle.

Once the presented custom sensors started to publish events, the sample application required a component to process these events and eventually execute some communication-related actions. Therefore, in a second step, the sample application was equipped with a very straightforward rule base implementing the ***RuleBaseService*** Java interface. The sample application contains only one rule base that responds to events coming form the Google Calendar sensor. Since this sensor publishes events only on demand, the rule bases triggers it to do so in a consistent interval. Upon receiving events from the sensor, the rule base is programmed to forward only new calendar entries to a specific sample contact using the communication tool Skype or to start a conference call between specific sample contacts also using Skype whenever an entry's label contains the string ***conference***. This procedure illustrates an elementary communication-related algorithm. In fact, this rule base implementation executes communication-related actions based on the analyzed data but does not incorporate any sophisticated ability to learn from collected knowledge about users' communication needs and habits, because the purpose of this rule base was primarily to prove a concept and demonstrate its functionality. Any further enhancement towards techniques fully exploiting the framework's potential would be an overwhelming task and beyond the scope of this work. This rule base was implemented in the ***at.jku.tk.hermes.sample.rulebase*** OSGi bundle.

As previously mentioned, the sample application has been enhanced by various communication tools. The first in this set is the Skype communication tool. It is basically a wrapper for the actual Skype

application implementing the ***ToolService*** Java interface and utilizing the official Skype API library. The Skype tool supports multiple communication-related actions, predefined by the HERMES framework, which are also supported by the actual Skype application itself. This tool was implemented in the ***at.jku.tk.hermes.sample.tool.skype*** OSGi bundle. Further communication tools include a tool enabled to open the host system's default email client in order to compose emails implemented in the ***at.jku.tk.hermes.sample.tool.email.compose*** OSGi bundle, a tool enabled to send emails via SMTP implemented in the ***at.jku.tk.hermes.sample.tool.email.smtp*** OSGi bundle, and finally a tool enabled to send text messages via a gateway implemented in the ***at.jku.tk.hermes.sample.tool.sms.com.wekay*** OSGi bundle.

4.2 Custom Components

The sample application demonstrates the main way of utilizing the HERMES framework in order to develop applications, namely by implementing custom components in the shape of OSGi bundles. The framework exposes sensors, rule bases, tools and IO services by means of the ***SensorService***, ***RuleBaseService***, ***ToolService***, and ***IOService*** Java interfaces in their corresponding Java packages ***at.jku.tk.hermes.sensor***, ***at.jku.tk.hermes.rulebase***, ***at.jku.tk.hermes.tool*** and ***at.jku.tk.hermes.io***. Custom components are simply developed by implementing these Java interfaces and packing the implementations into OSGi bundles. The bundles have to be installed and started inside an OSGi R4 core framework specification compliant implementation, such as Apache Felix or Eclipse Equinox, which as a matter of course has to run the HERMES framework bundle. Classes implementing these interfaces must be also registered as a ***SensorService***, a ***RuleBaseService,*** a ***ToolService*** or an ***IOService*** service along with the OSGi service registry. The greatest unique advantage derived from this approach or from SOA generally is that for each of these services exposed by the HERMES framework, multiple hot-swappable implementations may exist, also within a single OSGi instance at the same time. The sample application presented in this chapter contains custom implementations of all four Java interfaces. All of them are examined in this sub-chapter in order to provide a comprehensive view of the HERMES framework utilization by implementing custom components in the shape of OSGi bundles.

The sample application contains a ***SensorService*** implementation in the ***at.jku.tk.hermes.sample.sensor.google.calendar*** OSGi bundle. This sensor fetches the Google Calendar service for all currently active calendar entries. Such a component generally requires the implementation of two Java interfaces. Firstly, an OSGi bundle usually contains a so-called bundle activator, a Java class to be called up once a bundle is activated. Activators are generally responsible for performing initial tasks, like instantiating service implementations and registering these in the

OSGi service registry. Secondly, the actual service needs to be implemented. Listing 4.1 shows a fragment of the sensor's bundle activator instantiating and registering the custom ***SensorService*** whose fragment is shown in Listing 4.2. The sensor is registered without any additional properties.

```
// package statement and imports
public class Activator implements BundleActivator{
  public void start(BundleContext context) throws Exception {
    context.registerService(SensorService.class.getName(), new SensorServiceImpl(), null);
  }
  // further implementation
}
```

Listing 4.1: Fragment of sample SensorService bundle's activator.

The ***SensorService*** implementation, partially presented in Listing 4.2, demonstrates how this sample sensor operates. The sensor's constructor sets up a task repetitively fetching the Google Calendar service in a specified interval by utilizing the Google Data API library. The Google Calendar service is fetched for all calendar entries within the last 24 hours and ending now or in future. The task stores all received calendar entries in a list.

```
// package statement and imports
public class SensorServiceImpl implements SensorService {
  private final List<CalendarEventEntry> calendarEventEntries =
      new ArrayList<CalendarEventEntry>();
  public SensorServiceImpl() {
    addSupportedEventTopic("at/jku/tk/hermes/sample/sensor/google/calendar/event");
    final CalendarService calendarService = new CalendarService("atJkuTk-hermesSensor-1");
    calendarService.setUserCredentials(
        System.getProperty("at.jku.tk.hermes.sample.sensor.google.calendar.username"),
        System.getProperty("at.jku.tk.hermes.sample.sensor.google.calendar.password"));
    final CalendarQuery calendarQuery = new CalendarQuery(
        new URL("http://www.google.com/calendar/feeds/" +
            System.getProperty("at.jku.tk.hermes.sample.sensor.google.calendar.username") +
            "/private/full"));
    new Timer(true).scheduleAtFixedRate(new TimerTask() {
      public void run() {
        calendarEventEntries.clear();
        calendarQuery.setMinimumStartTime(
            new DateTime(System.currentTimeMillis() - 24 * 60 * 60 * 1000));
        calendarQuery.setMaximumStartTime(DateTime.now());
        CalendarEventFeed calendarEventFeed =
            calendarService.query(calendarQuery, CalendarEventFeed.class);
        for (CalendarEventEntry calendarEventEntry : calendarEventFeed.getEntries()) {
          for (When when : calendarEventEntry.getTimes()) {
```

```
            if (when.getEndTime().compareTo(DateTime.now()) >= 0) {
              calendarEventEntries.add(calendarEventEntry);
              break;
            }}}},
      0, 60000);
  }
  // further implementation
}
```

Listing 4.2: Fragment of sample SensorService implementation.

As mentioned earlier, the Google Calendar sensor does not publish events to the HERMES framework autonomously but only on demand. The sample ***RuleBaseService*** implementation calls the sensor's ***executePublishing()*** method in order to receive all currently active calendar entries. Listing 4.3 shows the ***executePublishing()*** method implementation. When called, the method publishes an event with the topic ***at/jku/tk/hermes/sample/sensor/google/calendar/event*** and the calendar entries included in the event properties utilizing the OSGi Event Admin system service.

```
public void executePublishing() {
  EventProperties eventProperties = new EventProperties(
      "at.jku.tk.hermes.sample.sensor.google.calendar");
  eventProperties.setEvent(calendarEventEntries);
  Event event = new Event(
      "at/jku/tk/hermes/sample/sensor/google/calendar/event",
      eventProperties.toProperties()));
  EventAdmin eventAdmin =
      (EventAdmin) Activator.getDefault().getEventAdminServiceTracker().getService();
  eventAdmin.sendEvent(event);
}
```

Listing 4.3: Sample SensorService's executePublishing() method implementation.

Additional to the ***SensorService*** implementations, the sample application contains a ***RuleBaseService*** implementation in the ***at.jku.tk.hermes.sample.rulebase*** OSGi bundle. This rule base fetches all currently active Google Calendar entries from the sample Google Calendar sensor. Listing 4.4 shows a fragment of the rule base's bundle activator instantiating and registering the custom ***RuleBaseService*** implementation, whose fragment is shown in Listing 4.5. The rule base is registered with an additional property, namely the event topic ***at/jku/tk/hermes/sample/sensor/google/calendar/event*** because this rule base is only interested in receiving events with this topic through the OSGi Event Admin system service.

```
// package statement and imports
public class Activator implements BundleActivator {
  public void start(BundleContext context) throws Exception {
    Properties properties = new Properties();
    properties.put(EventConstants.EVENT_TOPIC, new String[] {
        "at/jku/tk/hermes/sample/sensor/google/calendar/event" });
    context.registerService(
        EventHandler.class.getName(),
        new RuleBaseServiceImpl(),
        properties);
  }
  // further implementation
}
```

Listing 4.4: Fragment of sample RuleBaseService bundle's activator.

The ***RuleBaseService*** implementation partially presented in Listing 4.5 demonstrates how this very elementary sample rule base operates. Firstly, the rule base's constructor sets up a task repetitively triggering the sample Google Calendar sensor to publish all currently active calendar entries via event in a consistent interval. Due to the event topic being registered along with the rule base in the activator, events published by this sensor are routed by the OSGi Event Admin system service to the rule base's ***handleEvent(Event)*** method. This method puts every received calendar entry into a queue. The rule base's constructor sets up a thread intended to process each calendar entry in the queue as long as it is not empty or the thread is waiting for new calendar entries in the queue. While processing, the thread applies the elementary communication-related algorithm introduced in the previous sub-chapter: Each so far unprocessed calendar entry is either sent to a specific sample contact via the communication tool Skype or causes a conference call between specific sample contacts also via Skype whenever the entry's label contains the string ***conference***.

```
// package statement and imports
public class RuleBaseServiceImpl implements RuleBaseService {
  private LinkedBlockingQueue<CalendarEventEntry> calendarEventEntryQueue =
      new LinkedBlockingQueue<CalendarEventEntry>();
  private List<String> processedCalendarEventEntryIds = new ArrayList<String>();
  public RuleBaseServiceImpl() {
    addSupportedEventTopic(EventTopic.getInstance().getEventTopic());
    new Timer(true).scheduleAtFixedRate(new TimerTask() {
      public void run() {
        ((SensorService) Activator.getDefault().
            getSensorServiceGoogleCalendarTracker().getService()).executePublishing();
      }, 1000, 0);
    Thread thread = new Thread(new Runnable() {
      public void run() {
```

```
        while (true) {
          CalendarEventEntry calendarEventEntry = calendarEventEntryQueue.take();
          if (! processedCalendarEventEntryIds.contains(calendarEventEntry.getId())) {
            Contact contactOne = new Contact("Hermes One");
            contactOne.putIdentity(SkypeProtocol.class, "at.jku.tk.hermes.contact.1");
            Contact contactTwo = new Contact("Hermes Two");
            contactTwo.putIdentity(SkypeProtocol.class, "at.jku.tk.hermes.contact.2");
            Action action;
            if (calendarEventEntry.getTitle().
                getPlainText().toLowerCase().contains("conference")) {
              action = new AudioConferenceStartAction(contactOne, contactTwo);
            } else {
              // message is prepared
              // String message = ...
              action = new MessageSendToOneAction(contactOne, message);
            }
            ((ToolService) Activator.getDefault().getToolServiceSkypeTracker().
                getService()).executeAction(SkypeProtocol.class, action);
            processedCalendarEventEntryIds.add(calendarEventEntry.getId());
          }}}});
    thread.setDaemon(true);
    thread.start();
  }
  public void handleEvent(Event event) {
    for (CalendarEventEntry calendarEventEntry :
      (List<CalendarEventEntry>) event.getProperty(EventConstants.EVENT)) {
        calendarEventEntryQueue.put(calendarEventEntry);
      }}
  }
  // further implementation
}
```

Listing 4.5: Fragment of sample RuleBaseService implementation.

Apart from ***SensorService*** implementations and a ***RuleBaseService*** implementation, the sample application contains a ***ToolService*** implementation in the ***at.jku.tk.hermes.sample.tool.skype*** OSGi bundle. This tool is a wrapper for the actual Skype application supporting multiple communication-related actions. Listing 4.6 shows a fragment of the tool's bundle activator instantiating and registering the custom ***ToolService*** implementation whose fragment is shown in Listing 4.7. The tool is registered without any additional properties.

```
// package statement and imports
public class Activator implements BundleActivator {
  public void start(BundleContext context) throws Exception {
    context.registerService(ToolService.class.getName(), new ToolServiceImpl(), null);
  }
  // further implementation
}
```

Listing 4.6: Fragment of sample ToolService bundle's activator.

The ***ToolService*** implementation partially presented in Listing 4.7 demonstrates how this sample tool operates. Communication-related actions generated in the sample rule base are passed along with the protocol to the tool's ***executeAction(Class<? extends Protocol>, Action)*** method. Firstly, this method checks the passed protocol. If the protocol can be handled, the method continues, if not an exception is thrown. The method then checks the passed action. If the action can be handled the method continues, otherwise an exception is thrown. The actual handling is carried out in further methods where the service interacts with the Skype API library. Listing 4.7 shows the methods ***audioConferenceStart(Class<? extends Protocol>, AudioConferenceStartAction)*** and ***callOrConferenceStart(Class<? extends Protocol>, List<Contact>, boolean)*** whereby the latter interacts with the communication tool Skype by setting off a proper call according to the parameters and throwing an exception on failure.

```
// package statement and imports
public class ToolServiceImpl implements ToolService {
  public boolean supportsProtocol(Class<? extends Protocol> clazz) {
    return clazz == SkypeProtocol.class || clazz == SkypePstnProtocol.class;
  }
  public void executeAction(Class<? extends Protocol> clazz, Action action)
      throws ProtocolNotSupportedByToolException, ActionNotSupportedException,
      ActionExecutionFailedException {
    if (! supportsProtocol(clazz)) {
      throw new ProtocolNotSupportedByToolException();
    }
    if (action instanceof AudioCallStartAction) {
      audioCallStart(clazz, (AudioCallStartAction) action);
    } else if (action instanceof AudioConferenceStartAction) {
      audioConferenceStart(clazz, (AudioConferenceStartAction) action);
    // further actions
    } else {
      throw new ActionNotSupportedException();
    }
  }
  private void audioConferenceStart(Class<? extends Protocol> clazz,
```

```
      AudioConferenceStartAction action) throws ActionExecutionFailedException {
    callOrConferenceStart(clazz, action.getContacts(), false);
  }
  private void callOrConferenceStart(Class<? extends Protocol> clazz, List<Contact> contacts,
      boolean isVideoEnabled) throws ActionExecutionFailedException {
    List<String> usernames = new ArrayList<String>();
    for (Contact contact : contacts) {
      usernames.add(contact.getIdentity(clazz));
    }
    try {
      Call call = Skype.call(stringListToArray(usernames));
      call.setReceiveVideoEnabled(isVideoEnabled);
      call.setSendVideoEnabled(isVideoEnabled);
    } catch (Exception e) {
      throw new ActionExecutionFailedException(e.fillInStackTrace().getCause());
    }
  }
  // further implementation
}
```

Listing 4.7: Fragment of sample ToolService implementation.

Finally, the sample application also contains an ***IOService*** implementation in the ***at.jku.tk.hermes.sample.io.swing*** OSGi bundle. This IO service allows inputs and outputs of strings via the sample application's graphical user interface. Listing 4.8 shows a fragment of the IO service's bundle activator instantiating and registering the custom ***IOService*** implementation whose fragment is shown in Listing 4.9. The IO service is registered without any additional properties.

```
// package statement and imports
public class Activator implements BundleActivator {
  public void start(BundleContext context) throws Exception {
    context.registerService(IOService.class.getName(), IOServiceImpl.getInstance(), null);
  }
  // further implementation
}
```

Listing 4.8: Fragment of sample IOService bundle's activator.

The ***IOService*** implementation partially presented in Listing 4.9 demonstrates how this sample IO service operates. Inputs and outputs are executed in the methods ***executeInput(Input<?>, IOCallback)*** and ***executeOutput(Output<?>, IOCallback)***. When called, these methods check the passed input and output respectively. If the input or output can be handled the methods continue, otherwise an

exception is thrown. The actual input or output is carried out using Java Swing widgets and in case of an input the result is returned to the ***IOCallback*** passed as parameter before.

```
// package statement and imports
public class IOServiceImpl implements IOService {
  public void executeInput(final Input<?> input, final IOCallback callback)
      throws InputNotSupportedException {
    if (input instanceof StringInput) {
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          // Swing widgets are prepared
          dialog.setVisible(true);
          if (((Integer) optionPane.getValue()).intValue() == JOptionPane.OK_OPTION) {
            ((StringInput) input).setInput(textField.getText());
            if (((StringInput) input).getInput() != null &&
                ! ((StringInput) input).getInput().equals("")) {
              callback.onInput(input);
            } else {
              callback.onInputExecutionFailed();
            }}}});
    } else {
      throw new InputNotSupportedException();
    }
  }
  public void executeOutput(final Output<?> output, final IOCallback callback)
      throws OutputNotSupportedException {
    if (output instanceof StringOutput) {
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          JLabel label = new JLabel(((StringOutput) output).getOutput());
          // Swing widgets are prepared
          dialog.setVisible(true);
          callback.onOutput();
        }});
    } else {
      throw new OutputNotSupportedException();
    }
  }
  // further implementation
}
```

Listing 4.9: Fragment of sample IOService implementation.

4.3 Framework Extension

The previous sub-chapter provided a comprehensive view of how to develop HERMES-based applications by implementing custom components in the shape of OSGi bundles. This approach leaves

the framework or more specifically the ***at.jku.tk.hermes*** OSGi bundle completely untouched and ensures great modularity and flexibility. However, circumstances may arise where it is not enough to just implement the Java interfaces exposed by the framework but to actually extend the framework specification itself. For example, it may be necessary to introduce new inputs and outputs not covered by the predefined input and output implementations for basic Java data types currently provided by the HERMES framework reference implementation.

Implementing the following custom HERMES framework components would result in a modified ***at.jku.tk.hermes*** OSGi bundle and thus in a modified HERMES framework: inputs, outputs, actions and protocols. The framework exposes these components in the shape of the ***Input***, ***Output***, ***Action***, and ***Protocol*** Java interfaces in their corresponding Java packages ***at.jku.tk.hermes.io.input***, ***at.jku.tk.hermes.io.output***, ***at.jku.tk.hermes.action***, and ***at.jku.tk.hermes.protocol***. Due to the nature of the underlying Java-based OSGi service platform, it is not possible to handle these components in the same manner as sensors, rule bases, tools, and IO services where custom components are made available in the shape of separate OSGi bundles. However, it is still very simple to implement custom inputs, outputs, actions, and protocols when accepting the fact that this requires an extension of the framework specification itself. This sub-chapter explains how new inputs, outputs, actions and protocols may be shaped.

Expanding the predefined input implementations provided by the current HERMES framework reference implementation requires the creation of a new Java class, implementing the ***Input*** Java interface exposed by the framework. The ***Input*** implementation presented in Listing 4.10 demonstrates how such a custom class may be constructed. This particular implementation symbolizes a wrapper object for a string input and is also able to carry the actual input value and additional metadata. The class implements the methods specified by the interface and stores the metadata and the actual input string in its member fields. Custom ***Input*** implementations are supposed to be placed in the framework OSGi bundle's ***at.jku.tk.hermes.io.input*** package since this package is exposed by the bundle to the entire OSGi environment by default and thus the new class may be used by other bundles instantly.

```
package at.jku.tk.hermes.io.input;
import at.jku.tk.hermes.core.MetaData;
public class StringInput implements Input<String> {
  protected MetaData metaData = new MetaData();
  protected String input;
  public StringInput() {
    metaData.setName("String Input");
  }
```

```
  public MetaData getMetaData() {
    return metaData;
  }
  public void setMetaData(MetaData metaData) {
    this.metaData = metaData;
  }
  public String getInput() {
    return input;
  }
  public void setInput(String input) {
    this.input = input;
  }
}
```

Listing 4.10: Custom Input implementation for string inputs.

Expanding the predefined output implementations provided by the current HERMES framework reference implementation requires creating a new Java class implementing the ***Output*** Java interface exposed by the framework. The ***Output*** implementation presented in Listing 4.11 demonstrates how such a custom class may look like. This particular implementation symbolizes a wrapper object for a string output and is also able to carry the actual output value and additional metadata. The class implements the methods specified by the interface and stores the metadata and the actual output string in its member fields. Custom ***Output*** implementations are supposed to be placed in the framework OSGi bundle's ***at.jku.tk.hermes.io.output*** package since this package is exposed by the bundle to the entire OSGi environment by default and thus the new class may be used by other bundles instantly.

```
package at.jku.tk.hermes.io.output;
import at.jku.tk.hermes.core.MetaData;
public class StringOutput implements Output<String> {
  protected MetaData metaData = new MetaData();
  protected String output;
  public StringOutput() {
    metaData.setName("String Output");
  }
  public MetaData getMetaData() {
    return metaData;
  }
  public void setMetaData(MetaData metaData) {
    this.metaData = metaData;
  }
  public String getOutput() {
    return output;
  }
  public void setOutput(String output) {
```

```
    this.output = output;
  }
}
```

Listing 4.11: Custom Output implementation for string outputs.

Expanding the predefined communication-related actions implementations provided by the current HERMES framework reference implementation requires creating a new Java class implementing the ***Action*** Java interface exposed by the framework. The ***Action*** implementation presented in Listing 4.12 demonstrates how such a custom class may be constructed. This particular implementation symbolizes a communication-related intent for sending a message to a single contact. The class implements the interface that actually does not specify any methods to implement. However, the class stores the contact and the message required to fulfill the intent in its member fields and offers corresponding getter and setter methods. Custom ***Action*** implementations are supposed to be placed in the framework OSGi bundle's ***at.jku.tk.hermes.action*** package since this package is exposed by the bundle to the entire OSGi environment by default and thus the new class may be used by other bundles instantly.

```
package at.jku.tk.hermes.action;
import at.jku.tk.hermes.contact.Contact;
public final class MessageSendToOneAction implements Action {
  protected Contact contact;
  private String message;
  public MessageSendToOneAction(Contact contact, String message) {
    setContact(contact);
    setMessage(message);
  }
  public Contact getContact() {
    return contact;
  }
  public void setContact(Contact contact) {
    this.contact = contact;
  }
  public String getMessage() {
    return message;
  }
  public void setMessage(String message) {
    this.message = message;
  }
}
```

Listing 4.12: Custom Action implementation for sending a message to a single contact.

Expanding the predefined protocol implementations provided by the current HERMES framework reference implementation requires creating a new Java class implementing the ***Protocol*** Java interface exposed by the framework. The ***Protocol*** implementation presented in Listing 4.13 demonstrates how such a custom class may look like. This particular implementation symbolizes a protocol for a text message gateway as a communication tool. The class implements the methods specified by the interface and stores the communication-related actions it supports in its member field. Custom ***Protocol*** implementations are supposed to be placed in the framework OSGi bundle's ***at.jku.tk.hermes.protocol*** package since this package is exposed by the bundle to the entire OSGi environment by default and thus the new class may be used by other bundles instantly.

```
package at.jku.tk.hermes.protocol;
import at.jku.tk.hermes.action.Action;
import at.jku.tk.hermes.action.MessageSendToManyAction;
import at.jku.tk.hermes.action.MessageSendToOneAction;
import java.util.ArrayList;
import java.util.List;
public final class SMSGatewayProtocol implements Protocol{
  protected final List<Class<? extends Action>> supportedActions =
      new ArrayList<Class<? extends Action>>();
  public SMSGatewayProtocol() {
    supportedActions.add(MessageSendToOneAction.class);
    supportedActions.add(MessageSendToManyAction.class);
  }
  public boolean supportsAction(Class<? extends Action> clazz) {
    return supportedActions.contains(clazz);
  }
  public List<Class<? extends Action>> getSupportedActions() {
    return supportedActions;
  }
}
```

Listing 4.13: Custom Protocol implementation for a SMS gateway as communication tool.

5 Conclusion

The objective of this thesis was to design and implement a framework for integrated communication management. Applications developed on the basis of this framework should be able to know about users' communication needs and, depending on a set of parameters, present appropriate communication tool support. Furthermore, these applications should be able to learn from experience, connect to sensors and interact with standard existing communication infrastructure. The previous chapters discussed the HERMES framework as the outcome of efforts to put that objective into practice.

The idea for an integrated communication management framework was substantially inspired by the work contributed in [1], in which the authors refer to a paradigm described as device independent communication and reachability or simply ubiquitous connectivity. Ubiquitous connectivity can be defined as technology assisting nomadic users' communications regardless of their location, underlying communication protocol, and communication tools. In order to understand the best approach to design and implement a complex framework like HERMES, a wide spectrum of related work had to be taken into consideration. Consequently, this work contains influences from contributions of various domains, most notably device independent connectivity, communication management and contact management, service-oriented architectures, event-driven architectures, peer-to-peer architectures, instant messaging, and data replication and reconciliation.

The presented HERMES framework demonstrates the fulfillment of this thesis' objective influenced by the ideas and results of the related works. HERMES was designed in compliance with the service-oriented architecture and the reference implementation was built on top of the Java-based OSGi service platform. The entire framework is split into numerous communication-related components, consisting of sensors, events, rule bases, inputs, outputs, actions, protocols, and tools. Additionally, the framework contains an embedded relational database intended to persist any kind of XML-serializable data. Partitioning the framework into these components helped to emphasize the service-oriented paradigm and simplified the extremely complex task of dissipating integrated communication management into manageable chunks of communication-related entities and jobs while strengthening modularity and flexibility at the same time. From the very beginning, the HERMES framework was designed to enable applications to operate and interact in distributed heterogeneous networks simultaneously. Intra-framework communication between components is mostly carried out by events. By contrast, inter-framework communication between distributed HERMES instances and especially data replication were accomplished by employing peer-to-peer architecture. The framework utilizes

the XMPP communication platform as a hybrid decentralized unstructured P2P technology. XMPP enables HERMES-based applications to directly open instant messaging channels between one another. These channels are then used for full lazy non-optimistic multi-master replication of the applications' databases. The replication mechanism itself uses logging, timestamping, and a custom reconciliation strategy. As already mentioned several times in this work, the data replication is performed by the framework autonomously, canceling the need of any user interaction. The previous chapter illustrated that utilization of the HERMES framework for real-world applications is already possible in its current state and fairly straightforward, as demonstrated by the sample application.

In summary, it can be said that the HERMES integrated communication management framework and the full lazy non-optimistic multi-master replication utilizing the XMPP communication platform are the two major contributions of this thesis.

The results of this work leave no doubt that integrated communication management is practicable. The basic communication-management-related functionalities provided by the HERMES framework were shown on the basis of the implemented sample application. Furthermore, during tests, the sample application delivered fully satisfying results regarding data replication. The anticipated full lazy non-optimistic multi-master replication worked out of the box even when multiple instances of the sample application were deployed in distributed heterogeneous networks. However, it was not possible to explore some of the framework's possibilities in-depth within the scope of the thesis. More precisely, the aforementioned rule base components were introduced as magic boxes intended to perform the miracles of an integrated communication management system. Highly advanced rule bases making use of specialized learning algorithms or advanced technologies like artificial intelligence were suggested, but their full evaluation and actual utilization is an immense task beyond the scope of this work and yet to discover. It should also be noted that, although integrated communication management is an extraordinarily exciting research topic, this thesis may only serve as a foundation for further advancements, including extended superior HERMES-based applications, evaluations, empirical studies, and research on integrated communication management in general.

Some suggestions on potential future work shall conclude this thesis. Firstly, improving the current HERMES framework specification and reference implementation should be the top priority. HERMES contains numerous critical areas requiring further enhancement. The previous chapter mentioned that the HERMES framework contains the components input, output, action, and protocol, which when extended, also require the actual framework specification to be extended. This is due to the nature of the underlying Java-based OSGi service platform. Ways to overcome this limitation – for example

using the Eclipse Equinox OSGi R4 core framework implementation and Eclipse Extension Points – and to handle these components in the same way as sensors, rule bases, tools, and IO services, could be researched. Another topic with a large amount of potential for improvement is HERMES' data replication. The current specification and reference implementation is more or less a proof of concept. The entire replication process is pull-based. It should be analyzed how this approach influences the network load and the overall performance of HERMES-based systems deployed at large scale and if a push-based approach or even a combined pull-push-based approach might not improve the overall performance. Furthermore, the replication process could be split into multiple steps in order to optimize and increase efficiency. Currently, the metadata are instantly transmitted between replicas along with the actual data objects. Instead, the replication process could first request the comparatively smaller metadata and in a second step decide whether it makes sense to request the data object – for example in cases when the timestamp of the remote data object is higher compared to the local timestamp – or to skip and proceed. Additionally and in an intermediate step, the replication process could fetch the timestamps for a specific data object from all peers to learn which replica can deliver the most recent data object. However, such an intermediate step would require a more sophisticated algorithm not to mention the higher network load and should be analyzed regarding its overall performance improvement. Eventually, the entire replication process could be divided into multiple, concurrent threads especially in multi-processor-core environments facilitating faster replication but increasing network load at the same time. Finally, it should investigated whether the current static replication tribes handling could be refactored in a way, that enables peers to dynamically join and leave any number of replication tribes at any time. For this purpose, again the XMPP communication platform could be used. The addresses of fellow peers could be managed in a peer's contact list, called ***roster***, and multiple dynamic replication tribes could be reflected in roster groups utilizing the XMPP extension XEP-0083 [35].

Apart from improving the current framework's state, extending it offers an additional pool for future work. In general, the topic of integrated communication management should be further researched. There are certainly other exciting paths to approach this complex task than by means of a service-oriented framework like HERMES and it would be thrilling to see them in comparison with HERMES. One such approach is the application Locale [36], developed for the Android operating system. Locale allows defining location-based rules that influence phone settings whereby the application is expandable by plug-ins like HERMES. While Locale's functionality mirrors a fraction of integrated communication management, it is imaginable to port and use it inside a custom HERMES rule base. Within the domain of this thesis and, more precisely, the HERMES framework, the first topic to approach scientifically should be rule bases, as mentioned before. Currently it is not determined what

kind of rule bases are applicable and actually make sense for the framework. Test should be run with some sophisticated rule bases utilizing learning algorithms, for example. Furthermore, as a matter of course the Java-based OSGi service platform – on top of which the HERMES reference implementation was developed – is not the be-all and end-all as illustrated by [6]. The author criticizes OSGi's failure in providing full reliability especially for network, devices, and applications due to its limited support of SOA's loose coupling and late binding. The work continues to analyze OSGi's software reliability issues and proposes a proxy-based reliable extension for OSGi. This extension could be embedded into the HERMES framework, in which reliability plays a major role. Distributed OSGi is a further enhancement of the OSGi service platform introduced in the upcoming OSGi 4.2 specification [37]. According to [37], distributed OSGi intends to introduce a minimal set of distributed computing functionality to OSGi and seeks distribution of OSGi services including service discovery and access to and from external environments. This is definitely something that should undergo future work in conjunction with the HERMES framework. A lightweight application discovering and providing communication-related services like sensor or tools globally and on demand is just one imaginable use case scenario for a HERMES-based application backed by distributed OSGi. An evaluation of this technology blend in terms of architecture, performance, reliability, and usability presents an inspiring scientific challenge.

All in all, the HERMES framework presented in this thesis demonstrates the basic principles of applied integrated communication management and provides a wide open scope for future research.

Bibliography

[1] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, M. Spasojevic. *People, places, things: Web presence for the real world*. In: Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications, 2000, pp. 19-28.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann. *Service-Oriented Computing: State of the Art and Research Challenges*. In: IEEE Computer, Volume 40, Number 11, 2007, pp. 38-45.

[3] P. R. Pietzuch, G. Muhl, L. Fiege. *Distributed Event-Based Systems: An Emerging Community*. In: IEEE Distributed Systems Online, Volume 8, Number 2, 2007, pp. 2-2.

[4] G. Koloniari, E. Pitoura. *Peer-to-peer management of XML data: issues and research challenges*. In: ACM SIGMOD Record, Volume 34, Issue 2, June 2005, pp. 6-17.

[5] H. Ahn, H. Oh, C. O. Sung. *Towards reliable OSGi framework and applications*. In: Proceedings of the 2006 ACM Symposium on Applied Computing, 2006, pp. 1456-1461.

[6] K. Groth. *A technological framework supporting knowledge exchange in organizations*. In: Proceedings of the Third Nordic Conference on Human-Computer interaction, 2004, pp. 381-384.

[7] Thai, B., Wan, R., and Seneviratne, A. 2001. *Personal Communications in Integrated Personal Mobility Architecture*. In: Proceedings of the Ninth IEEE international Conference on Networks, 2001, pp. 409-414.

[8] B. Thai, R. Wan, A. Seneviratne, T. Rakotoarivelo. *Integrated personal mobility architecture: a complete personal mobility solution*. In: Mobile Networks and Applications, Volume 8, Issue 1, 2003, pp. 27-36.

[9] E. C. Epp. *Relationship Management: Secure Collaboration in a Ubiquitous Environment*. In: IEEE Pervasive Computing, Volume 2, Issue 2, 2003, pp. 62-71.

[10] J. C. Tang, J. Lin, J. Pierce, S. Whittaker, C. Drews. *Recent shortcuts: using recent interactions to support shared activities*. In: Proceedings of the 2007 Conference on Human Factors in Computing Systems, 2007, pp. 1263-1272.

[11] P. K. Biswas, M. Schmiedekamp, S. Phoha. *An agent-oriented information processing architecture for sensor network applications*. In: International Journal of Ad Hoc and Ubiquitous Computing, Volume 1, Issue 3, 2006, pp. 110-125.

[12] R. Nienaber, E. Cloete. *A software agent framework for the support of software project management*. In: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology, 2003, pp. 16-23.

[13] S. Jones. *Toward an Acceptable Definition of Service*. In: IEEE Software, Volume 22, Issue 3, 2005, pp. 87-93.

[14] M. P. Papazoglou, W.-J. Heuvel. *Service oriented architectures: approaches, technologies and research issues*. In: The VLDB Journal, Volume 16, Issue 3, 2007, pp. 389-415.

[15] T. Gu, H. K. Pung, D. Q. Zhang. *Toward an OSGi-Based Infrastructure for Context-Aware Applications*. In: IEEE Pervasive Computing, Volume 3, Issue 4, 2004, pp. 66-74.

[16] V. Dheap, P. A. Ward. *Event-driven response architecture for event-based computing*. In: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research, 2005, pp. 70-82.

[17] The OSGi Alliance. *OSGi Technology*. http://www.osgi.org/About/Technology, last visited on October 10th 2008.

[18] The OSGi Alliance. *OSGi Service Platform Service Compendium*. Release 4, Version 4.1, April 2007.

[19] S. A. Theotokis, D. Spinellis. *A survey of peer-to-peer content distribution technologies*. In: ACM Computing Surveys, Volume 36, Issue 4, 2004, pp. 335-371.

[20] U. Norbisrath, K. Kraaner, E. Vainikko, O. Batrasev. *Friend-to-Friend Computing – Instant Messaging Based Spontaneous Desktop Grid*. In: Proceedings of the Third International Conference on Internet and Web Applications and Services, 2008, pp. 245-256.

[21] F. Calefato, F. Lanubile, M. Scalas. *Porting a distributed meeting system to the Eclipse communication framework*. In: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange, 2007, pp. 46-49.

[22] P. Saint-Andre. *Streaming XML with Jabber/XMPP*. In: IEEE Internet Computing, Volume 9, Issue 5, 2005, pp. 82-89.

[23] V. Martins, E. Pacitti, P. Valduriez. *Survey of data replication in P2P systems*. Technical Report, INRIA Rennes, 2006.

[24] M. Tlili, W. Kokou Dedzoe, E. Pacitti, R. Akbarinia, P. Valduriez. *P2P Logging and Timestamping for Reconciliation*. Technical Report, INRIA Rennes, 2008.

[25] The OSGi Alliance. *OSGi Service Platform Core Specification*. Release 4, Version 4.1, April 2007. Available online: http://www.osgi.org/Download/Release4V41.

[26] Sun Microsystems. *Java Specification Request 220 – Enterprise JavaBeans 3.0*. Version 3.0, Final Release, May 2006. Available online: http://www.jcp.org/en/jsr/detail?id=220.

[27] Codehouse. *XStream*. Available online: http://xstream.codehaus.org/. Retrieved in January 2009.

[28] Apache Software Foundation. *Xindice*. Available online: http://xml.apache.org/xindice. Retrieved in February 2009.

[29] Sun Microsystems. *Java Specification Request 221 – JDBC 4.0 API Specification*. Final v1.0, November 2006. Available online: http://www.jcp.org/en/jsr/detail?id=221.

[30] Apache Software Foundation. *Apache Derby*. Available online: http://db.apache.org/derby. Retrieved in January 2009.

[31] Ignite Realtime. *Smack API*. Available online: http://www.igniterealtime.org/projects/smack/index.jsp. Retrieved in March 2009.

[32] Ignite Realtime. *Openfire Server*. Available online: http://www.igniterealtime.org/projects/openfire/index.jsp. Retrieved in March 2009.

[33] Sun Microsystems. Java Platform, Standard Edition 6 API Specification. Available online: http://java.sun.com/javase/6/docs/api/java/lang/System.html#currentTimeMillis(). Retrieved in March 2009.

[34] XMPP Standards Foundation. *XEP-0055: Jabber Search*. Version 1.2, March 2004. Available online: http://xmpp.org/extensions/xep-0055.html.

[35] XMPP Standards Foundation. *XEP-0083: Nested Roster Groups*. Version 1.0, October 2004. Available online: http://xmpp.org/extensions/xep-0083.html.

[36] two forty four a.m. LLC. *Locale*. Available online: http://www.twofortyfouram.com/product.html. Retrieved in June 2009.

[37] The OSGi Alliance. *OSGi Service Platform Release 4*. Version 4.2, Early Draft 3, Revision 1.2, March 2009. Available online: http://www.osgi.org/download/osgi-4.2-early-draft3.pdf.