Jadabs – An Adaptive Pervasive Middleware Architecture

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
for the degree of
Doctor of Technical Sciences

presented by
ANDREAS RALPH FREI
Dipl. Informatik-Ingenieur ETH
born October 9th, 1976
citizen of
Widnau/SG, Switzerland

accepted on the recommendation of
Prof. Dr. G. Alonso, examiner
Prof. Dr. K. Aberer, co-examiner
2005
The most profound technologies are those that disappear.  
They weave themselves into the fabric of everyday life  
until they are indistinguishable from it.  

Mark Weiser
Seite Leer / Blank leaf
Abstract

A big challenge in mobile computing is its versatility. Mobile computing is affected by resource constraint devices, unreliable connections, application diversity with different protocols and diverse standards. The result is an inherent heterogeneous infrastructure. Due to the user's mobility the environment for mobile devices is continuously changing. User's mobile devices have to adapt over and over again to the changing environment.

Run-time adaptation was proposed as a mechanism to cope with changing requirements [128]. This idea has been adopted to adapt applications on mobile devices, and allows to adapt applications to the current requirements in a heterogeneous mobile environment. In this thesis, an architecture is proposed which allows to transparently adapt and extend application components with new requirements. The architecture is a lightweight middleware which can be run on small devices like mobile phones, handhelds, gateways or desktop machines. The proposed middleware platform – Jadabs – thereby uses a service-oriented architecture (SOA).

The Jadabs model describes applications by annotating components and services which run on the service-oriented architecture with metadata. The model describes also the platform of the mobile device and the properties of the used SOA. By using this metadata, new dependencies can be evaluated at run-time to adapt the application with the extensions provided or required by the environment. Due to the variety of available mobile devices, a dual approach for the SOA and adaptation mechanism was followed by choosing a segmented-adaptive and a monolithic-dynamic approach. Depending on the system different implementations were done. The segmented-adaptive approach uses a combination of SOA and dynamic aspect-oriented programming. It is mainly used for mobile devices in the range of handhelds and higher. The monolithic-dynamic approach uses the SOA concept for self-contained monolithic containers. The flexibility of adaptation in this approach is achieved through the loading and unloading of monolithic containers. For communication between devices a peer-to-peer infrastructure is used. This enables a communication in decentralized as well as centralized environments. The implementation can be used on small devices like mobile phones up to normal machines and integrates different network technologies like Bluetooth, WiFi, LAN, and so on.

The Jadabs benchmarks look promising and its applicability was shown in different mobile scenarios. Examples range from (i) extending applications with an event system, (ii) autonomous cooperating robots, (iii) an adaptive messenger scenario, and (iv) integration of Web services for mobile phones.
Zusammenfassung


integriert verschiedene Netzwerk-Technologien wie z.B. Bluetooth, WiFi und LAN.

Acknowledgments

This dissertation would have never been finished without the help of many people, to whom I would like to express my sincere gratitude.

First of all, Prof. Gustavo Alonso who gave me the opportunity to work as a researcher on the mobile middleware topic. I also thank him for his good advice keeping me focused on such an interesting research subject. He was an active and motivating supervisor who helped discovering new research venues, and formulating ideas especially when writing papers. He was the ideal supervisor as I could imagine. I would also like to thank Prof. Karl Aberer for accepting to be co-examiner.

This work was also possible through the dedicated work of several master and term-project students and scientific assistants. Thanks to Otmar Caduff, Marco Cicolini, Michael Duller, Jean-Luc Geering, Patrick Grawehr, Kaspar von Gunten, Thomas Heynen, Daniel Käppeli, Franz Maier, Janneth Malibago, René Müller, Jan S. Rellermeyer, Andreas Scherrer, Barbara Scheuner, Maslić Slaviša, Mario Tadey, Francois Terrier, Micha Trautweiler, Markus Vitalini, and Stefan Vogt who contributed to the Jadabs project.

Many thanks to all the past and present ETH colleagues in the IKS group whom I met during my PhD. Without people like Win Bausch, Biörn Biörnstad, Thomas Heinis, Daniel Jönsson, Angela Nicoară, Cesare Pautasso, Christian Plattner, Andrei Popovici, Patrick Stüdi, Martin Wyser and Jianbo Xue a lot of research ideas would have never been discussed. Special thanks to Heidy Schümperlin for her patience and support in correcting my English. She added a great deal to the readability of this thesis.

Many words of thanks to my family, Charlotte, Eduard, and Corinne who supported me in many ways. Finally, I dedicate this thesis to Esther. Her patience and love gave me the strength to surpass the most difficult moments during this time. Thousand thanks for her understanding, support, and encouragement.

Andreas Frei
Zürich, Switzerland
July 2005
Seite Leer / Blank leaf
## Contents

1 Introduction .......................................................... 1  
   1.1 Motivation ...................................................... 1  
   1.2 Contributions ................................................ 2  
   1.3 Structure ...................................................... 4  

2 Architecture Overview ............................................ 5  
   2.1 Introduction ................................................ 5  
   2.2 Middleware Scope ........................................... 6  
      2.2.1 Model Driven Architecture (MDA) ...................... 6  
      2.2.2 Middleware Overview .................................. 6  
   2.3 Mobile Environment Challenges ............................... 8  
   2.4 Jadas Overview .............................................. 10  
      2.4.1 Introduction .......................................... 10  
      2.4.2 Architecture .......................................... 11  

3 Jadas – Model and Service Composition ......................... 15  
   3.1 Introduction ................................................ 15  
   3.2 Jadas-PSM .................................................... 16  
   3.3 Jadas-PSM XML Mapping ..................................... 19  
      3.3.1 OSGi Service Plugin Description (OPD) ............. 20  
      3.3.2 OSGi Bundle Description (OBR) ....................... 21  
      3.3.3 OSGi Bundle Properties ............................... 22  
      3.3.4 OSGi Container Description (OCD) ................... 22  
      3.3.5 OSGi Platform Description (PAD) .................... 22  
   3.4 Dynamic Service Composition ................................ 23  
   3.5 Jadas Service Composition ................................ 23  
      3.5.1 Infrastructure Repository ............................. 25  
      3.5.2 Mobile Peer-to-Peer Repository ....................... 26  
   3.6 Jadas Plugin Loading ....................................... 27  
      3.6.1 Service Manager ....................................... 27  
      3.6.2 Plugin Loader ......................................... 28  
      3.6.3 Bundle Loader ......................................... 30  
      3.6.4 Dynamic Service Composition ......................... 31  

xi
4 Jadabs Middleware Platform

4.1 Introduction ............................................. 35
4.2 Segmented Containers ...................................... 36
  4.2.1 OSGi/Nanning ........................................ 37
  4.2.2 OSGi/PROSE .......................................... 40
  4.2.3 Discussion of dynamic AOP in SOA ..................... 41
4.3 Monolithic Containers ..................................... 42
  4.3.1 mOSGi/CLDC for mobile phones ....................... 42
  4.3.2 mOSGi/.NET ........................................... 46
4.4 Distributed Infrastructures ............................... 48
  4.4.1 Peer-to-Peer .......................................... 49
  4.4.2 JXTA .................................................. 50
  4.4.3 UPnP ................................................. 52
  4.4.4 SIP ................................................... 52
4.5 Jadabs – Distributed Infrastructure .................... 53
  4.5.1 Jadabs-JXME ...................................... 55
4.6 Discussion ................................................... 56

5 Application Scenarios ........................................ 59
5.1 Introduction ............................................... 59
5.2 Event System .............................................. 59
  5.2.1 Architecture .......................................... 60
  5.2.2 Descriptors .......................................... 61
  5.2.3 Benchmarking ........................................ 64
  5.2.4 OSGi/Nanning Benchmark ..................... 65
  5.2.5 Contribution ...................................... 67
5.3 Autonomous Cooperating Robots ......................... 68
  5.3.1 Architecture .......................................... 69
  5.3.2 Prototype Traffic Scenario ................... 72
  5.3.3 Contribution ...................................... 75
5.4 Messenger Scenario ....................................... 76
  5.4.1 Architecture .......................................... 77
  5.4.2 Descriptors .......................................... 77
  5.4.3 Contribution ...................................... 79
5.5 Web Services ............................................... 80
  5.5.1 Architecture .......................................... 81
  5.5.2 Descriptors .......................................... 82
  5.5.3 Contribution ...................................... 84
Chapter 1

Introduction

1.1 Motivation

In his work on ubiquitous computing [176], Mark Weiser in 1991 anticipated that the most profound technologies are those which disappear and which allow the computer to vanish into the background. Mark Weiser predicted the disappearance to happen in two decades. Today, mobile phones and handhelds have become widespread and are becoming the devices users most often carry with them. This makes these devices a preferred target for an obvious platform in which to explore the development of adaptive architectures for ubiquitous computing. Adaptability has already been proposed in [20, 128, 146]. However, the diversity of devices and resource limitations present in small mobile devices, requires a new approach to adaptation.

Current software architectures have been built for infrastructure based environments. These environments consist of powerful machines where resource consumption of an application plays a minor role. Therefore, increasing usage of memory, processor power and network bandwidth is in most cases solved by extending the required resources in a scale up or scale out [127] approach. These architectures with their resource requirements are not applicable to mobile environments. In a mobile environment resource limitations, random network connectivity behaviour, and heterogeneous issues have to be considered. In such environments applications are influenced by available services in its environment and the devices’ capabilities.

An architecture which fits a heterogeneous environment must therefore meet two requirements:

Adaptive which allows the architecture and the application components to be extend as required. The premise is to extend the application transparently with new functionality and if security allows even to the user. An application is started with its required components inside the minimal required middleware. The components and the middleware itself can then be adapted to the changing environment. For example, instead of supporting all possible application protocols, another protocol
can be added and adapt the already running one for the time it is needed to be in synchrony with the other nodes application protocol.

**Pervasive** in the sense that the architecture is aware of changes in mobile environments. This allows us to compose services automatically and adapt other services without the need of the adapted service to be specifically designed for extensions. To be aware of changes in the environment, the architecture should be usable in ad hoc mobile environments without a centralized node as well as in infrastructure based environments.

In this thesis an **adaptive pervasive middleware architecture – Jadabs** – is proposed which supports the ubiquitous computing and therefore the disappearance of technology as stated by Mark Weiser. The architecture has been applied to many of today’s used mobile device platforms and mobile environments.

### 1.2 Contributions

To be able to manipulate all elements of the system, we need to describe the components, platforms, and architecture with metadata. Therefore, in a first step, a component and architecture model, Jadabs model, is used to describe the components and architecture in descriptors. This enables the system to evaluate appropriate run-time extensions before they are instantiated and activated.

By describing all resources with metadata, the system is able to evaluate the application specific needs and platform constraints. So far, other distributed infrastructures do not describe the architecture to such a large extend as proposed in our solution. In other systems, it is assumed that every node has the same architecture and that no coordination of the architecture’s capabilities is required.

From the Jadabs model, it follows that a middleware platform is required which allows the loading and unloading of components. Whereas the loading of components is supported in many systems, unloading is problematic if not handled correctly. In our system we use a service-oriented architecture [158] to support the unloading of components. To support adaptation the service-oriented architecture was extended with dynamic aspect-oriented programming (dAOP).

Finally, to use the system in a mobile environment, an appropriate distributed infrastructure is required. Some peer-to-peer systems have the advantage of being usable in centralized as well as in decentralized infrastructures. Remote method call concepts are not applicable to small devices due to resource constraints and unstable connections. A standardized message-oriented infrastructure is therefore used to seamlessly connect heterogeneous architectures.

This thesis proposes the following three core concepts and combines them in a new adaptive pervasive middleware platform whereby it conforms to standards and contributes to them with extensions.
1.2. Contributions

Dynamic heterogeneous service composition  A new dynamic service composition mechanism is proposed to enable an autonomous dynamic adaptation of running services. Devices which join and leave ad hoc environments need to react to the resources they encounter in these environments. New available services are matched against their own platform, exposed extensions, and required extensions. The service matching is performed on the service metadata. Matching services can then be downloaded and activated automatically. Additional services may be normal services or use the adaptive capabilities of the underlying adaptive platform. To allow dynamic service composition, different resource descriptions are required for devices, platforms, and services.

Lightweight adaptive architecture  This work uses a service-oriented architecture proposed by OSGi [115, 116] as the core service and component architecture. Its clear separation of concerns [166] allows Jadabs to add or remove additional services at run-time.

The contribution lies in the adaptation of the already running services. Instead of stopping a service and restarting its extended service, a more flexible dynamic approach is proposed which allows the extension of any services without any disruption.

In more resource constraint devices, services are regarded as the deployable unit and allow to extend other services with special exposed interfaces over a local loop-back interface. The special exposed interfaces use a locale event mechanism which enables to intercept at the exposed calls.

In less resource constraint devices, exposed interfaces can be dynamically extended with dynamic aspect-oriented programming [113, 117, 129]. The combination of dynamic aspect-oriented programming and service-oriented architecture provides a more efficient adaptation mechanism than with the locale event mechanism used in more resource constraint devices.

Therefore, depending on the devices' capabilities different lightweight adaptive architectures can be applied.

Distributed ad hoc infrastructure  The message-oriented communication mechanism used is based on a peer-to-peer concept. It is based on a message protocol and can be implemented for different platforms, ranging from mobile phones up to desktop machines.

This work contributes to the messaging layer by extending it with dynamic plugins to add additional transport layers. Once new transports are available they can be loaded and activated. Furthermore, the messaging layer has been used to allow marshaling of different protocols like events from an event system [57], instant messenger and Web services.

The resulting adaptive pervasive middleware architecture was implemented for different platforms and languages such as Java and C#. The architecture was deployed on devices like handhelds, mobile phones, gateways, and desktops to show its feasibility in heterogeneous environments.
1.3 Structure

This thesis is organized as follows:

Chapter 2 starts with an overview of the middleware scope and shows why a new middleware architecture is required for the mobile environment. An overview of Jadabs is provided.

Chapter 3 explains in detail the Jadabs component and service model. A graphical as well as an XML mapping is given for the Jadabs model. Furthermore, a dynamic heterogeneous service composition is described which allows a node to dynamically extend its capabilities with new services found in the surrounding computing environment.

Chapter 4 introduces the pervasive middleware architecture and its different implementations. The implementations can be used on different platforms depending on the devices' capabilities. Furthermore, extensions to a distributed infrastructure based on a peer-to-peer system are presented.

Chapter 5 shows different application scenarios where Jadabs was applied. The examples range from an event system extension over a robot infrastructure to a messenger scenario and Web service integration.

Chapter 6 describes the usage of three core parts of Jadabs from the programming point of view. It is a small introduction to get started with writing services for the mobile environment.

Chapter 7 gives an overview of related work in component platforms, distributed and peer-to-peer infrastructures, and dynamic AOP.

Chapter 8 summarizes the contributions of this thesis and discusses future work.
Chapter 2

Architecture Overview

2.1 Introduction

Common applications for mobile devices have been built for stand-alone usage or in combination with specialized server applications. For example the most commonly used remote applications on handhelds like AvantGo, ActiveSync, and MSN Messenger work only with a direct server proxy. These applications are built for one specific configuration or specific settings on the target platform. Changes in the mobile environment are not yet considered in these applications due to the lack of an adaptive architecture.

Container models [40, 110] as it is used in server infrastructures [70, 83] are not yet accepted for mobile devices. The resource limitations on such devices do not allow to run a container model as it is used in infrastructure environments. Therefore, a new adaptive pervasive middleware architecture is needed for today’s and future mobile environments. We first compare related architectures before we present an overview of the proposed Jadabs middleware.

![Figure 2.1: The three major steps in the MDA development process](image)

Figure 2.1: The three major steps in the MDA development process
Chapter 2. Architecture Overview

2.2 Middleware Scope

2.2.1 Model Driven Architecture (MDA)

The Model Driven Architecture [85] has been defined by OMG, the Object Management Group [108]. MDA provides an open, vendor-neutral approach to separate business or application logic from underlying platform technologies. Figure 2.1 shows the separation of business model, technology model, and implementation code. The business model is also regarded as the Platform Independent Model (PIM). It describes the business system, but does not show details of use of the platform. A PIM might consist of enterprise, information and computational distributed viewpoint specifications. A transformation tool then transfers the PIM into the next model.

The Platform Specific Model (PSM) implements the system with the desired architectural qualities from the PIM. The transformation from the PIM into one or several PSMs depends on the requirements of the business system.

Following the PSM an implementation is created, which can be deployed on the different platforms. Again, a transformation tool transfers the PSM into an implementation skeleton.

Different PSM and implementation specifications are shown in the middleware overview in Figure 2.2. In this thesis we focused on the PSM and its implementations, while the transformation tool in between has been neglected.

2.2.2 Middleware Overview

Jadabs is compared to other middleware platforms in Figure 2.2. Different platforms are shown depending on the architecture’s environment and device’s capabilities. The infrastructure side shows the most common platform for infrastructure based environments whereas the mobile side shows solutions for smaller devices ranging from mobile phones over PDAs to gateways.

CCM stands for the CORBA Component Model [110] which was proposed by OMG, and defines a standard for component design and development architecture. The CCM model is built on top of CORBA [68], which defines a distributed infrastructure, and is based on the CORBA component programming model. CCM provides a run-time environment, ORB, for CORBA components. To simplify the task of building CORBA applications. The CCM model is implemented in different languages depending on the run-time environment. Implementations are available in C++, e.g., Qedo [112], MicoCCM [6] or Java, e.g., OpenCCM [107].

J2EE is the Java 2 Enterprise Edition [154] architecture with the specification lead by Sun Microsystems. J2EE is defined as a set of different specifications whereof Enterprise Java Beans (EJB) [40] defines the component concept.

Even though J2EE is very similar to the CORBA component model, it is not CCM compatible. Nevertheless, J2EE and CCM models have the possibility to interact with each
other. Through the use of the CORBA interworking model [109], EJBs can be used in CCM and vice versa.

**Jini** [12] is a distributed service platform defined by Sun. It is based on Java and enables a platform to dynamically load code from the network. A service requestor is able to look up services in a centralized registry. The registry is known in advance or uses broadcast to announce its presence. As it uses RMI as underlying communication protocol, Jini can only be used on devices, which are capable of running RMI based on Java.

**.NET** [70] provides a distributed infrastructure which is built on standards like XML, HTTP, and SOAP. The .NET components are comparable to the Enterprise Java Beans whereas distributed .NET components communicate through the .NET Remoting [136] framework. The .NET Remoting framework enables the use of different transfer mechanisms like HTTP and TCP. The .NET Framework consists of a Common Language Runtime (CLR) [159] environment comparable to the Java run-time environment, and has been standardized at ECMA as the Common Language Infrastructure (CLI) [48].

On normal desktop machines the .NET Framework (F) is used while for handhelds with PocketPC, the .NET Compact Framework (CF) is available.

**lwCCM** is the lightweight CORBA Component Model [111], which represents a subset of the functionality of the full CCM specification. Lightweight CCM components are interoperable with the CCM components such that a component-based application may use components of both models. In the lightweight CCM components have some of their operations disabled. They lack of persistency, transactions and security.
GAIA [140] is a middleware infrastructure for active spaces. The system focuses on the management of active space resources and provides location, context and event services. GAIA is based on GaiaOS [25], a component based meta-operating system, that runs on top of existing systems such as Windows2000, WindowsCE, and Solaris. GaiaOS uses dynamic TAO [141], a CORBA compliant reflective ORB, that supports run-time configuration. Applications on smaller nodes require the CORBA infrastructure based on more powerful machines. The GAIA Microserver [27] which runs on mobile phones always requires a proxy on the GAIA infrastructure side.

Jadabs fits infrastructure based environments as well as mobile environments. Jadabs is shown in Figure 2.2 in two boxes to represent the capability of using it without a proxy in the infrastructure environment. For example a mobile phone with Jadabs is able to communicate with other mobile phones in the same way as it does with more powerful nodes. Due to the adaptive behaviour of Jadabs, the architecture can be adapted to fit into current infrastructure based environment.

2.3 Mobile Environment Challenges

The architectures previously described have been designed in most cases to be used in infrastructure based environments. In many ways however, they lack the capabilities to be used in mobile environments. A middleware platform for mobile environments has to cope with a wide diversity of challenges as is described in the following.

Mobility

The application framework has to be extendable in two mobility directions. First, the user is mobile, carrying his device to any possible place which requires of the communication to change accordingly. Second, depending on the location, the environment may provide different applications for use which means the applications code needs to be transferred to the device. Jadabs supports the application mobility type through its dynamic adaptive behaviour where new components can be loaded, and the mobility of a device is supported through a messaging communication framework.

Resource-Awareness

Mobile environments consist of several different devices including sensors, displays, mobile phones, handhelds, and laptops. Applications running on these devices must be aware of each other and therefore need to inform each other of their location. In a meeting room, for example, a projector announces its availability. The laptop of a speaker entering the room may automatically recognize the projector and transfer the speakers presentation to the projector.
2.3. Mobile Environment Challenges

Resource-awareness is not only about recognizing other devices but also about finding other services which provide additional or adaptable extensions. For example, an SMS is usually sent from the mobile phone over GSM. If the user is in an infrastructure environment where a Bluetooth connection is available and a free SMS gateway is accessible, the SMS could be sent over this connection. This requires the mobile phone to be aware of other services which could fulfill the task of sending an SMS.

Resource-Constraints

Devices in the mobile environment may be restricted with regard to processor power, energy, memory, display functionality, audio/video input and output and so on. An application may require more resources than what are available from that device. It is for example not feasible to load a whole set of libraries onto a device with scarce memory if anyway not all functionality is used. By providing a descriptor of the platform, potential resources can be sorted out to activate only those which are actually needed.

Heterogeneous Environments

Mobile environments with a large number of entities interacting with each other are inherently heterogeneous. Such an environment can be a place where people meet spontaneously. The devices these people carry with them may interact with other devices.

A messenger application on a handheld may correspond with other users in its range. But the other users may have a different messenger application. Such systems therefore consist of different platforms, distributed infrastructures, and protocols. The problem of heterogeneity can be categorized as following:

Platforms. In mobile environments many different platforms are used, ranging from mobile phones running for example Symbian OS, Brew over PDAs with Palm OS, PocketPC, and embedded Linux up to gateways with an even wider range of operating systems.

To solve the operating system diversity, we base our application framework implementations on a higher layer than the OS platform. By choosing the virtual machine abstraction proposed by Java and .NET, we benefit from the broad acceptance and distribution over all platforms mentioned above.

Distributed Infrastructure. In a mobile distributed infrastructure the diversity of available network technologies is increasing. A mobile environment may not only consist of devices with 802.11 a/b/g, there may also be devices with Infrared, Bluetooth [19], ZigBee/IEEE 802.15.4 [73, 185] and so on. The distributed infrastructure therefore needs a way to bridge all these different technologies in order to facilitate a seamless communication.

Several protocols exist for distributed infrastructures. For example for communication their exists: RPC, DCE, RMI, .NET Remoting, and so on. There are
about the same number of discovery mechanisms available: Jini, UPnP, Salutation, SDP, UDDI, and so on. To overcome these diversities the middleware needs to be adapted to the required protocols.

**Application Protocols.** Diversity in protocols can also be found in application stacks. For example there are several instant messenger protocols like Jabber, ICQ, SIMPLE, Yahoo!. In order, to let messengers with different protocols communicate with each other in a mobile environment, protocols must be adapted accordingly.

2.4 **Jadabs Overview**

2.4.1 **Introduction**

To solve the mobile environment challenges, Jadabs is proposed. Adaptation has been recognized as a promising way to deal with dynamic environments [20, 128, 146].

The middleware architecture should be usable on different platforms. Therefore, a PSM has first been defined as the Jadabs model. From the PSM several implementations for different platforms are introduced. The Jadabs PSM is based on the service-oriented architecture concept whereas a service can also be seen as a component. A component is described by Szyperski [165] as follows:

*A Software Component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Due to resource constraints on the devices a two way approach was followed for the PSM implementations. In the first PSM implementation approach, the middleware platform was extended with adaptation capabilities. This approach is also referred to as the **segmented-adaptive** approach. Applications and its components are therefore at design time unaware of possible changes which may occur at run time. In the second PSM implementation approach, referred to as **monolithic-dynamic**, the adaptation capabilities can only be achieved through services specifically designed for later adaptation.

**Segmented-Adaptive.** The segmented-adaptive PSM approach proposes a dynamic lightweight container infrastructure which allows the components to be adapted transparently. In a segmented approach, any component can be dynamically loaded and unloaded in its container.

**Monolithic-Dynamic.** The monolithic-dynamic PSM approach requires the components itself to expose its adaptable interfaces. With a monolithic approach only specialized services have the capability of being loaded and unloaded again by a manager.
2.4. Jadabs Overview

Figure 2.3: Jadabs pervasive environment scenario

2.4.2 Architecture

Figure 2.3 shows an example consisting of four nodes. Three nodes are connected with each other in a mobile peer-to-peer infrastructure. This peer-to-peer infrastructure can be used for devices ranging from mobile phones up to desktop machines. The peer-to-peer communication paradigm fits very well into a mobile environment. Peers may enter and leave anytime in an ad hoc mobile environment. Once a node, in our example the gateway, is reachable from the mobile environment, all devices from the mobile environment can benefit and use services from the infrastructure based environment. The peer-to-peer infrastructure represents an overlay network over different network protocols like TCP, UDP, Bluetooth, and so on.

Overlay networks [18, 45, 157] have the advantage of incorporating different network protocols. This allows an application to interchange different transport protocols and use them in a transparent manner. Beside this property, overlay networks can be decentralized infrastructures. This allows that a distributed application may communicate directly with other applications without a centralized registration instance like in CORBA or Jini. This makes overlay networks a preferred concept for mobile environments. A minimal peer-to-peer implementation is used by Jadabs to interconnect resource constraint devices. More powerful devices are then able to connect the ad hoc environment to an enhanced infrastructure peer-to-peer network.

For heterogeneous dynamic service composition the possibilities of the peer-to-peer infrastructure and the adaptive possibilities of the container are combined. A service manager
on each device is responsible of keeping track of services provided by others. Once new services are recognized by the service manager, a matching with the local platform and services is performed based on the metadata of the service. This enables a service composition even though the services have not been designed for distributed infrastructure. On a match, the code is downloaded and activated.

Figure 2.3 shows the different proposed container models, the choice of which depends on the capabilities of the device. We distinguish two types of containers, a segmented component container and a monolithic component container which relates to the segmented and monolithic approach.

A service manager (SM) is responsible for activating and deactivating dynamic services. In a distributed infrastructure, the term service is used in addition to components. A service is a managed deployable component in Jadabs. Such a service consists of a descriptor which can be advertised by the service manager to other peers. Depending on the container, a service is fully self-contained or contains dependencies to other services or components. A fully self-contained service does not have additional dependencies and can be run on its own.

In the segmented container a new service can be loaded by giving a service description which is not self-contained. The service manager is then responsible of obtaining and activating all dependent components or services. Other components or services can be invoked directly or through interfaces. Segmented containers can be run on more enhanced devices like handhelds or laptops.

For more resource constraint devices like the mobile phone, the monolithic service container is provided. Such a node contains a service manager as well. Once a new service description is available for the extension of the node, the service manager downloads the service as one package over the peer-to-peer system and activates the service. In this system, local services cannot communicate directly with each other. They need a local event mechanism by using a publish and subscribe principle.

For the container model, a service-oriented architecture [158] proposed by OSGi [115] was chosen. The OSGi container was specified for Java environments. As we also provide containers for .NET devices, the OSGi API has been ported to C# and implemented as a monolithic container.

Both, the segmented and the monolithic container approach have been extended to allow adaptive behaviour. For a segmented-adaptive approach the container is extended with a dynamic aspect-oriented programming approach. This allows components and services to be implemented independently of possible future adaptations. In the architecture implementation, two adaptive variants are provided. This allows a more powerful device to use the adaptive container which is faster.

For the monolithic-dynamic approach the adaptation possibilities have to be integrated into the services itself. The service extension together with the service manager allows Jadabs to be dynamic in changing mobile environments. These adaptation possibilities have to be specified explicitly as such in the Jadabs PSM by using the local event mechanism.

In the segmented-adaptive approach the components and services are independent of possible future adaptation requirements. This is different from the monolithic-dynamic ap-
2.4. Jadabs Overview

proach where services have to expose their interfaces explicitly to the local event mecha-
nism.
Chapter 3

Jadabs – Model and Service Composition

3.1 Introduction

Jadabs uses a service-oriented architecture defined by the OSGi Alliance [115]. A service-oriented architecture (SOA) [158] can help to reduce software complexity as part of separation of concerns [166]. The concept of a service-oriented architecture is shown in Figure 3.1. A Service Provider registers a service implementation with its interface in the OSGi container’s local registry (1). A Service Client component is then able to lookup the registered service (2) and bind to the Service (3).

The SOA mechanism was chosen for many reasons:

- Service-oriented programming forces designers and developers to follow the separation of concerns paradigm. Dependencies among different services have to be specified at design time in order to increase the usability of such services.

- An SOA mechanism can be implemented in a very small footprint. Depending on the functionality of the container the footprint ranges from 10 KBytes to 200 KBytes. This makes such a component model interesting not only for desktop machines but also for handhelds and even mobile phones. A container implementation can also be kept totally independent of remote functionality.

- The SOA mechanism defined by OSGi is already a proven solution in many different fields ranging from home residential gateways [178] over deployed vehicle solutions [58, 170] to development environments like Eclipse. OSGi implementations are available by IBM, OSCAR, Sun Microsystems, Gatespace, AveLink, and ProSyst.

- The SOA mechanism is language and platform independent whereas OSGi has defined its service platform in Java. A .NET implementation of the OSGi API is possible but the unloading of components cannot be supported. Physalis [123] provides such
an implementation and a simple OSGi.NET implementation is also provided in this thesis.

### 3.2 Jadabs-PSM

The model of the service-oriented architecture is reflected in the proposed platform specific model (PSM) for Jadabs. The representation of the component model has been chosen from the CORBA Component Model (CCM) [110].

#### Components

Figure 3.2 shows in CCM notation a service running in an OSGi container. In CCM a main component executor interface is used to control the component’s lifecycle. In OSGi such an executor interface is called BundleActivator. OSGi defines a bundle as a deployable package which may contain a BundleActivator. In the proposed Jadabs model, a bundle which contains an activator is called service plugin and a bundle without an activator is simply referred to as a bundle. A service plugin references its required bundles so that they can be loaded automatically. An application is then assembled out of a list of service plugins.

Figure 3.2 shows an application example with two service plugins, one of which is the Model and the other a GUI service plugin. These service plugins require further bundles A, B, C, and D whereof bundle C is referenced in both service plugin’s dependency paths.

A bundle which is used in several service plugins is a common occurrence in component frameworks. A key factor in component frameworks for resource constraint devices is its footprint [8] which requires the same bundle to be active only once. Two types of footprint are important for such devices: static and dynamic footprints. The static footprint is the
amount of storage needed on the disk to hold the program code. The dynamic footprint represents the amount of memory used by a running instance of the application. Therefore, it consists of code, heap, and stack size.

Different component systems address this issue in a variety of ways:

- Native component systems like COM, and CORBA use dynamically linked libraries deployed as DLL files on Windows systems or shared object, SO, files on Linux. These libraries should appear only once per system in order to decrease the static footprint. At run time, several applications are able to use the same shared library which decreases the dynamic footprint. Difficulties in managing DLLs have lead to the term "DLL-Hell" [125]. An often chosen solution is to provide each application with all the required DLLs which in turn foils the intended advantage of shared libraries to keep down the static footprint.

- In .NET, metadata was introduced which describes an assembly, the DLL, with its version, authors, and external assembly references. This solves the "DLL-Hell" and leads to a minimal static and dynamic footprint. In J2EE the application server is able to load classes from the same physical libraries on disk. To separate hosted applications from each other many J2EE implementors use different class-loaders for the same library which, however, increases the dynamic footprint.

- The OSGi specification proposes to include libraries in the bundle which contains the BundleActivator. The libraries are then exported from this bundle to other bundles which require such a library. This way only one library is loaded which leads to a minimal static and dynamic footprint.

However, the proposed assembling of service plugins in the OSGi specification may lead to an increased dynamic footprint once bundles are not used anymore. As each service plugin is loaded in its own class-loader, single bundles cannot be removed from a service plugin. Once a service plugin is no longer needed other service plugins may still reference a required bundle which is in the service plugin about to be unloaded. This prevents the container of unloading the inactive service plugin, thus keeping vital memory occupied. The proposed solution of separating service plugins and regular bundles allows Jadabs to unload bundles independently. As shown in Figure 3.2 when for example the GUI service plugin is unloaded, bundle C is independent of the service plugin and the other bundle A can be unloaded.

**Container**

The loading of new bundles as well as the complete unloading of bundles is essential for memory-constraint devices. Depending on the capabilities of the platform where the Jadabs-PSM is mapped to, two different container models are supported.
Segmented containers take full advantage of OSGi bundle loading. Each bundle and service plugin is loaded by its own class-loader. This allows the removal of those bundles at a later stage when they are no longer required, leaving the others up and running. The segmented container therefore needs the possibilities to load bundles independently of each other. As all bundles reside in the same process, method invocation over bundle boundaries have no performance penalty. Figure 3.2 shows the segmented container where only regular method invocations are used. For example bundle C, which is used by the GUI service plugin as well as the Model service plugin, is loaded only once.

Monolithic containers do not have the possibility of unloading each bundle independently which however does not conform to the OSGi specification. Monolithic containers assemble all bundles and service plugins which require regular method invocations into one container. A monolithic container is therefore the loadable and unloadable unit. At least one service plugin is responsible for the assembling of the required bundles into one monolithic container. Inter-container communication requires a local event mechanism. Figure 3.2 shows the monolithic containers as dashed bounding box with a message communication interface between the two containers. This time bundle C was loaded twice both in the GUI and Model service plugins. Bundle C does not come with an inter-container communication as it would be to expensive to call methods on bundle C over a local event mechanism. The monolithic container approach enables a small footprint container which can be used on mobile phones. It can also be used on platforms which do not allow a dedicated unloading of single bundles as it is the case for .NET.
Platform independence

The proposed component and container types allow us to design local and remote applications for heterogeneous environments. Applications are assembled in two different ways: first, at design time when the application goal has to be solved, and second, at run-time when the application can be adapted by other devices in its environment.

Components designed for a variety of devices need metadata to describe where they can be run on. A description is therefore required which allows a node to gather information about components to figure out if they can be run on the node's platform.

According to the OSGi specification the bundle metadata inside the Java package must be saved as a MANIFEST.MF file. This would require a heterogeneous node to load Java packages even though they are running on different platforms. Therefore, a similar concept to the eclipse plugin descriptor [116] or the J2EE [83] deployment descriptor was chosen. By describing the packages in an external file only that description have to be downloaded to match it against the own platform. Additional information to the already proposed manifest was included, for security, extension possibilities, and platform requirements.

3.3 Jadabs-PSM XML Mapping

Once the application design conforms to Jadabs-PSM, the necessary metadata needs to be generated. Comparable to the CCM-PSM to XML mapping, we require a mapping from the Jadabs-PSM to XML. CCM additionally generates an IDL [106] from which the code implementation is generated. The generation of an OSGi like IDL was skipped as it would only be used to get the code frame. Even though we support two platforms, Java and .NET, the code frames can easily be done by hand.

The mapping of the graphical representation into an XML representation is shown in Figure 3.3. All elements of the container and the bundle including the platform where it runs on are described.

- An OSGi service plugin descriptor contains an OSGi service and activator description; it has the ".opd" file extension.

- An OSGi bundle descriptor contains a bundle description; it has the ".obr" file extension.

- An OSGi container descriptor describes each container's properties and configuration; it has the ".ocd" file extension. The OCD can also be included in the platform descriptor.

- A platform descriptor contains platform information including hardware and operating system configurations; it has the ".pad" file extension.
3.3.1 OSGi Service Plugin Description (OPD)

The plugin description is used to define both the required properties and the provided services. The service description is exposed to other peers in the mobile environment for querying and finding matching services.

UUID specifies the service plugin descriptor in form of a UUID. The structure of the UUID is <location>:<service>;<version>:opd. A location represents a reversed DNS entry as proposed in Java for package names. The service and version entry allows the container to distinguish between different services under the same location name. Each version is represented as major.minor.bug which facilitates the matching of plugin descriptors with various versions.

Service specifies the service plugin name which is used as an alias for the package name.

Extension defines a provided extension which is usually a service interface. Extensions describe the interfaces which are provided to other service plugins. An extension is referenced by an <Extension-Point> to define the required extension.

ID names an alias for the provided service interface. It can preceded by "REMOTE/" to specify a remote extension.
3.3. Jadabs-PSM XML Mapping

<Service> states the service interface name including the package name.

<Extension-Point> defines a required extension of another resource. A resource may be an extension of a service plugin, a container property, or a platform property described in any of these descriptor files.

<ID> an exact interface name or an alias. It can preceded by "REMOTE/" to specify a required remote extension.

<Service> states the exact service interface name including the package name.

<ServiceActivatorBundle> describes the implementing bundle which includes the BundleActivator. In case the plugin provides remote extensions the ServiceActivatorBundle does not need to be specified. In this case the service manager stores the OPD to be able to resolve remote extensions.

<UUID> defines the bundle descriptor which includes the service plugin code. The UUID is of the form <location>:\<service>:\<version>:\obr.

3.3.2 OSGi Bundle Description (OBR)

Each service plugin also requires one bundle descriptor, which specifies the bundle dependencies. The generated bundle descriptor is compatible with the OSCAR Bundle Repository [114] and was extended as required for our Jadabs-PSM.

(Bundle-UUID) specifies the bundle descriptor which includes the bundle code. The UUID is of the form <location>:\<service>:\<version>:\jar.

(Bundle-UpdateLocation) refers to a URL from where the bundle can be downloaded.

(Bundle-Security) include a signature to verify the authenticity of the bundle.

<digestGenerationAlgorithm> specify the digest algorithm to use. The digest can also be used as a checksum only.

<keyGenerationAlgorithm> refers to the algorithm used to generate the signature.

<digest> denotes the digest generated with the previously described algorithm.

<signature> references the signature generated over the digest by using the specified key algorithm.

<Import-Package> refers to the package name exported by another bundle.

<Export-Package> lists the own exported packages.

<Dependency-UUID> specifies the dependencies on other bundles in the form of a Bundle-UUID.
3.3.3 OSGi Bundle Properties

The property descriptions allow us to parameterize the plugins and bundles depending on the platform configuration. Therefore, the properties are included in the platform description. The property configuration follows the form: \texttt{-name = value}.

For more advanced configuration requirements, Knopflerfish proposes his own metadata [179]. To use Knopflerfish’s property configuration, additional basic platform bundles are required. To keep the platform as lightweight as possible the name-value pair solution was chosen. Nevertheless, the advanced configuration possibility could be added at a later time when it is needed by a service plugin.

3.3.4 OSGi Container Description (OCD)

The container descriptor determines the properties of the container wherein the bundles are run. The minimal feature is the container description while other properties such as name-value pairs may be specified when required. As part of this work different containers with varying functionality are presented. Service plugins which then require a container with a specific property add an extension-point to their OPD description. Apparently the OCD properties are included in the platform description file.

\texttt{
<ID> names the container. \\
<Container> specifies which container is to be used. So far, different containers are supported. \\
<Container-Version> specifies the version of the container. 
}

3.3.5 OSGi Platform Description (PAD)

The platform descriptor defines all the platform specific issues which are required by the container or might be required by service plugin extension-points.

Properties like processor and display define hardware specific issues. More properties can be added when required.

\texttt{
<ID> names the platform specification. It can also specify a type of a platform. \\
<OSGiContainer> specifies the containers run on that platform. A platform may run different containers, for example a laptop is capable of running at least three different types of containers. \\
<NetIface> specifies the different network interfaces which are available. This element is not required and only available on platforms with a network interface. 
}
3.4 Dynamic Service Composition

With the proposed Jadabs-PSM we can build and assemble applications for small devices. An application in a mobile environment is no longer bound to a specific location or even device. Whenever the user moves, applications need to adapt to the changing environment. For example, a mobile phone uses GSM to send out SMS from the countryside. Once the phone is within a company’s Bluetooth infrastructure, it may send the SMS over a cheaper local infrastructure. But the Bluetooth connection or the cheaper short message service first needs to be recognized as a possible SMS replacement service and additional code needs to be installed on the phone.

Several solutions have been proposed for service discovery [12, 60, 144, 172]. None of them is available for resource constraint devices like a mobile phone nor applicable in a heterogeneous environment as required by mobile environments. For example, the discovery mechanism of Jini [12] is a Java only solution. Therefore, a discovery solution is required which is based on a message protocol which can be implemented on different systems. A suitable service discovery solution must be applicable in a mobile and infrastructure environment as well. A peer-to-peer system is used which allows us to overcome such infrastructure hazards as it can be used both in mobile environments and on different platforms.

In addition to the service discovery mechanism, a service activation is required. With Jini it is possible to download a proxy over RMI [105] after the service discovery phase. As services are used in heterogeneous environments, it should be possible to download services for other platforms than Java. In Jadabs, a peer announces its platform description to get services which fit to its architecture. The proposed service discovery does not only match platform descriptions but also provides and required extensions and extension-points as defined in the Jadabs-PSM.

We propose a dynamic service composition mechanism which matches available services in the nodes mobile environment with the nodes running application. The required services are described declaratively by the service plugin. This stands in contrast to the service mechanism as proposed in Jini where each service lookup is done programmatically inside the requesting service.

3.5 Jadabs Service Composition

In Jadabs applications are designed following the Jadabs-PSM. Thereafter, applications are assembled by plugins and bundles which can be run on a specific Jadabs platform defined by the container and platform descriptor.

The dynamic service composition framework in Jadabs distinguishes two different phases:

1. **startup phase** to start an application from a set of given service plugins,

2. **dynamic phase** where an application is dynamically and autonomously adapted at runtime.
In the startup phase an application is composed of different plugins and bundles. According to Jadabs-PSM, the different components are described with plugin descriptors (OPDs) and bundle descriptors (OBRs). At load-time these descriptors are matched against the local container descriptor (OCD) and platform descriptor (PAD). Therefore, in order to startup an application, at least one OPD of the application and the OCD/PAD is required. A service manager is in charge of loading the appropriate OPDs and OBRs. A dependency graph is built up for the plugin with all required additional OPDs and OBRs. In case all required dependencies of the application plugin can be resolved, that is the extension-points can be matched and OBRs can be downloaded, the application is started.

Once the startup phase has finished, the service manager switches to the dynamic phase. Thereby, the service manager monitors its mobile environment for new available services which match and extend the already started plugin descriptors. Matching services are then automatically downloaded and activated with respect to the security mechanism.

If we want to startup an application with only a plugin descriptor, a remote repository can be queried from where the dependencies can be resolved. Otherwise, the plugins are resolved from the local repository. Two different remote repositories are distinguished: infrastructure repository and mobile peer-to-peer repository. The mobile peer-to-peer repository has the ability of providing services and bundles to peers in its mobile environment. The infrastructure repository is used whenever a connection to the infrastructure based en-
Figure 3.4 shows a mobile environment scenario with a mobile phone, handhelds, and a gateway. The gateway has a connection to the infrastructure repository. The infrastructure repository consists of all available plugins and bundles which can be installed on any device. When an application gets started on a device with an infrastructure connection, the infrastructure repository is queried for available bundles. In case the peer-to-peer infrastructure is not already locally available, it is downloaded from this repository.

The mobile peer-to-peer repository consists of all plugins and bundles which are available over the local peer-to-peer infrastructure. When a new service is going to be installed in a mobile environment without an infrastructure connection, all required dependencies need to be available in the mobile peer-to-peer repository.

### 3.5.1 Infrastructure Repository

As shown in Figure 3.5, infrastructure repositories are accessed over two different interfaces: ServiceManager and HTTP (Repository.xml). Internally, the repository can be structured in a proprietary way. In our approach, we chose the filesystem as it allows to structure the plugins and bundle resources in the UUID form. As shown in the Jadabs-PSM for example, a plugin UUID is of the form shown below and maps to an appropriate folder structure. The filesystem structure is related to Mavens’ [11] repository structure for compiled packages.

**UUID:** <location>:<service>:<version>:opd

**Folder:** location → opds → service-version.opd

The repository interfaces allows the service manager to query the infrastructure repository for matching services. A node which starts up with a set of given plugin descriptors connects to a repository by connecting to the repository’s service manager or querying the repository.xml. These infrastructure repositories can be distributed over different servers. For example, one server provides an HTTP interface whereas a service manager is hosted by another machine, for example the developer machine.

**Repository.xml** interface consists of an HTTP connection to access the repository.xml file. The XML file contains a list of available OBRs and OPDs on the server. A remote node requesting the repository.xml file is then able to build up a dependency graph and load the bundle from the specified location defined in the OBR.

**Service Manager** is listening as a peer-to-peer node for service discovery messages. Service discovery messages contain a filter description which allows the other service managers to match the filter against the providing services. With the filtering mechanism, a list of service descriptions can be returned to the requesting peer.
3.5.2 Mobile Peer-to-Peer Repository

In cases where an infrastructure repository is not available, e.g., in a mobile environment, services have to be loaded from other devices. For example, as shown in Figure 3.5, PDA-B comes into the communication range of PDA-A. PDA-A and the mobile phone already use a service. Once PDA-B recognizes the new device PDA-A, a service discovery message is sent from B to the others. A service filter string is sent to the other nodes which can then match the filter against the local available plugins. In case a service matches the filter, the others send back a service description which B can then download and activate. The mobile peer-to-peer repository uses a similar service manager already used for the infrastructure repository.

The advantage of querying the peer-to-peer repository is that only services in its mobile environment are returned. The mobile environment is therefore defined as a local peer group where a peer connects to. Usually, the peer group is defined locally. This allows a new peer to discover services on devices which are connected to the same group. As the discovery acts on the overlay network, different network technologies are incorporated. Only a gateway has the capabilities of connecting the mobile environment to the infrastructure based environment. Discovery messages, which are only dedicated to the mobile environment, are not further propagated by the gateway.
3.6 Jadabs Plugin Loading

The Jadabs loading is separated into a *startup* and *dynamic* framework according to the startup and dynamic composition phases, see Figure 3.6. The startup framework is responsible for the loading and activation of the required plugins. The dynamic framework, which requires the peer-to-peer libraries, is in charge of detecting changes in the mobile environment. Service changes in the mobile environment may force the service manager to load new plugins or unload them again.

The startup framework contains a Plugin-Loader and a Bundle-Loader. The plugin-loader is in charge of resolving a plugin into its dependencies in a resolution phase. According to the Jadabs-PSM, a plugin may contain dependencies to other plugins and bundles. The resolution phase is separated into a *plugin resolution* and the *bundle resolution* phase. During the plugin resolution, all plugin dependencies have to be resolved into a plugin dependency tree. Once the plugins are resolved, the dependency tree is passed on to the bundle loader for bundle resolution.

To resolve plugins and bundles three different sources are available: Service Manager, HTTP-Source, and local Repository. The Service-Manager source uses the mobile peer-to-peer repository to resolve bundles and the HTTP-Source reverts to the infrastructure repository if any available. Lastly, local-repository looks up plugins and bundles in the local filesystem.

### 3.6.1 Service Manager

The service manager (SM) is a key component in the dynamic service composition mechanism of Jadabs. It is in charge of discovering matching services in its mobile environment. Therefore, the service manager was designed to be used in a wide variety of devices. The following key aspects were considered:

- The service discovery mechanism is independent of the used peer-to-peer infrastructure. We only require an infrastructure which transports our service messages and gives the opportunity to flood in mobile environments.
• JXTA [80] has been used as a peer-to-peer infrastructure. Its message format has been specified as an IETF draft. JXTA proposes an overlay network that allows us to abstract the different network technologies available in mobile environments.

• The service manager requires a simple protocol stack which can be implemented for small devices like a mobile phone.

Since the service manager needs to be implemented for resource constraint devices as well as for more powerful nodes, two types of service manager are distinguished: active and passive.

Active service managers are able to match plugin descriptors announcements. When a new peer reaches a new mobile environment it sends service discovery messages to the mobile peer-to-peer infrastructure. An active service manager is able to match such discovery messages. Matching plugin descriptions are then sent back to the requesting peer which downloads the plugin and its dependencies.

Passive service managers run on resource constraint devices and are only able to send a service discovery message to active service managers. They are not capable of reacting to service discovery requests as they may not be capable of matching the filter against available plugins. The passive mode is mostly used by resource constraint devices which are not capable of processing a match mechanism.

3.6.2 Plugin Loader

Figure 3.7 shows an example dependency graph of plugin A. Plugin A is the startup plugin and needs to be resolved before it can be started. In the startup phase, the plugin loader is able to use the different available repositories to find the required dependencies. For example, plugin A would resolve to plugin B and D. Each plugin has at least one related bundle which contains the bundle activator to startup the plugin. Additional bundles are then referenced by the activator bundle.

In the startup phase the plugin descriptors are processed. A plugin descriptor contains a set of extensions and extension-points which have to be resolved. Extensions are provided to other services whereas the extension-points have to be resolved matching other resources like extensions, platform, and container properties. In Figure 3.7, OPD-A defines an extension-point, ExtP-A, which is provided by two OPDs B and D.

To match the extension-points, all local available plugin descriptions are queried and compared. In case no matching extension could be found, a request to the information sources is sent out. As shown in Figure 3.6 the list of available repositories is walked through starting with the local repository and then the infrastructure repository if available, and finally the peer-to-peer repository inside the service manager. The result of the plugin resolution is the dependency tree of the to be installed plugin.

A filter to match service plugins follows the EBNF form shown below.
A filter request which is sent out may look as following. It represents a request for an extension with a PeerNetwork ID. Furthermore, the platform and container properties are specified which an appropriate extension has to meet.
Remote Service Plugins

The plugin mechanism explained so far composes extensions and extension-points locally. Many services require also remote services to provide appropriate remote services to the user. The Jadabs model provides therefore the prefix "REMOTE/" on the IDs for extensions and extension-points. The remote prefix marks the extension to be usable as a remote service. A service plugin with such remote services have to be published to other nodes. The service manager sends therefore the OPD without a <ServiceActivatorBundle> element. The receiving service manager stores this OPD as a remote OPD from the received node and does not download an OBR.

For the extension-point the remote prefix notifies the service manager to resolve the extension-point with a remote extension which has been stored previously or is requested form the available repositories if no such extension is available.

3.6.3 Bundle Loader

Once the plugin loader has resolved the plugin dependencies, the bundle loader can resolve each plugin to its bundle dependency graph. The bundle descriptor is specified as a UUID in the plugin descriptor. This allows the bundle loader to query its repositories for the appropriate bundle descriptor (OBR). This may result in a remote request for the required OBR. With the received plugin OBR, the dependent OBRs are specified declaratively which allows to create an unambiguous dependency tree.

As OBRs and its bundles are downloaded remotely, it is possible that not all dependent OBRs can be found in the mobile environment. In case an OBR tree cannot be built completely, the plugin cannot be installed and activated. This leads to a backtracking from the bundle loader to the plugin loader which will try to download another resolved service plugin.

A downloaded OBR and its corresponding package file need to be verified. First, the included digest or checksum in the OBR file is checked. Second, on a matching checksum the signature is verified. The signature has been generated on the digest. The security mechanism in Jadabs does check the authenticity of downloaded packages before their installation.
So far, there is no decentralized trust infrastructure available in Jadabs as proposed in the P-Grid [3] project. Each node needs therefore a public key of the repository. For infrastructure based repositories their public key is known at startup of Jadabs. For mobile environment repositories where the other peers are not known in advance, new unknown public keys have to be checked by the user itself. The implementation is based on the lightweight Bouncy-Castle [22] security provider.

Once all OBRs and bundles have been downloaded and the verification step was successful they are installed and activated immediately.

### 3.6.4 Dynamic Service Composition

The mechanisms described until now assumed that the starting plugin is provided by the user. The user who is in this case the administrator controls the mobile environment and starts and stops the plugins on different devices. But an administrator approach is not feasible in mobile environments with many joining and leaving nodes. To overcome this restriction, the service manager supports a dynamic service discovery mechanism which fulfills two properties: autonomous and adaptive.

**Autonomous**, because the service manager acts independently of a centralized node. So, the service manager does not require to compare its installed services against a local server. It rather compares its locally available services against the available services in the mobile environment. The service manager keeps track of all installed and provided plugins. It uses the mobile peer-to-peer concept to publish and discover new plugins.

**Adaptive**, as new available services may extend already running services. To extend services at runtime an adaptive container is required. Different solutions exist which allow adaptation on the container level. A plugin supporting a dynamic extension to another plugin therefore needs to specify in its plugin descriptor the type of required adaptation container. This type has then to match the platforms container description.

The autonomous behaviour of the service manager is illustrated in Figure 3.8. At startup a KeepAlive and LeaseCheck thread get started. The KeepAlive thread sends out messages that the device is still available in the current mobile environment whereas the LeaseCheck thread controls the leases of other peers. Each peer has its own lease and once the peer leaves, after a timeout the peer gets unavailable and all extensions provided by that peer need to be released. Services which have extension-points to extensions provided by the unreachable peer need to be released.

Three types of messages are required by the service manager. First, KeepAlive messages are received regularly by any node in the mobile environment. Second, if the peer is already known the lease is updated. If the peer is unknown a Filter-REQ is sent out. A received Filter-REQ is only handled by active service managers as they are capable of matching a filter. On a matching service the OPD of the service plugin is sent as an OPD-ACK. Third, on receiving of an OPD-ACK message, a service check is performed if it fits the platform. If it can be installed a thread is started for the OPD installation. In case it is a remote service OPD it is registered in the plugin loader.
Chapter 3. Jadabs – Model and Service Composition

3.7 Discussion

The Jadabs model and the dynamic service composition represent a powerful concept for the adaptation in mobile environments.

The model is required to describe different resources like the nodes’ platform, different containers, services, and bundles. By using metadata we are able to describe services run in a heterogeneous infrastructure. The supported metadata can be used on resource constraint devices and enables the dynamic service composition. This allows the service manager to take control of running services. Upon availability of new services in the mobile environment local services are recomposed. New services may be required by the mobile environment where the node has moved to or they are provided by new nodes in the mobile environment. The service manager which recognizes changes in the mobile environment is able to match appropriate services and download them for activation. This way activated services are able to transparently change the behaviour of running applications.

The proposed model and service composition differs from other solutions like CORBA, Jini, GAIA, .NET in different ways. The Jini proxy mechanism is a Java only solution which depends on resource powerful nodes. Similar, CORBA and .NET neither provide a component model nor metadata which can be resolved on resource constraint devices like a mobile phone. GAIA on the other hand, is a centralized solution, thus making it unsuitable for ad hoc mobile environments.

With Jadabs a model and service composition mechanism is introduces to start-up appli-
cations with basic components and services. Once new services are available, the application gets adapted automatically. Developers do not have to explicitly specify interest in specific services as it is done in Jini or CORBA. Through the description of services with metadata, the service manager is able to recompose services in an appropriate way. The model and service composition is mainly used for small devices but can also be used as a basic layer for more powerful devices to bring up a common, more enhanced distributed infrastructure.
Seite Leer /
Blank leaf
Chapter 4
Jadabs Middleware Platform

4.1 Introduction

Jadabs is based on two different types of container: segmented containers and monolithic containers (Figure 4.1). The container types allow to categorize the different implementations for the Jadabs middleware. A segmented container is more flexible and extendable than a monolithic container. On the other hand, not every system is capable of running a segmented container. Therefore, we provide containers for both types depending on the capabilities of the system.

The segmented container type can be further divided depending on the adaptive capabilities. Figure 4.1 shows three different container stacks proposed for dynamic adaptation. The OSGi/PROSE stack is the most flexible container where even the container can be adapted. The drawback is that it requires a normal Java virtual machine. For more resource constraint devices like handhelds or gateways, Nanning is used which runs on smaller Java VMs. This solution extends the OSGi container with dAOP and allows dynamic adaptation on the service interface level. The last container stack can be used for mobile phones and .NET platforms. Here, dynamic adaptation in the monolithic container is achieved through loading and unloading of services and an adaptation of the communication between service plugins. Listed below are the containers we propose with their specifications, all of which can be used with devices in the mobile environment.

Segmented Containers

**OSGi/PROSE.** PROSE [128] is a run-time aspect-oriented programming engine for Java. The run-time engine can be hooked into different Java virtual machines. In combination with OSGi, it supports a full run-time AOP container. Due to its requirements on the virtual machine it is best used on desktop machines. The OSGi/PROSE container is a segmented container and fully OSGi compatible.

**OSGi/Nanning.** Nanning [92] is a dynamic aspect-oriented programming library. Integrated into the OSGi container, the components can be extended dynamically
without replacing components. The small footprint of OSGi and Nanning allows the user to run this container on small devices like handhelds or gateways and requires only a Java VM with a Connected Device Configuration (J2ME/CDC) [160]. The segmented container allows service composition at a fine grained level.

### Monolithic Containers

**mOSGi-CLDC.** mOSGi-CLDC [56] uses a different container which is not OSGi specification compatible but is OSGi API compatible. Instead of a container where bundles or plugins can be loaded and unloaded, with mOSGi each plugin contains its own mOSGi container whereas unloading inside mOSGi is not possible. In this case, the adaptive system is restricted to the dynamic loading and unloading of plugins. This monolithic container can be run on mobile phones supporting a Java VM with Connected Limited Device Configuration (J2ME/CLDC) [161].

**mOSGi.NET(F)/(AOP-Engine).** As part of this thesis, an AOP-Engine [55] has been implemented for .NET. Its run-time AOP behaviour is comparable to PROSE. The proposed .NET AOP-Engine is supposed to run inside an mOSGi.NET Framework implementation. So far, the implementation of the AOP-Engine has not yet progressed to combine it for autonomous adaptation. The AOP-Engine requires the .NET Framework which runs on desktop machines. The mOSGi.NET implementation is a monolithic container comparable to the mOSGi-CLDC container.

**mOSGi.NET(CF).** The mOSGi.NET Compact Framework (CF) container uses the same mOSGi.NET container as for the normal Framework.

### 4.2 Segmented Containers

Segmented containers are based on the service-oriented architecture defined by the Open Service Gateway Initiative (OSGi) Alliance [116]. Several implementations are available [46,
whereof we chose Knopflerfish due to its small footprint of about 200 KBytes. In OSGi the deployment of components is packaged into so-called bundles which are Java files. These bundles are activated in the OSGi framework and services can be registered as explained in section 3.1. Services can be added and removed at run-time. But a single component cannot be replaced or updated at run-time. The reason is that components may have states which gets lost once the component is stopped. Therefore, a service which is going to be adapted with new functionality has to be stopped and restarted, which in turn leads to state loss. Another disadvantage is the continuous check at the service client side to make sure the service is still available before a call is made.

To overcome these problems we propose two containers which take advantage of dynamic aspect-oriented programming.

Aspect-Oriented Programming (AOP) is a software design technique that allows the separation of orthogonal concerns within an application. These orthogonal concerns are then programmed as separate aspects rather than locating them in many different places in the code. The main advantage of AOP is precisely the possibility of abstracting out concerns that crosscut through the application (i.e., appear in many different places throughout the code). These aspects can then be treated as separate software modules, thereby increasing the modularity of the design. For instance, Zhang and Jacobsen argue that aspect-oriented re-factorization can enhance the modularity of middleware and reduce the structural complexity.

An aspect defines what to do (e.g., invoke an additional method) when a particular point is reached in the code (e.g., invoking methods with certain signatures, modifying a variable, etc.). Conventional AOP uses a weaver to add the aspect code to the base code of a program at compile time, e.g., AspectJ. In dynamic AOP, the aspect code is added (woven) at run time by executing it whenever the specified point, the so-called interception point, in the execution is reached. Aspects can also be dynamically withdrawn (unwoven) leaving the application in its original state.

The OSGi/Nanning container uses a dAOP approach which can be used on small devices like a handheld and is part of the container. The second approach makes use of PROSE, a dAOP approach integrated into the virtual machine.

4.2.1 OSGi/Nanning

Combining SOA and dAOP into a reduced footprint container was the main concern for this container. Several solutions exist for either of these two concepts. We chose the combination which allowed us to use the core layer on many different platforms. As a common platform, ranging from small devices up to normal machines, we chose the J2ME/CDC so that we can run the higher layers on compatible Java VMs. On the other hand, it limits the capabilities for SOA and dAOP. In the following the chosen solution is described in more detail. As neither the SOA nor the dAOP solution were designed to be used together, we explain the enhancements made to each system to allow support for the other.
Aspect-Oriented Programming  The dynamic AOP capabilities are primarily restricted by the platform and resource limitations of small devices. Nanning [92] was chosen due to its small footprint and easy adaptability in a small container.

Nanning is a dAOP solution with a proxy oriented approach. Instead of calling a service directly, a representative, the proxy, will be called which is then responsible for the forwarding of the call to the real service implementation. Figure 4.2 shows the registration of the service by the provider. The dynamic proxy concept is available since the release of Java 1.3 and therefore also for J2ME/CDC, which is based on a stripped down Java VM 1.3. By using this Java API, a proxy can be generated to the given interface and the actual instance.

Every bundle has to be loaded with its own class-loader. This allows us later on to remove single bundles from the container. Nanning was not designed to be used in a container infrastructure where components can be removed at a later stage. New aspect components can be added to crosscut the functionality of already running components. Once the aspect components are not needed anymore they can only be deactivated but not completely removed. The class instances will remain in the Java VM. The reason is that in Nanning, the proxies are created with the system class-loader which requires a reference to the interfaces and implementations. In this case, aspect components need to be added to the system’s classpath.

To overcome this problem we extended Nanning to allow an independent unloading of aspect components. To create and load the new proxy class in the service implementation bundle, the services class-loader has to be used instead of the default’s system class-loader. Figure 4.2 shows the proxy as part of the service provider bundle. When the service provider registers the interface together with the implementation, the service provider’s class-loader is also passed along.

The proposed mechanism allows us to crosscut on the proxy interfaces inside the service provider. On activation of an aspect bundle which crosscuts the proxy, the service provider class-loader needs to load the aspect bundle classes from that corresponding class-loader.
To load classes from bundles which are not known at deployment time, OSGi provides a dynamic-import statement inside the bundle’s manifest file.

**Service-Oriented Architecture (SOA)** As the OSGi API does not support the integration of AOP functionality, we introduce an additional API. The following listing shows the `AOPContext` interface which is an extension to the OSGi `BundleContext`. The `AOPServiceRegistration` and `AOPService` interfaces, gathered through the `AOPContext`, allow the user to query the registration and the service. The implementation uses the underlying AOP concept for crosscutting concerns.

```java
interface AOPContext {
  AOPServiceRegistration registerAOPService(
    Class clazz, Object service, Dictionary properties);
  AOPService getAOPService(ServiceReference reference);
}
```

The following listing shows how services are registered in OSGi normally and what the process looks like with our adapted interface. Crosscuts can be dispatched inside the proxy. Therefore, the developer has to be aware of two design principles when designing such dynamic services. First, the service should only be called over the interface. This requires registering the service only with the interfaces (line 7) compared to the possible class registration example (line 12). Otherwise, the implementation is called directly and would circumvent the proxy which is then not able to intercept the method call. This leads to the second design principal which is already standard in component design. Services should be designed by separating the **interface** from the **implementation**. Therefore, the separation of interface and implementation is forced through this AOP approach leading to better application designs.

```java
void start(BundleContext context) throws Exception {
  eventserviceimpl = new EventServiceImpl();
  // register Eventsystem as an AOP Service
  eventsvcreg = ((AOPContext) context).registerAOPService(
    lEventService.class, eventserviceimpl, null);
  // register Eventsystem as a normal Service
  context.registerService(
    EventServiceImpl.class.getName(), eventserviceimpl, null);
}
```

As shown in line 7, the interface class-instance is used to register the service in the service registry. This allows the adapted Nanning library to choose the interface’s class-loader to generate the proxy with that class-loader.
A special service registration for AOP services is required, the developer has to pro-
gramm it explicitly. Two solutions would be feasibly to make the AOP service registration
transparent. A first solution could be to extend the OSGi API and specification to work with
the AOPContext only. A developer would not be aware of the changes in the container. A
second solution would be to generate the AOP special registration code by a transformation
tool. The service plugin description or the container description could then specify in which
cases to generate the AOP registration code before the classes are loaded. The combination
of both solutions would be ideal. No pre-compiler transformation would be required as the
OSGi API already provides the right AOPContext interface. Specifying the AOP require-
ment for a service plugin in descriptions would allow the developer to control the proxy
generation. This may be favoured for services with high probability of being adapted as it
would otherwise lead to unnecessary overhead.

**OSGi Container Description (OCD)** In a next step the capabilities of the container have
to be announced to the service plugin requiring such a container. The OSGi/Nanning con-
tainer description, as shown below, is required in order to be able to match a service plugin
with the container.

```xml
<OSGiContainer id="osgi-nanning" name="OSGi-nanning">
  <Property name="container" value="osgi-kf"/>
  <Property name="container-version" value="1.3.3"/>
  <Property name="aop" value="nanning"/>
  <Property name="aop-version" value="0.9"/>
  <Property name="version" value="0.1.0"/>
</OSGiContainer!>
```

### 4.2.2 OSGi/PROSE

The Nanning approach can be used on a wide variety of virtual machines as the only require-
ment is a J2ME/CDC VM or higher. Devices with more resources are capable of running
a more enhanced virtual machine. This allows us to take advantage of new functionality
available in such VMs. Using Jadabs on desktop machines therefore results in a middleware
platform with the more enhanced dynamic AOP mechanism of PROSE [132].

PROSE stands for (PROgrammable extenSions of sErVICES) and allows us to extend Java
programs at run-time. It is implemented as a Java virtual machine extension that can perform
interceptions at run-time. The PROSE VM extension is loaded together with the VM when
the container is started.

PROSE uses the Java Virtual Machine Debug Interface (JVMDI) [163]. With JVMDI,
crosscutting concerns can be inserted into the running application. Such applications do not
require to be prepared for AOP and can be extended anytime. When PROSE aspects are
inserted and woven, JVMDI enables the use of all method entries and exits, field sets and
gets as interception points. This approach contrasts with Nanning, where interception points
can only be set on AOP registered service interfaces.
4.2. Segmented Containers

As PROSE nicely integrates into the VM, the OSGi container does not have to cope with PROSE related functionality. To announce the PROSE platform, a container description is required.

**OSGi Container Description (OCD)** To match service plugin against the OSGi/PROSE container a similar container description to OSGi/Nanning is proposed.

    <OSGiContainer id="osgi-prose" name="OSGi-PROSE">
    <Property name="container" value="osgi-kf"/>
    <Property name="container-version" value="1.3.3"/>
    <Property name="aop" value="prose"/>
    <Property name="aop-version" value="1.2.17"/>
    <Property name="version" value="0.1.0"/>
    </OSGiContainer>

4.2.3 Discussion of dynamic AOP in SOA

OSGi services can be registered and unregistered in an OSGi container depending on the changes in the environment. Therefore, the programmer of such services has to keep track of changes in the environment. Services which are removed from the system may leave a client in an improper state with dangling pointers to unavailable services. Another problem occurs when a service user was not designed for multiple services. As a dynamic system may change anytime, new services can be registered which would then change the behaviour of the service client.

**Stale References** The key concept behind service-oriented architectures is the registration of a service by a provider and the lookup of the service by a client (Figure 3.1). In case the service provider is removed, all its registered services will get unregistered. This has a significant drawback as it leads to stale references (see [116], p. 72). The users of that service have to check before its usage if the service is still available. The service user therefore needs to check the references himself, as in case a service is removed, references would point to non-existing locations and throw a NullPointerException.

OSGi proposes a solution to the stale references problem by using the Service Tracker and Service Listener classes. They allow the user to listen for service events which are generated once a service is registered and unregistered. The observer pattern behind this mechanism has to be implemented by the bundle developers themselves.

Another solution to the stale references problem was proposed with the Service Binder [26], an automatic service dependency management. The registering and unregistering of services is externalized into an XML descriptor. The descriptor defines the methods which have to be called to register a service once it is available or unregister it when the service is removed. This observer concept is very similar to the Service Tracker. The Service Binder extends this functionality by a cardinality checker. This allows it to track the life cycle of the bundle depending on the numbers of required services. This solution requires a listener registration mechanism in every service client and XML files which have to be maintained.
In the Service Binder concept, the service has to provide observer methods. With our proxy approach, the observer pattern is hidden inside the proxy and transparent to the user.

4.3 Monolithic Containers

Monolithic containers are required for two reasons. First, a segmented container cannot be run on small devices like a mobile phone with a CLDC VM. Second, the underlying system does not always support a segmented container, e.g., the .NET Framework or the Compact Framework [70].

4.3.1 mOSGi/CLDC for mobile phones

In J2ME/CLDC all classes that interact with each other must be present in the same process context, the MIDlet-suite. This introduces an important limitation as modules downloaded at different times cannot communicate directly with each other. To solve this problem, a monolithic container was implemented for the mobile phone and a local event mechanism for inter-container communication.

The monolithic container uses the dependency injection [52, 98] principle to decouple components. A component declares the dependencies it requires instead of creating them itself. A container will take over the required connectivity. Many open source projects like PicoContainer [124], Spring [156], Metro [99], OSGi [116] use this concept. For our purposes, we use the service locator and interface injection. The former means components first have to be looked up before they are returned from the container. Interface injection means that an implementation of a component interface is returned on a lookup.

Our dependency injection container for CLDC implements the OSGi API with a simple service lookup mechanism in the mOSGi container. OSGi components written for CDC or higher VMs can now be used on CLDC if only the CLDC API is used. In mOSGi, services have to be started explicitly in the MainMIDlet (as shown below). The mOSGi container is created (line 6) and then the bundles are started by the call to startBundle(…) (lines 11–13). The bundles correspond to other MIDlets which can register their services inside the container as if it were a normal OSGi container.

```java
class MainMIDlet {
    public void startApp() {
        //create the container, set some properties
        OSGiContainer osgicontainer = OSGiContainer.getInstance();
        osgicontainer.setProperty("name", "nokia 6600");
        osgicontainer.setProperty("log4j.priority", "debug");

        // start the osgi bundles
        osgicontainer.startBundle(new LogActivator());
        osgicontainer.startBundle(new JxmeActivator());
    }
}
```
By using the OSGi API for monolithic containers, we are able to use the same code-base for services in segmented as well as in monolithic containers. The main difference is that a service plugin with different bundles for a segmented container is assembled in monolithic containers into one service plugin, the MIDlet-suite in CLDC.

**Plugin Container**

Since mOSGi does not support dynamic AOP, the adaptation has to be performed on a higher layer. The Jadabs-PSM provides a plugin mechanism where each service plugin is implemented with its own monolithic container. Local service plugins may communicate over the local loop-back mechanism with each other. In the Jadabs-PSM, the service plugins have to be aware of possible adaptive requirements at design time. Requirements which may change or should be kept open for later changes have to be encapsulated in a service plugin. Service plugins are then using other service plugins over the local loop-back like a local event mechanism.

Figure 4.3 shows the mOSGi architecture developed for J2ME/CLDC which was tested on the Nokia 6600 and the Ericsson P900 mobile phone. The service manager is responsible for the coordination and activation of the service plugins. The service manager maintains two connections, first one with the remote infrastructure (1) and second one with the local loop-back (2). The remote communication is based on the usual peer-to-peer layer and uses a Bluetooth connection. It allows us to use the service discovery mechanism in the mobile environment. The local communication uses a micro-service layer which is a trimmed down local event mechanism.

The service manager handles the service plugins on the mobile phone. On mobile phones, a Java Application Manager (JAM) [74] is responsible for the activation of installed MIDlet-suites. With the Jadabs-PSM model, an additional service plugin manager architecture is required above the JAM. This service manager is responsible for the automatic starting and stopping of the required service plugins which are packaged and deployed in a MIDlet-suite.

**Setup of local loop-back**

When the user starts a service plugin, all dependent service plugins need to be started in addition to the own plugin. This is coordinated by the service manager which dynamically starts other service plugins. In MIDP, a PushRegistry mechanism performs the startup of other MIDlet-suites. This mechanism is now used to coordinate the startup of additional service plugins. Once the other plugins have been started a local loop-back communication path is opened to these services. This allows the services to communicate among each other and use the local loop-back as a local event mechanism.
The local loop-back implements a simplified JXME interface for a smaller footprint local communication. Service A and B communicate with each other (Figure 4.3 - 2) and with the service manager by using the micro-service bundle.

**Service plugin discovery and activation**

For the purpose of autonomous adaption the service manager is responsible for the matching of services to the already installed service plugins and acts as a passive service manager as described in section 3.6.1. Therefore, the service manager sends out a matching filter in its mobile environment (Figure 4.4 - 1). The matching filter contains the locally installed service plugin, extension-points, the platform and the container description. In the mobile environment, a more powerful node with an active service manager, providing services for mobile phones, matches the received filter against its providing services. After a matching service plugin descriptors has been found, a port handshake is required with the mobile phone (2). The port is used later on to automatically startup service plugins over the local loop-back. The mobile phone sends a free available port which is required to activate the service plugin later on. With the port a temporary .jad file is generated on the download server. The .jad file describes the MIDlet-suite and is used for the installation on the mobile phone. The matching device sends back the service descriptor which includes the download URL of the service plugin (3).

Once a new service plugin descriptor is received by the mobile phone, the plugin can be
downloaded. For the mobile phone architecture, the downloading of additional code can be done automatically by using the "Over The Air" (OTA) [74] mechanism (4). OTA allows the service manager to download a MIDlet-suite which includes jar and jad files over GPRS from a HTTP server. The service plugin is packaged in a MIDlet-suite. When the MIDlet-suite is downloaded and installed, the startup port is registered in the JAM. The initially configured port is used for the Push-Registry mechanism. This allows the service manager to send a datagram packet to the specified port and start up the service plugin (5). After startup, a micro-service loop-back connection is opened to the service plugin for internal communication.

Adaption of dynamic service plugins

Once new service plugins are available and activated, they may adapt already activated plugins. With the service plugin descriptor, the service manager is able to properly connect the extensions with the extension-points of different service plugins.

Figure 4.3 shows a new dynamic Service–B which matched the service filter request and the service description was downloaded. Apparently, Service–A is using the InterfaceA extension from the service manager. With the new InterfaceA extension provided by Service–B the extension-point from Service–A can be adapted. Therefore, the service manager deregisters itself as the responsible for InterfaceA and registers the InterfaceA extension of Service–B. This allows Service–A to use the extension of Service–B on the next usage of the InterfaceA call. The registration and deregistration of extensions is performed over the already available local loop-back mechanism.
4.3.2 mOSGi/.NET

The .NET run-time environment provides the concept of application domains [159]. These application domains are the deployable units which can be unloaded again. Application domains are built with assemblies and compiled into one application domain. Method calls performed over different application domains have to use .NET Remoting [136]. By having a .NET infrastructure we profit from several advantages:

- Devices with no Java run-time but .NET capabilities can be integrated into the mobile environment.
- Interoperability of available .NET applications with a .NET Jadabs middleware is easier than with a Java implementation.
- With a .NET implementation the heterogeneous capabilities of the proposed pervasive middleware can be shown.

For the .NET architecture a monolithic container approach was chosen. The reason is that assemblies which represent bundles are loaded into an application domain and cannot be unloaded again from the application domain. Therefore, an application domain represents a service plugin with its monolithic mOSGi container.

The mOSGi container for .NET has a similar API as the mOSGi container for J2ME-CLDC. The container was ported to C# for the .NET Framework on desktop machines and Compact Framework for PocketPC on handhelds.

Adaptation capabilities

For the dynamic adaptation in mOSGi/.NET we considered a two way approach. Firstly, we use a concept similar to the J2ME/CLDC container. Instead of programming a completely new local event mechanism, the .NET Remoting can be used on .NET Framework devices. For Compact Framework devices, a micro-service loop-back mechanism would have to be implemented. As part of this thesis, we started a second approach by implementing a dynamic AOP-Engine [55] comparable to PROSE. This dynamic AOP engine currently runs only on desktop machines as it requires debugger and profiler functionality not available in the compact framework. The implementation has yet to be connected to the mOSGi/.NET container for dynamic adaptation. The current AOP-Engine implementation requires user intervention to insert aspects. As the insertion and activation of aspects was to be performed automatically, a COM interface between the mOSGi/.NET container and the AOP-Engine is required.

The engine allows us to exchange methods and support the insertion of stubs at the beginning and/or at the end of a method. These requirements are part of the basic functionalities of an AOP system and are called around, before, and after crosscuts.
4.3. Monolithic Containers

**Figure 4.5: AOP engine overview**

**AOP Engine Architecture** All the components of our dynamic AOP-Engine are shown in Figure 4.5. The program which will be crosscut at run-time is shown on the lower left side with the CLR in the middle as shadowed box. The CLR is divided by the process boundary, indicating that a switch from the running program to the debugger would stop all running threads in the program and switch to the debugger.

The top part shows the AOPDebugger and its subcomponents. These are the common parts of a debugger for managed applications, as suggested by the CLR documentation. The AOPDebugger controls the execution of all other components, it also controls the AOPProfiler and contains the large data structures that are needed to manage the system. Finally, it also provides the AOPController as a user interface to control the aspect insertion and removal. The AOPDebugger and AOPProfiler use the corresponding CLR debugger interface and CLR profiler interface [159].

The approach taken by PROSE and Woole [145] suggests implementing dynamic AOP using the Java Platform Debugger Interface (JPDA). Whereas the two projects use two different layers of the JPDA, they have in common that both intercept code by setting breakpoints on places where a join-point is needed. When a breakpoint is reached, the thread continues its execution in the engine to execute the aspect advice. In .NET however, reaching a breakpoint suspends all threads before continuing its execution. The debugger approach is therefore not applicable to .NET since on reaching a join-point, the whole application would
To avoid this problem, we use a combination of the debugger and profiler API. Instead of setting a breakpoint, a stub is inserted into the method bodies by the AOPProfiler.

The AOPProfiler consists of two almost independent parts: the first contains the callback interface, whose methods are called by the CLR when certain events happen. These events include loading and unloading of modules, loading and unloading of classes, suspending and resuming of threads, Just in Time Compilation, entering and leaving of methods, memory allocation, and garbage collection. At Just-in-Time compilation, the AOP-Engine intercepts and replaces one method for another.

The second part of the AOPProfiler contains the communication classes which are used to communicate with the AOPDebugger and the AOPController. To communicate with the AOPDebugger, the AOPProfiler uses a specialized shared memory interface, and to communicate with the AOPController, it uses COM’s interoperability possibility. COM Interop is a feature provided by the CLR that allows the use of managed classes over COM. The AOPController is written in C#, so that it is possible to use the managed reflection API to handle the aspects. The main task of this component is to manage the aspect libraries and to provide the run-time environment for them.

Finally, we have the user providing aspects in the bottom right corner of the picture. An aspect can be written in any language as long as it is both supported by the CLR and compiled to a standalone DLL before execution. Unlike other AOP implementations, we do not need external files to describe the aspect. It is completely self-contained and stores the information on where and how methods should be crosscut. An aspect DLL must at least contain one class that implements the Aspect interface. The AOPController will use this interface to ask the aspect which functions should be crosscut and what kind of crosscuts should be made. At a later stage, when the crosscut locations are hit, the aspect advice is executed.

Therefore, the dynamic AOP-Engine consists of the AOPDebugger, AOPProfiler, and AOPController which get loaded into the process when the program is started.

## 4.4 Distributed Infrastructures

A distributed infrastructure is another core functionality in mobile environments. Many different distributed technologies are used in infrastructure environments like CORBA [68, 109], RMI [105], Jini [12], JMS [64], JXTA [24, 80]. These technologies lack the ability to suite mobile environment requirements.

There are several reasons why current distributed infrastructure technologies are inappropriate:

- **Platform restricted** like Jini and RMI which are Java only solutions.
- **Centralized** like CORBA, Jini, and JMS which require a centralized infrastructure for service registration.
Heavy weight like CORBA, Jini, JMS, JXTA which require at least the capacity of desktop machines to be run on. Their footprint cannot be decreased to be usable on small devices like handhelds or mobile phones.

Mobile devices used in mobile environments have different requirements on the distributed architecture than infrastructure devices. To solve all the above listed disadvantages a new mobile distributed infrastructure has to fulfill the following requirements:

Decentralized which allows mobile devices to connect to each other even when there is no centralized infrastructure available.

Heterogeneous to allow devices with different platforms and systems to communicate with each other.

Small footprint to be able to use it also on small devices like a mobile phone.

Extendable which allows more resource powerful nodes to extend their distributed architecture to a more appropriate infrastructure, and allows to plugin other protocols used by other devices.

4.4.1 Peer-to-Peer

Peer-to-peer (P2P) networks are proposed as distributed infrastructure solutions in decentralized and centralized environments with heterogeneous network structures. Until now the P2P networks have been mainly used in infrastructure networks. Napster [102], Gnutella [59], and P-Grid [1, 5] are the most prominent ones. Whereas Napster makes use of a client server architecture to find files, Gnutella is completely decentralized but inefficient whereas P-Grid is an efficient decentralized peer-to-peer infrastructure.

In general, P2P does not mean a decentralized system only. In case peers provide direct services for other peers they provide a decentralized service. On the other hand, specialized nodes in a peer-to-peer system can still provide server functionality making a centralized infrastructure for a specific service. Peer-to-peer systems are therefore a mix of centralized and decentralized infrastructures where nodes connect to it without knowledge of the internal structure.

Different standards, or de facto standards, exist which enable a peer-to-peer infrastructure, for example JXTA, SIP [143], and UPnP [172]. In the following we explain these protocols as far as it is required to explain the differences and our choice of JXTA as our core communication infrastructure. The core infrastructure should serve two purposes, service discovery and basic messaging. New services can then be loaded to support other communication possibilities which fit the applications needs. Basic messaging can be used for applications on small devices to communicate with other devices in the P2P network.
4.4.2 JXTA

JXTA is supposed to span programming languages, platforms, and networking infrastructure. JXTA is based on protocols which establish a virtual network on top of existing networks, hiding their underlying physical topology (Figure 4.6). The peer-to-peer infrastructure is based on the idea of peer groups which enables the organization of peers in groups of virtual entities. The JXTA protocols also address the advertising and discovery of resources and communication among peers. JXTA defines a super-peer virtual network [169] to locate peers and resources efficiently in large networks.

XML documents are used to advertise services and resources. For communication applications may use pipes to send and receive messages. Pipes are reliable, unreliable, unicast or multicast endpoints that deliver messages between peers. JXTA’s architecture is a layered set of protocols as shown in Figure 4.7.

- **Peer Endpoint Protocol** – Routing of messages which can be directed to any of the above layers.
- **Peer Resolver Protocol** – Generic query protocol for finding specific resources.
- **Rendezvous Protocol** – Propagates messages to other rendezvous peers which connect to the super-peer virtual network.
- **Peer Discovery Protocol** – Resource search for peers, peer groups, and pipes.
- **Pipe Binding Protocol** – Addressable messages which are directed to specific pipes.
4.4. Distributed Infrastructures

- **Peer Information Protocol** – Monitoring for peer information.
- **Peer Membership Protocol** – Security to gain access to peer groups.

JXTA provides a virtual network over the Internet. Its abstraction layer adds an overhead to the normal networking. Nevertheless, its clean messaging protocols can be implemented for different systems and can be used as a basic communication platform. With this in mind JXTA provides a powerful messaging infrastructure which can integrate mobile environments with infrastructure based environments.

**JXTA-JXME**

JXTA solves two of the stated requirements for mobile environments: decentralization and heterogeneity. It can be used in decentralized as well as in centralized infrastructures. The protocols allow its usage in heterogeneous infrastructure with devices running on different systems. However, it lacks the possibility to be run efficiently on small devices like on mobile phones or handhelds.

Therefore, the JXTA community has started the JXTA for the J2ME (JXME) [79] project to bring the JXTA protocols to such devices. The footprint of JXME has been decreased significantly. JXTA has a footprint of about 2 MByte, whereas that of JXME is around 50
KBytes. To decrease it in such a way, many protocol specification had to be left away or simplified.

JXME-JXTA is a library which cannot be easily extended with additional functionality. Extensions made for JXME-JXTA are explained further down.

4.4.3 UPnP

Universal Plug and Play (UPnP) is an architecture to connect devices in mobile environments. It is used for home networks, in small businesses, or ad hoc environments. It uses zero-configuration and automatic discovery of devices and services in its environment. Once a device joins a network, it obtains an IP address, publishes its services, and discovers services of other devices.

UPnP was defined as a stack of protocols including IP, UDP, HTTP multicast and Simple Service Discovery Protocol (SSDP) [60]. This requires to run a simple HTTP server to react to multicast service discovery messages. SSDP allows to describe services and make them available for other devices as so called control points. Once control points are discovered, they can be used to call actions on the devices. To do so, a remote procedure call stack, based on IP, TCP, HTTP and SOAP is used. Many devices are implementing UPnP ranging from printers over handhelds up to desktop machines. It has become the de-facto standard in local networks with Windows machines.

Nevertheless, UPnP is more of a local service discovery mechanism. The scalability into large networks remains an open issue. As the discovery is based on UDP multicast, such messages are mostly blocked by routers and firewalls. An open peer-to-peer system has to tackle these problems and solve them for a wide range of different network topologies.

The JXTA protocols proposes so called rendezvous and gateway nodes which allow to connect internally with external JXTA networks. By providing a peer group concept, the discovery of services can be restricted to groups of certain interests or responsibilities. Furthermore, JXTA has been designed from the beginning to be used in heterogeneous infrastructures. With its super-peer virtual network, its service discovery can scale to a large amount of different services.

4.4.4 SIP

The Session Initiation Protocol, SIP, is a signaling protocol for telephony, presence, events notification and instant messaging. SIP defines the protocol to locate a user’s phone over the Internet whereas users are able to register their phone on different location. Each user has a unique SIP URL like an email address, e.g., sip:frei@iptel.org which can also be mapped to a normal telephone number. The SIP URL is registered on a SIP server with the current IP address of where the user’s phone is located. Once the user moves and takes the phone with him, the new IP address is updated on the SIP server. The user registration mechanism allows a calling participant to setup a call to a user without knowing the present location of the user’s phone. Once the users phone has been located, an audio and video session can
be established. To setup the multimedia session, SIP uses the Session Description Protocol (SDP) [66]. This allows the participants to negotiate the multimedia session at initialization.

SIP is mainly used for Voice over IP (VoIP) applications but it can also be used for presence applications or instant messaging. SIP for Instant Messaging and Presence Leveraging Extensions (SIMPLE), was defined as an IETF draft [142] and allows to locate user's presence.

SIP was designed for the infrastructure based environment and relies on an IP infrastructure. Several issues in mobile environments complicate the usage of the SIP protocol:

- Ad hoc environments do not provide connections to SIP servers. Devices therefore need a way to locate users in its mobile environment.

- Mobile environments are not only IP based, for example mobile phones do not yet support an IP stack over Bluetooth. Devices which only support Infrared, or ZigBee would be disconnected from the mobile environment.

- Gateways in mobile environments provide the devices with private network addresses and usually do Network Address Translation (NAT). Specially configured gateways are required which would do the translation of SIP messages.

SIP infrastructure can also be referred to as an overlay network infrastructure. Whereas SIP focuses on presence and multimedia applications, it lacks the possibility of being used as a general peer-to-peer network as presented by JXTA. JXTA is independent of the services which are run on it. It provides the distributed infrastructure for the services.

4.5 Jadabs – Distributed Infrastructure

We argue that a general overlay network is required for mobile environments which can be extended with additional functionality when needed. An overlay network which is already specific for special applications as in SIP for audio/video applications, it cannot be extended anymore. For Jadabs we need a peer-to-peer infrastructure which can be run in ad hoc mobile environments as well as in infrastructure environments. The peer-to-peer system is responsible for the routing of messages on the overlay network of mobile devices with different transport technologies. For world-wide discovery, JXTA has defined a super-peer virtual network to enable a scalable Internet solution.

JXTA and JXME have been extended in two ways. First, Jadabs requires a general service discovery mechanism which allows it to find plugins and bundles provided by other peers or which are available through repositories. Second, Jadabs-JXME extends the JXME implementation to take advantage of the adaptive Jadabs middleware.

Service Discovery Layer

A new service discovery layer is required to cope with the service request messages in the mobile environment and coordinate the required services on different nodes. As explained
in section 3.6.1, the service manager is in charge of discovering other services and matching them against required extensions.

Discovery messages are sent on a propagation pipe where each node in the mobile environment is listening on. A propagation pipe is a communication abstraction for overlay networks. Several different network implementations can be plugged to this pipe which allows a uniform communication. A propagation pipe for the mobile environment is created by default. Figure 4.8 shows two different mobile environments, A and B, which are distributed over an infrastructure network. Each node in the two mobile environments is listening on the propagation pipe for its mobile environment. Gateways, on the boundary of the mobile environment with the infrastructure environment, do not route mobile environment discovery messages. Therefore, the two mobile environments are separated.

The core messaging layer is basically used for service discovery in mobile environments. In addition it also allows us to use the messaging layer as a general communication layer for applications. For example nodes in the mobile environment are using an application which communicates with nodes in the infrastructure environment or in other mobile environments. The mobile environments are therefore interconnected over the JXTA backbone, as shown in Figure 4.8. The gateway nodes are responsible of routing messages from the mobile environment to infrastructure nodes through the established JXTA network.

The JXTA/JXME solution differentiates from the SIP infrastructure because JXTA is used for service discovery and simple message communication. In Jadabs JXTA/ JXME is used as the core messaging and service backbone for heterogeneous devices and network connections to easily pass NAT, firewalls, and gateways. SIP is already a specific service for audio and video applications which is run in parallel to the used JXTA/JXME stack. In this
4.5. Jadabs – Distributed Infrastructure

Figure 4.9: JXME-OSGi protocol stack

case, JXTA/JXME is only used to coordinate the configuration of the SIP infrastructure for devices participating in the SIP infrastructure.

4.5.1 Jadabs-JXME

Whereas the JXTA-JXME provides JXTA's core messaging protocol, its implementation is not yet fully applicable to mobile environments. We were missing some key issues which we were able to implement with the proposed Jadabs middleware. This way we can fulfill the last two requirements for the mobile distributed infrastructure: small footprint and extendable.

- As not every device is capable of running all possible communication protocols like WiFi, Bluetooth, Infrared, and so on, only the ones which are apparently needed should be loaded and activated. This required JXME-JXTA to be separated into different bundles with appropriate service functionality. The services are then provided with their service descriptors for matching purposes.

- For dynamic adaptation, JXME was extended with the Jadabs-PSM services. This required an appropriate local event mechanism for mobile phones and the AOP registration mechanism for handhelds.

In the first implementation we proposed and implemented a similar decentralized infrastructure for small devices in the range of handhelds [82]. Later on, the JXTA-JXME's proxyless version came out which was basically the approach we followed. From then on, the code from JXTA-JXME was adapted to fit our infrastructure needs.

Figure 4.9 shows the different components provided by the new JXME-OSGi implementation. Each component is implemented as an OSGi bundle. This allows us to use only parts of the architecture without loading everything into memory.

Core. The core layer is a messaging infrastructure which is compatible with the normal JXTA messaging specification. This allows JXTA and Jadabs-JXME to communicate directly with each other.
Transport. The transport layer implements different transport types, e.g., UDP, TCP, Bluetooth. Each transport can be loaded and activated once it is required.


Only the core message layer from JXME is JXTA specification compatible. The higher service layer in JXME does not implement the JXTA protocols completely. The reason is, that otherwise the implementation would end up in re-implementing JXTA. Nodes capable of running a full JXTA implementation can still load the whole JXTA library as a bundle.

Due to the clear separation of the Jadabs-JXME implementation, several platforms can be supported. Figure 4.8 shows devices with supported JXME implementations and depending on their capabilities they are using different transport protocols.

CLDC. The three layers are implemented for different systems. Each component takes between 5 to 20 KByte code size. The same code for the service and core layer can be run on mobile phones with J2ME/CLDC and on handhelds with J2ME/CDC. Depending on the resource capabilities of the device, the whole JXTA library can be added at a later time.

.NET F/CF. JXME is implementing the messaging JXTA protocols in Java. A .NET implementation was missing which allows us to integrate devices of different platforms. The UDP, JXME-Core and JXME-Service were implemented for .NET Framework and Compact Framework. The JXME.NET implementation were then integrated into the mOSGi.NET container.

4.6 Discussion

The Jadabs middleware platform including the JXTA/JXME distributed infrastructure provides a powerful platform for resource constraint devices. To enable adaptation for the wide range of different devices two types of containers are proposed: segmented-adaptive and monolithic-dynamic. By using dynamic AOP for the adaptation in segmented containers, applications can be adapted transparently. To account for diverse capabilities of various platforms, we used the two AOP projects Nanning and PROSE. In order to apply the Nanning solution to a dynamic adaptation the SOA container OSGi as well as Nanning itself had to be extended.

Monolithic containers can be used on resource constraint devices like a mobile phone or on platforms which do not provide loading and unloading of fine grained components. Adaptation can be performed in monolithic systems by adapting the exposed services. Where an exposed service communicates over the local loop-back and through a registration mechanism, the communication can be switched to other local services.

The different containers provided reflect the heterogeneity of available platforms. It is not feasible to provide the same container to all platforms. Each platform has its own peculiarity and a container has to cope with such diversities. The model and dynamic service
composition proposed in previous chapter is able to cope with these differences by providing metadata for a heterogeneous infrastructure.

Whereas different containers are provided for different platforms a common distributed infrastructure is required for the communication among the different nodes. As a basis we used the messaging infrastructure proposed by JXTA. This allows us to interconnect devices in an overlay network, abstracting the network topology. By using the adaptive properties integrated in the Jadabs containers, new network stacks can be added when available or required.

The Jadabs middleware is run on a wide range of platforms like J2ME/CDC, J2ME/CLDC, JRE, and .NET F/CF and devices like mobile phones, handhelds, gateways and normal desktop machines. Compared to other middleware solutions like GAIA, CORBA, or Jini, we implemented a decentralized and small footprint adaptive architecture which can be run on resource constraint devices.
Seite Leer / Blank leaf
Chapter 5

Application Scenarios

5.1 Introduction

We have built several applications using Jadabs. The following examples show the feasibility of one or several aspects of the architecture.

**Event System** shows how an application can be extended with an event system on top of two different segmented containers. Benchmarks are given for both scenarios to show its feasibility on desktop machines and handhelds.

**Autonomous Cooperating Robots** presents an architecture to prototype traffic scenarios. The robots use Jadabs to setup a cooperating infrastructure. In this example, adaptation is given on different layers, on the container, by dynamic services and in the application by using logic extensions.

**Messenger Scenario** illustrates the adaptive behaviour of the container in an environment with different messenger protocols. The protocols are adapted to allow messenger communication among each other.

**Web Service** applications have been deployed to mobile phones to access Web services over a Bluetooth infrastructure. This scenario shows the automatic service composition on resource constraint devices in a mobile environment.

5.2 Event System

We would like older applications to be able to deal with events without having to redesign or change them in a significant manner. We propose to add dynamically a software layer that deals with the new events and acts as a translator for the older, less capable application.

A second example are changes in policy that may force applications to cope with new types of events not foreseen at the time they were designed. An automobile control system, for instance, may be required to generate an event when the combustion in the engine is
Figure 5.1: Application extended with event system and dynamic AOP

less than perfect. The event can then be used to monitor the pollution level caused by the automobile. The ability to monitor the combustion is present in most modern cars but the software that turns the information into an event is probably not. Ideally, it should be possible to dynamically add the ability to generate such an event without having to change the software already installed in the automobile.

These examples are similar in that they require a dynamic extension of already deployed and possibly already running applications. A short overview is given of how an application is extended with events without a prior knowledge of an event system. An application designed and already running can be extended to publish relevant events and react to events produced by its surrounding infrastructure.

5.2.1 Architecture

Figure 5.1 depicts the basic architecture of the system. The architecture is based on the OSGi/PROSE container or the OSGi/Nanning container. We assume there is an application running in this container and that this is the application that will be adapted as needed. The dynamic service manager which uses the JXME messaging service for communication were started by default.

Through the service manager the event system is discovered. This new functionality is provided by two service plugins. The first is the core event system plugin which contains the event system functionality and allows to publish and subscribe for events. The second plugin is an application aspect service plugin which contains the application specific exten-
5.2. Event System

In case the extension-points of the event system service and aspect service match the platform, they can be downloaded and activated. Once the aspect plugin with the dynamic AOP functionality is activated it inserts crosscuts into the application. In addition, it is in charge of creating and publishing events and subscribing for relevant events and reacting accordingly. The adaptations that we consider are of two generic types. The first type of adaptation consists of endowing an application with the ability to generate events (either because it did not generate any event at all or because it did not generate the necessary events). The second type of adaptation transforms applications into consumers of events (again, either because the application did not accept any events to start with or because it did not accept the necessary events).

Event system implementation

The event system we use is based on a publish and subscribe model [38, 50, 67, 126]: a producer node publishes an event and a consumer node may subscribe for this event and be notified of the occurrence of the event. TPS [150] is a type based publish and subscribe model built on top of JXTA [24, 80]. In our implementation we use a similar approach to TPS. As we rely solely on JXME, we implemented our own event system on top of this messaging layer.

The implementation of the event service uses the JXME messaging layer to send events to other nodes. Before sending an event, it gets serialized into the JXME message format which is then sent over the peer-to-peer network. On the other nodes, the message gets deserialized back to the event type. The subscription to events is done through an event filter following the type- and attribute-based subscription model introduced in [126, 150].

As we are targeting a decentralized infrastructure, each node manages its own subscriptions. When a node subscribes for an event, no subscription messages are sent out. On the other hand, when an event is published, it has to be sent to all nodes to match a subscription.

5.2.2 Descriptors

The architecture and all discoverable components have been provided with appropriate descriptors as proposed in the Jadabs model. For the purpose of this event system example, a set of descriptors are required. This allows us to dynamically extend the application with new functionality. When a node reaches a mobile environment where the event system is provided, it automatically gets discovered by the node without prior knowledge of an event system.

Platform & Container

The platform descriptor is shown below and allows us to run the application and the two containers: OSGi/PROSE and OSGi/Nanning on it.

It runs on a Laptop with an Intel Pentium (line 4) and Linux (line 5). The platform contains a network interface (line 10) which consists of an ad hoc wireless network where it gets
an IP address from DHCP. The platform is able to run an OSGi/PROSE and OSGi/Nanning container (line 7,8). The container descriptions are shown in section 4.2.

```xml
<Platform id="lap.wlab.ethz.ch">
  <Property name="processor" value="pentium" />
  <Property name="os" value="linux" />
  <OSGiContainer id="osgi-prose"/>
  <OSGiContainer id="osgi-nanning"/>
  <Netface type="wlan-adhoc" ip="dhcp"/>
</Platform>
```

Application

The application component is described by the following plugin descriptor. It contains a main application extension which allows other services to use this application (line 6). The application code is provided by the bundle and referenced accordingly (line 10).

```xml
<OSGiServicePlugin uuid="jadabs:app!ication:1.0.0:opd"
  service="ch.ethz.jadabs.eventsystem.application">
  <Extension id="Application"
    service="ch.ethz.jadabs.eventsystem.application.AppMain"
    description="provides a main application"/>
  <ServiceActivatorBundle activator-uuid="jadabs:application:1.0.0:obr"/>
</OSGiServicePlugin>
```

Event System

The event system is completely independent of the application. This allows us to reuse the event system package for many other application scenarios where an event system is required. The descriptor defines an extension-point and requires the JXME layer for sending and receiving events (line 5). The event system defines an extension to provide its service to other services (line 10).

```xml
<OSGiServicePlugin uuid="jadabs:eventsystem:1.0.0:opd"
```
63 5.2. Event System

```
<Extension-Point
  id="Extension/id:GroupService"
  service="ch.ethz.jadabs.jxme.GroupService"
  description="requires service from JXME layer"/>

<Extension
  id="EventService"
  service="ch.ethz.jadabs.eventsistem.EventService"
  description="provides event system service"/>

<ServiceActivatorBundle
  activator-uuid="jadabs:eventsistem:1.0.0:obr"/>
</OSGiServicePlugin>

Aspect

The aspect service is the main plugin which links the application and the event system by
using the dynamic AOP functionality of PROSE. The aspect descriptor therefore defines
three extension-points, first one for the application (line 5), second the event system (line
10), and a third one for the container providing the PROSE environment (line 15).

```
<OSGiServicePlugin
  uuid="jadabs:appaspect:1.0.0:opd"
  service="ch.ethz.jadabs.appaspect">

<Extension-Point
  id="Extension/id:Application"
  service="ch.ethz.jadabs.application.AppMain"
  description="requires the application"/>

<Extension-Point
  id="EventService"
  service="ch.ethz.jadabs.eventsistem.EventService"
  description="requires the event system service"/>

<Extension-Point
  id="Container/aop:prose"/>

<ServiceActivatorBundle
  activator-uuid="jadabs:appaspect:1.0.0:obr"/>
</OSGiServicePlugin>
5.2.3 Benchmarking

In order to test the viability of the approach, we conducted a number of experiments. First, the event system was measured with the OSGi/PROSE container and second with OSGi/Nanning.

We tested the different building parts of the architecture and the scalability of the whole architecture for messaging and the event system in a wireless environment. The overhead of PROSE alone was measured in [132].

The local benchmarks were performed on an IBM Laptop A31 running Red Hat 9.0 which is our master laptop. For remote benchmark machines we used IBM Laptops R32 running Red Hat 7.3 with built-in wireless cards. In a first benchmark, we measured the behaviour of the architecture on the same node and on two different nodes with a wireless connection in ad hoc mode. In the scalability benchmark we increased the nodes involved up to 9 and analyzed the behaviour with various wireless parameters.

The measurements in the first benchmark have a standard deviation of less than 1%. In the scalability benchmark we achieved less than 5% deviation because during the access point test, the access point was used by other people.

Local, and 2 Node Benchmark

In the first measurement section the benchmarking parts of the architecture have been categorized as follows:

**PROSE/JXME.** Since PROSE and JXME run as a basic resource on every node, we measured the time of sending messages between two different nodes. In a first test the two different nodes were running on the same machine. The second test sent messages over a wireless connection in ad hoc mode.

**Event System (ES).** As the event system is the extension to the application we measured the type- and attribute-based overhead.

First, we analyzed the overhead for the application when started including the service manager. Second, we measured JXME and the event system extension (Figure 5.2). It shows that the round trip time (RTT) of a basic message between two different JVMs on the same machine takes 3.5 ms on JXME whereas on two different machines and over wireless it takes 10 ms. When the event system is included, an event round trip takes in average 10.5 ms on the same machine which is quite significantly more compared to a basic message. This increase is due to the serialization and deserialization steps of an event type into a basic message and affects the filtering mechanism involved in both JVMs. Lastly, an event round trip over wireless in ad hoc mode takes 13.7 ms.

**Scalability Benchmark**

In the scalability benchmark we analyzed the behaviour with an increasing number of nodes in a wireless infrastructure, publishing and subscribing for events. The infrastructure rep-
resents a star topology where the master sends events and the other nodes respond immediately. The system architecture remains the same in all tests but with an increasing number of responding nodes. The master laptop sends out messages and events with a delay of 50 ms and 500 ms. Beside the average round trip time the package loss is included in Figure 5.3. We also compared the access point (AP) infrastructure mode with the ad hoc (AH) infrastructure-less mode.

Figure 5.3 shows that with an increasing number of nodes, the round trip time increases. In our star topology, a client node has to respond with a reply message which with an increase in the number of clients results in a corresponding increase of reply messages. The response messages are queued until the measuring thread is able to process them and stop the clock. When comparing the wireless infrastructure mode it can be seen that the round trip time in ad hoc mode is in average 14% smaller than in the access point mode. The reason is that in access point mode a message/event is sent over an access point thus adding to the delay. The drawback on the other hand is the message loss, which increases up to 8% in ad hoc compared to 2% in access point mode. From this 9-nodes scalability benchmark we observed a strong increase of round trip time when decreasing the time interval of sending messages. Therefore, the system will currently not scale for environments with a lot of nodes sending messages with a small time interval. Switching the wireless ad hoc infrastructure mode to an access point improves the package loss for the 9 nodes infrastructure by 75%. A combination of access point mode and larger time intervals may scale better for more nodes.

5.2.4 OSGi/Nanning Benchmark

In the following benchmark we measured the overhead of the OSGi/Nanning container. Only a two-node benchmark was performed and with a different setup than in the previous
benchmark. The goal is to show the overhead of using the Nanning approach in a real setup.

In a first measurement, two layers of calling methods through proxies (Figure 5.4 - 1) were performed. On a second benchmark we looked at the overhead of the full system when sending events between two nodes (Figure 5.4 - 2). The test setup consisted of a laptop with a Java Runtime Environment (JRE) and a PDA with a J2ME/CDC virtual machine.

The laptop we used was an IBM Thinkpad A31 with 1.8 GHz running under Fedora Core 1. The PDA was an iPAQ H3970, 400 MHz with familiar Linux 0.7.1. As JRE we used version 1.4.1 on the laptop and on the iPAQ, J2ME/CDC 1.3 from Sun.

**Framework Benchmark**

In this benchmark we measured the proxy overhead of the Jadabs architecture with the event and messaging systems (Figure 5.4 - 1). We also measured the framework by publishing an event and stopping the timer before the message was sent through the UDP connection. This way we could measure the overhead of going through two proxies.

The measurement was repeated 10'000 times with a standard deviation of less than 1.5%. The numbers in Table 5.1 show an overhead of 6.5 % for the JRE/Laptop and 7.3% for the CVM/iPAQ.

**End-to-End Benchmark**

In the end-to-end benchmark we measured the round trip time of an event from a laptop to an iPAQ. Figure 5.4 - 2 shows the steps involved. An event is published by the Benchmark service. The event is sent through a wireless UDP connection in ad hoc mode to the iPAQ.
Once received on the iPAQ, the event is extracted from the message and the subscribed benchmark service is notified. Upon notification the event is published again and sent back to the original publisher where the timer is stopped. The results show the average round trip time for 10,000 published events (Table 5.2).

<table>
<thead>
<tr>
<th>VM/device</th>
<th>normal</th>
<th>proxy</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>JRE/lap</td>
<td>0.232 ms</td>
<td>0.247 ms</td>
<td>6.5%</td>
</tr>
<tr>
<td>J2ME/ipaq</td>
<td>15.8 ms</td>
<td>17.0 ms</td>
<td>7.3%</td>
</tr>
</tbody>
</table>

Table 5.2: Proxy overhead for real scenario

The measurement shows an overhead of 2.5% when using the proxy approach, as opposed to the static compiled benchmark (normal). With this, we can conclude that the observed overhead introduced by OSGi/Nanning is negligible compared with the design advantages it provides. Furthermore, when those extensions involve messages over a network, the overhead of OSGi/Nanning is completely hidden behind the cost of the functionality added to the application.

### 5.2.5 Contribution

The event system example shows the novelty of enabling a running application with the capabilities to send events and react to events at a later time. The application is adapted to
enable it with new remote capabilities. The adaptive service is recognized by the service manager and downloaded automatically into the dynamic Jadabs container. The service contains the event system and an aspect which weaves into the running application. The aspect has been implemented once using Nanning and second using PROSE. Compared to the Nanning and PROSE projects, Jadabs adds the possibility to recognize new services in its mobile environment and to download them. The new service can be kept as long as the service manager decides to do so. For example when the connection is lost which is common in a mobile environment, the code does not get un-weaved. In case the new service is not needed anymore it can be removed. With PROSE, which support remote weaving with RMI, un-weaving is an open issue. In the OSGi/Nanning container we were able to completely remove an added service and an aspect from the container. Compared to GAIA [140] which uses dynamic TAO [141] for adaptation, Jadabs provides a peer-to-peer infrastructure which is capable of adapting devices in complete decentralized infrastructures.

5.3 Autonomous Cooperating Robots

In this application example [53] we focus on cooperative, autonomous entities such as mobile robots. This is a problem often tackled in robotics [183] but has not yet been approached as an adaptive middleware problem. The idea is that each robot should have its own basic application code while everything else that is needed for cooperation, adaptation, and interaction with other robots or unexpected entities is acquired at run-time as need dictates. A possible scenario where such adaptation capabilities could be very useful is, e.g., an automated highway systems [44, 71, 96] (Figure 5.5). In such environments, cars have a built in microprocessors and are able to communicate with other cars. The idea is to use a middleware platform so that the software in each car is not set when the car is built but can be adapted dynamically. Possible adaptations include:

- On entering a new country, the car may acquire new software modules that monitor certain parameters like speed or CO$_2$ emission levels. That way the car software can automatically be adapted to the local regulations.

- Under heavy traffic conditions, the highway control infrastructure might change the behaviour of the cars so that they drive as a coordinated platoon where each car follows the next one at a fixed distance and speed [76].

- Upon facing an unexpected obstacle (an accident, a closed lane in the highway) a car may provide other cars with software extensions that allow them to coordinate their motions to successfully circumvent the obstacle.

This scenario shows how Jadabs can be used to dynamically construct an application running on mobile robots. The hardware in the robots is similar to that used in many mobile and pervasive computing settings (from car to sensor networks). Over this hardware, the software used by the robot to function and communicate with other robots, is dynamically
5.3. Autonomous Cooperating Robots

Figure 5.5: Traffic scenario

assembled in a first step. Then an inference engine is added which uses rules to control the robot's movements. Using this inference engine a series of collaboration protocols among the robots are implemented that allows them to perform coordinated tasks like drive around an obstacle.

5.3.1 Architecture

Figure 5.6 shows the different components of the adaptive platform. It contains three layers: a sensor/activator layer, an autonomous layer, and the federated cortex layer.

The sensor and activator layer might be physically separated from other layers and is responsible for data acquisition. Sensors can vary from specialized ones, e.g., for cars, to sensor nodes like MicaZ [36] or B'Tnodes [69]. We looked at a sensor and activator network in the range of several meters like in a car or a robot. In case of sensor nodes the sensor network provides its own distributed infrastructure for communication between each other. The sensors or activators can communicate over ZigBee, Bluetooth, or a serial connection. With the sensors a very limited set of functionality is available. In case of TinyDB [97] queries can be sent out to get notified about changes in the environment. To connect the sensor network to the higher layers a gateway is used, e.g., the Stargate [37]. Through Stargate, the sensor network can then be controlled and queried.

The autonomous layer is responsible for an autonomous behaviour of the single systems and runs on a single board computer like the Stargate. This layer is also responsible for the controlling of activators like in a robot scenario where motors are used for movements.
The autonomous layer contains additional wireless communication (IEEE 802.11 b) which allows it to update the software or let the entities communicate with each other. This layer is responsible for keeping the system autonomous. Additional functionality can be added when required.

Finally, the federated cortex layer is an extension to the autonomous layer. It is responsible for the autonomous federated behaviour. Cortex stands for the logical behaviour and solution finding of problems. Once the autonomous layer discovers other nodes and a scenario requiring a solution finding problem, the cortex layer can be added. The cortex, which consists of an inference engine and the application logic, is then transferred to the participating nodes. Furthermore, the running system can be extended by additional application logic.

**Autonomous Layer**

The autonomous layer contains an additional Autonomous Control module. This module allows the robot to make autonomous decisions. For example the standard behaviour of a car is to advise the driver on which route to take according to GPS information. Depending on the traffic, the destination and personal preferences, routes differ from driver to driver. The autonomous control module is therefore specific to the platform used. The autonomous layer is responsible for the loading of the federated cortex layer whenever it is required.

**Federated Cortex Layer**

The cortex's main task is to coordinate the goals or tasks of different autonomous nodes and is comparable to TEAMWORK [41]. Since each autonomous node is responsible for its own goals, it may interfere with other autonomous nodes in performing its goals. Furthermore, a particular goal may be better handled by another autonomous system to get a better overall

---

**Figure 5.6: Autonomous federated architecture**
performance. For example cars would be able to coordinate their speed on the highway for a better road utilization. The problem and solution space for such scenarios can be very large. Furthermore, this space is application and task specific. Therefore, we require a mechanism to find solutions for upcoming problems. In the logic programming approach, solutions can be deducted out of given rules and predicates in a simpler manner than in other programming approaches.

As we look at ad hoc distributed systems, we require an inference mechanism over a changing number of machines. It is also not reasonable to have an inference engine on a server in ad hoc environments like the traffic scenario. The inference engine is therefore advertised as a service plugin to allow other nodes to adapt to the federated cortex layer. The cortex module on a system consists of the Prolog engine - JIProlog [30]. Since JIProlog is a local inference engine, we extended the engine with a distributed deduction handling mechanism. The communication between the Prolog engines is done over the event system from Jadabs. This allows to send Prolog events to a set of nodes which are in an inference process.

To infer in a distributed ad hoc environment we distinguish two concepts. First, new problem and solution rules need to be added or removed from the running system. Second, during the deduction step an overall best solution has to be found. It is no longer a single autonomous system which makes a decision, but rather the federated system which makes the decision.

Logic-Extension Handling

The problem and solution patterns which extend the knowledge of the entities consists of a set of predicates. To integrate new rules the predicates shown in Table 5.3 have to be supported.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start/1</td>
<td>returns a set of clauses that in turn register the features and extend other predicates</td>
</tr>
<tr>
<td>remove/1</td>
<td>is called with the extension name as argument when the extension is unloaded</td>
</tr>
<tr>
<td>resolution/1</td>
<td>takes the deduced tasks and transforms them into concrete actions</td>
</tr>
</tbody>
</table>

Table 5.3: Extension Predicates

A new logic-extension can either be requested by a single node over the event system or it can be uploaded from another node. Added or removed logic-extensions will affect the solution finding process on the next inference step.
Inference Mechanism

Whenever a node processes a new goal, it starts a new deduction group, as in Figure 5.7. A new goal could be for example to drive from A to B. To form a deduction group, the initiating node becomes the master and sends a request for joining the group to the other available nodes. A node may join the group when it is not busy with another task and is listening for join events. Nodes, outside the communication range, do not get informed and can therefore not join the group. The joining nodes become the slaves. There could be situations where the same goal, e.g., two robots are on collision course, is initiated by two nodes. In this case, the node with the lower IP number becomes the master.

Until all members of the deduction group have acknowledged the list of members to the master, the group can be enlarged or merged with another group in case such a situation occurs. Once all its members of the group are in synchrony with regard to the member list of the group, the master will start requesting the goals from each member of the group. After it has collected the goals from all members it will distribute them to all participants with the request to compute solutions for all systems. The slave members send in their solutions they computed for all robots participating in the deduction group. Among all solutions sent in and the solution the master computed himself, the master will choose the best with respect to a cost function and distribute them to all members of the group. The members load the solution and acknowledge receiving the solution to the master. Finally, all participating members of the group proceed with the solution.

5.3.2 Prototype Traffic Scenario

To show the feasibility of our proposed architecture for autonomous federated systems we prototyped this traffic scenario with robots in our laboratory.

In the above traffic scenario of Figure 5.5 coordination between cars can happen in several situations. For example cars could form a platoon and drive with the same speed, or
road work can be announced and cars must be informed to change the lane, or in dense fog we would be happy to get informed about traffic jams ahead of us.

**Robot Architecture**

In our scenario we took at a circular grid (Figure 5.8) where robots can move along lines. By using a grid the robots are able to drive autonomously without interacting with a camera as done by [14]. The two concentric ellipses represent a two lane road. The robots move on a lane and are able to switch to another lane on the next crossing for example to overtake another robot or circumvent an obstacle on the lane.

The robots were built with Lego Mindstorms [93] and the architecture is comparable to the proposed car architecture (Figure 5.9). Instead of distance sensors and speed gauges we have two sensors to detect the line. As activators we use the motors to move the robot between the lines. The RCX is a microcontroller which collects the data and allows to set the speed for the motors. On top of the robot sits an iPAQ comparable to the Stargate. The iPAQ contains Jadabs, which in our scenario is the *Autonomous Controller*. It is responsible for the next movement like straight ahead, a left or right turn. As the autonomous layer runs on a more enhanced device like the iPAQ, it is possible to control several activators and coordinate them. In our robot scenario only simple tasks can be sent to the RCX which then turns on and off the motors. As microcontrollers are limited in their functionality, they can only perform a special set of movement tasks. The combination of sequence of such tasks has to be computed on another device. The same constraints holds also for the RCX with only 32 KByte of memory for the operating system and program. A macro engine has therefore been created [175] to execute macros which are a set of different movements. The macros are computed on the enhanced device and transmitted to the RCX which processes the movement tasks one by one.

When a robot is set on the ellipse it starts moving along the lines and sends out an event about its position. Once a robot detects another robot on the position it has planned to move
to, both robots wait. We can now upload the cortex layer including JIProlog to the robots. If there is already a robot with the logical layer they exchange the required federated cortex bundles automatically.

**Robot Conflict Management**

To show the feasibility of the federated cortex layer, we implemented a conflict management as logic extensions by using the inference mechanism. For our robot scenario we concentrated on a mechanism to let robots move around the ellipse grid without stopping on a conflict. Once a robot detects a conflict, for example another robot moving on the same line towards him (Figure 5.8), the inference mechanism has to find a solution.

With only the cortex layer activated the robots cannot find a solution. The cortex layer needs to be fed our grid specific solution handling. We can now upload such rules to the robots. By using the logic-extension mechanism provided by the cortex layer, the rules are added to the solution handling. Several such rules have been implemented, e.g., *on a conflict do a turn around*, or *do change to the next lane*, or *wait until the other robot moves and form a platoon*.

To simulate an obstacle a controller interface allows to occupy a position of the ellipse grid, marked as (!) in Figure 5.10.

Shown are four phases of the implemented conflict management for our scenario. 1) Robot D2 has recognized the obstacle and as the system includes a *change lane* extension the robot does a left turn and circumvents the obstacle. 2) On the next movement of R2/D2 a conflict occurs and the inference engine starts a deduction group to solve the problem. The produced solution requires R2 to move to the inner lane and let D2 pass 3). As the robots' goal is to move to their initial position, they move forward and R2 has to change the lane.
5.3. Autonomous Cooperating Robots

Figure 5.10: Robot conflict management

Due to the obstacle 4). Adding a new solution pattern like a *turn around* to R2 would allow the robot to turn in place and drive back to its initial position.

5.3.3 Contribution

Goldberg et al. [61, 155] proposed a layered architecture for the coordination of mobile robots. The three levels, behavioural level, executive level and planning level can communicate between the different robots without involving the other levels. In our approach we can cover the executive level and planning level on the federated cortex level. The behavioural level which allows to communicate the sensor/activator to other robots is not supported in our architecture. Since we regard our robots as autonomous, we hide the sensor/activator layer. Whereas market-based technics are used in [42, 95] for task allocation and planning, we use a cost function to select the best suitable robot for a task. A similar market-based approach could be implemented in Prolog.

The ALLIANCE [118] architecture facilitates fault tolerant cooperative control of teams of heterogeneous mobile robots working towards a common goal. The goal of this project differs from ours that instead of a common goal our robots have their own goals but try to solve the other problems as well. We can achieve this by the federated cortex layer.

Baliga et al. [14] presented a federated control system (FCS) to provide a global behaviour, such as traffic control. This control system is mainly used in a centralized infras-
structure and cars or so called elements are monitored by external systems. In our scenario we analyze autonomous federated systems where each single element can behave in an autonomous way in a decentralized infrastructure.

The robot example shows the novelty of extending a robot on different layers. The complexity of robots with its different hardware and software layers makes it an interesting target to apply adaptation. By using a proper separation of functionality, we are able to extend the robot on three different layers: sensor/activator layer, autonomous layer, and cortex layer. No other robot project is capable of extending its architecture on these three different layers. Using runtime adaptation mechanisms to cope with the complexity in robot infrastructure has not yet been tackled by others. Due to the layered architecture adaptation can be performed in different places enabling a smooth extension of behaviour.

The robot project started with the macro engine for the RCX [175]. This allowed us to execute additional and new macros which were a set of different movements. Later on, the Jadabs container was added to enable the robots with distributed functionality. Finally, the cortex layer allows us to describe the complex robot tasks, problems, and solution handling in a logic form. To extend the logic resolution itself additional predicates were added to the highest layer of adaptation. The three adaptation layers are independent of each other, so as not to affect each other.

5.4 Messenger Scenario

The SIP-Messenger scenario shows the feasibility of using Jadabs in heterogeneous infrastructures with changing standards. Mobile devices move between different infrastructures which have their own standards and peculiarities. This requires applications on the device to adapt to other standards and protocols of the infrastructure.

A huge diversity of standards can be found for messenger protocols. Messenger applications are nowadays used to get faster response time than for emails. The involved users are online at the same time and can share information like messages, and files immediately. Nevertheless, many different protocols have been proposed and de facto standards have emerged like ICQ, Yahoo!, MSN, AIM, IRC, Jabber, SIMPLE [142].

One approach to cope with the diversity of these messenger protocols is suitable only for desktop machines. Several applications like Kopete, Everybuddy, Centericq, Laffer and so on, include in their messenger client more than one of the available messenger protocols. This leads to bulky applications with messenger implementations which are never used. For resource constraint devices, such clients are not adequate. Vital memory resources are wasted. Furthermore, new messenger protocols or even company proprietary protocols would require the installation of a new messenger application.

With the proposed architecture additional protocols can be added once they are required by the user or needed in a special environment. Figure 5.11 shows such a scenario. A group of devices having an ad hoc meeting are using the JXME based messenger protocol. Once a new device, equipped with the SIMPLE protocol, enters the ad hoc meeting group, the device has to adapt its messenger protocol to the one used in the new environment.
5.4. Messenger Scenario

Figure 5.11: Messenger scenario

5.4.1 Architecture

Figure ?? shows the SIP messenger device from Figure 5.11 which moved to the ad hoc meeting group. The MessengerUI plugin represents the user interface which is independent of the underlying messenger protocol that is used. When the messenger application is started up Jadabs resolves to the SIMPLE messenger protocol for devices in the infrastructure environment. SIMPLE stands for SIP for Instant Messaging and Presence Leveraging Extension. It is based on SIP, the Session Initiation Protocol [77, 143, 171]. SIP is an IETF signaling standard used for creating, modifying, and terminating sessions with one or more participants. It is used for example in voice over IP infrastructures to set up audio sessions for telephone calls or multimedia conferences.

Once the SIMPLE device reaches a mobile environment, the service manager is able to start the dynamic service discovery. As other devices in its mobile environment are using and also providing a JXME-IM protocol, the service manager is able to download and activate the new protocol. By using an additional bundle, the messenger bridge, also discovered in the mobile environment, as well as the SIMPLE and MessengerUI services can be crosscut and extended with the additional JXME-IM.

This scenario assumes, that the JXME-IM protocol and the instant messenger bridge are available in the mobile environment. Either of these services are provided by a node in the environment or accessible from an infrastructure node.

5.4.2 Descriptors

The architecture and all discoverable services and bundles have been provided with appropriate descriptors as proposed in the Jadabs model. For the purpose of the messenger application a set of descriptors is required. This allows us to dynamically extend the application with new functionality. When a device reaches a mobile environment where other messenger protocols are used, the service manager loads and activates appropriate protocols.
The descriptors and its corresponding components are shown in Figure ???. The platform and container description is similar to the one described in the event system scenario and uses the OSGi/Naming container on an iPAQ.

**MessengerUI**

The *MessengerUI* component is described by the following plugin descriptor. It requires by default one messenger protocol implementation. An extension-point requires therefore an *IMService* implementation (line 5). The dynamic service composition mechanism chooses the protocol which is available in the current infrastructure. For example when the messenger client is started in a SIP infrastructure, the SIMPLE messenger protocol is loaded from the infrastructure. The application code is provided by the bundle and referenced accordingly (line 14).

```xml
<OSGiServicePlugin
    uuid="jadabs:im-ui:1.0.0:opd"
    service="ch.ethz.jadabs.im.Messenger">

  <Extension-Point
    id="IMService"
    service="ch.ethz.jadabs.im.IMService"
    description="requires an implementation for IMService interface"/>

  <Extension-Point
    id="SWT"
    description="requires the SWT library"/>

</OSGiServicePlugin>
```

**SIMPLE, JXME-IM Messenger**

The SIMPLE, and JXME-IM protocol implement both the *IMService* interface. For the SIMPLE based protocol, the descriptor is shown below. The SIMPLE and JXME-IM implementations provide an extension of the *IMService* interface (line 5).

```xml
<OSGiServicePlugin
    uuid="jadabs:im-simple:1.0.0:opd"
    service="ch.ethz.jadabs.im.simple">

  <Extension
    id="IMService"
    service="ch.ethz.jadabs.im.simple.IMServiceImpl"
    description="provides implementation for IMService interface"/>

</OSGiServicePlugin>
```
Messenger Bridge

The messenger bridge plugin is required to bridge between different messenger protocols. It fulfills two tasks. First, it does the proper activation of the dynamic AOP interception points. Second, it acts as a dispatcher and forward messages going to one or the other messenger protocol.

Once the service manager recognizes an additional extension-point for the IMService interface, the manager looks for an aspect component. The aspect component allows us to dynamically add additional messenger protocols. To the outside the aspect acts as a normal IMService and provides the appropriate IMService extension (line 5). The aspect plugin itself requires a Nanning container (line 10).

The aspect component is separated from the second or further messenger protocol implementation. This improves the separation of components and allows us to reuse the same messenger protocol implementation in other architectures where no Nanning is supported. This would for example allow us to write a PROSE aspect for an OSGi/PROSE container and combine it with additional messenger protocols.

5.4.3 Contribution

This scenario shows the novelty of extending applications with additional protocols. So far, applications supporting another protocol needed to be recompiled and deployed again or include all possible protocols at once. With the proposed Jadabs model, new protocols are recognized automatically without ever specifying them as potential extendable services. Other systems like GAIA [140] or MIDAS [130] need to specify explicitly where extensions can be applied. In Jadabs the service manager is responsible for discovering new possible
extensions by sending out filter request messages once devices have joined the mobile environment. Extending applications with new protocols can then be done at run-time and transparent to the application. With proper security settings new protocols can be added, in a manner transparent to the user.

The Jadabs approach enables a small footprint architecture where only the necessary bundles are used. Additional bundles, like the support of other protocols, are only loaded and activated when needed.

5.5 Web Services

The Web service scenario has two goals. First, the use of Web services from mobile environments and, second, the integration of mobile phones into Jadabs’ dynamic service composition.

In our Web service scenario we analyse an infrastructure to query for railway timetables on mobile phones. Users are able to query from their mobile phones the Swiss Federal Railway (SBB) [147] timetable. In a first step, the users should be able to query the timetable over a GSM connection. Once the users enter a mobile environment which provides the SBB Web service over Bluetooth for free, additional code is downloaded and activated. Instead of using the expensive GSM connection, a Bluetooth connection can be used.

The infrastructure is partitioned into the mobile environment part and the Web service infrastructure part, see Figure 5.12. A user who enters the Bluetooth range of the Bluetooth-Proxy is able to discover and download the SBB Client Service.
5.5. Web Services

5.5.1 Architecture

Figure 5.13 shows the architecture for the mobile environment as well as the Web service site. For the Web service client application the monolithic container approach is used so as to enable the user to run plugins on a mobile phone. The mobile phone contains a manager plugin which consists of the JXME bundle including a Bluetooth bundle for communication.

Once a mobile phone detects a gateway in the Bluetooth's communication range, it starts a service discovery to find appropriate service plugins. The service manager sends out filter service requests which the gateway tries to match. The gateway provides a Web service proxy bundle. This allows any device in the gateways mobile environment to send Web service requests to the gateway where they are forwarded to the Web service. The proxy bundle on the gateway has one restriction, it requires Web service clients to send the Web service requests over GSM to the gateway. This functionality is provided by the JXMEForwarder service plugin which can be downloaded by the Web service clients. Therefore, the gateway matches the filter request from the mobile phone and after a port handshake the gateway sends back the JXMEForwarder service plugin description. The service can then be downloaded by the "Over The Air" (OTA) mechanism. When the service has been downloaded it is activated and changes the route where Web service requests are sent and received.

The SBB service plugin sends SOAP messages over GSM. With the new JXMEForwarder service, the SOAP message can be forwarded from the SBB service plugin to the service manager and further to the gateway all over Bluetooth. Section 4.3.1 explains in more details how the adaptation on the local loop-back is performed on mobile phones.

Figure 5.14 (1) shows the user interface to query the timetable. The response message
is again shown to the user as a list of possible connections (2).

### 5.5.2 Descriptors

In the following, the descriptors for the mobile phone plugins are explained in more details. The gateway descriptors are not shown but are comparable to the ones described in previous examples.

**Platform & Container**

**Platform** The platform descriptor is shown below and is able to run the application and the extended OSGi/Nanning container.

It runs on the Nokia 6600 with an ARM processor (line 4) and Symbian (line 5). The platform contains a network interface (line 9) which is a Bluetooth connection supporting the Java JSR-82 API [75]. Second, a GSM connection should always be available (line 13). The platform is able to run the monolithic mOSGi container (line 7).

```xml
<Platform
  id="nokia6600.wlab.ethz.ch">
  <Property name="processor" value="arm" />
  <Property name="os" value="symbian" />
  <OSGiContainer id="j2me-mosgi" />
  <Netlface
    type="bt_jsr82"
    connection="dynamic"/>
  <Netlface
    type="gsm_http"
    connection="static"
    description="for GSM/HTTP Connections"/>
</Platform>
```

**Container** The containers used for the mobile phone consists currently only of its name and version (line 2).

```xml
<OSGiContainer id="j2me-osgi" name="Core J2ME-OSGi">
  <Property name="container" value="j2me-mosgi"/>
  <Property name="container-version" value="1.0.0"/>
</OSGiContainer>
```
Service Manager

The service manager plugin descriptor consists of an extension so that it can be used for the local loop-back mechanism (line 9).

```xml
<OSGiServicePlugin
  uuid="jadabs:micromanager:1.0.0:opd"
  service="ch.ethz.jadabs.micromanager">
  <Extension
    id="SOAPTransport"
    description="transport for sending/receiving SOAP messages"/>
  <Extension
    id="MicroService"
    description="provides local-loopback"/>
  <ServiceActivatorBundle
    activator-uuid="jadabs:micromanager:1.0.0:obr"/>
</OSGiServicePlugin>
```

SBB Web Service Client

The Web Service client plugin requires a MicroService extension-point (line 9) which is provided by the manager plugin. In order to send out SOAP messages a SOAPTransport is required (line 5).
Chapter 5. Application Scenarios

JXME Forwarder

The JXME Forwarder service plugin provides the capability of forwarding a SOAP message to a gateway by using the JXME layer. The descriptor therefore provides a SOAPTransport extension (line 5). When the service is activated it has to register itself with the SBBWSClient service to be responsible for the forwarding of the SOAP message. Upon registration the previously registered service manager will be unregistered.

5.5.3 Contribution

This application scenario shows the novelty of supporting dynamic adaption for mobile phones. With Jadabs, new services can be matched offline on more resource powerful nodes for example a gateway. Once a matching service has been found the mobile phone receives
a descriptor which can be installed and activated. A previously installed service is then de-
activated. *Dynamic adaptation and offline matching* are new concepts and implementations
which are not available for small devices in other distributed service infrastructures.

The GAIA Microserver [27] provides also a middleware for mobile phones but does not
support adaptation and service composition as proposed in Jadabs.
Chapter 6

Jadabs – Primer

6.1 Introduction

This chapter explains in more detail how new services conforming to the Jadabs model can be designed, implemented, and deployed. At design time services are defined which have to be implemented for one or the other platform. The services and bundles are then implemented for the specific platform and specified by bundle and plugin descriptors.

Once the implementation together with the descriptor are available, they can be tested and deployed to a repository. Service descriptions which affect a mobile environment are then deployed to the appropriate gateways.

6.2 Building an Application

If we build an application for the mobile environment with Jadabs we need to follow different concepts which we outline below. These range from service-oriented programming defined by OSGi or aspect-oriented programming defined in Nanning and PROSE and distributed programming with JXTA/JXME. An additional section explains the descriptor generation and its deployment.

6.2.1 Service-Oriented Programming

Service-oriented programming forces developers to separate the code into manageable packages. This allows developers to program their packages independently of each other. A clearly defined interface is required to use services provided by other developers.

It is vital to properly separate interfaces from implementations. They need to be put into different bundles. For larger projects one API bundle and different implementation bundles exists. The separation is not only a good programming practice but allows later on, with the dynamic AOP approach, to replace one implementation with another, more appropriate one. If the previous implementation was packaged with the API bundle that implementation code could only be removed once the API was not used anymore as described in section 4.2.3.
Chapter 6. Jadabs – Primer

The service activator is the entry point for a service. Each service requires one implementation of the BundleActivator interface, as shown in code listing 6.1. In the start method the BundleContext is passed as argument from the service container. This allows the new service to gather further services which have been registered in the container.

### 6.2.2 Dynamic Aspect-Oriented Programming

PROSE and Nanning, the two supported aspect-oriented programming concepts work differently. Nanning requires a service to be registered with a specialized AOPContext which generates the required proxy instances, see section 4.2.1. On the other hand, using PROSE is transparent to the developer of normal bundles and plugins.

**Nanning**

In the OSGi/Nanning container usual AOP crosscuts like before, after, around, and mixins can be used. The code shown below shows two possible examples of AOP functionalities. First, the interface Foo is crosscut with a before method interceptor (line 11). To do so, the AOPContext is used to get the aspect instance of the Foo interface. Crosscuts can now be performed on that aspect instance. All methods starting with bar in Foo are crosscut (line 13). Second, an implementation for Foo is replaced with a new implementation FooImplB. Therefore, the mixin concept (line 17) is used to backup the old implementation (line 20) and replace it with the new one (line 22).

```java
public void start(BundleContext bc) throws Exception {
    ServiceReference sref = bc.getServiceReference(Foo.class.getName());
    Foo foo = (Foo)bc.getService(sref);
    AOPService aopsvc = ((AOPContext)bc).getAOPService(sref);
    ai = aopsvc.getAspectInstance();

    // do a method-crosscut
    interceptor = new BeforeMethodInterceptor();
    pointcut = P.methodName("bar.*");
    pointcut.advise(ai, interceptor);

    // do an implementation replacement
    Mixin foomixin = ai.getMixinForInterface(Foo.class);
    foomixin.setTarget(new FooImplB());
}
```
Descriptor  The descriptor for this service plugin specifies an extension-point to the OSGi/
Nanning container.

```xml
<Extension-Point
  id="OSGiContainer/container:osgi-kf"
/>
<Extension-Point
  id="OSGiContainer/container:nanning"
/>
```

The shown example can now be used on devices running a J2ME/CDC and higher VM.
The deployment is done by providing the descriptor to a gateway for a specific mobile
environment and storing it together with the bundle in the repository.

PROSE

With PROSE a crosscut can be defined as shown in the code below. Instead of gathering
an aspect instance of Foo as in Nanning, the crosscut is applied to the whole VM to find
appropriate interception points. These interception points are restricted to methods starting
with `bar` (line 10) in the Foo interface (line 3).

```java
public Crosscut withdrawing = new MethodCut()
{
  public void METHOD_ARGS(Foo target, ANY x)
  {
    System.out.println("-> Advice called before: ");
  }

  protected PointCutter pointCutter()
  {
    return (Executions.before().AND(Within.method("bar*(..)")));
  }
} 
```

Descriptor  The descriptor for this service plugin specifies an extension-point to the OSGi-
/PROSE container.

```xml
<Extension-Point
  id="OSGiContainer/container:osgi-kf"/>
<Extension-Point
  id="OSGiContainer/container:prose"/>
```

This example can now be used on more powerful devices in infrastructure based envi¬
ronment. The deployment is performed therefore in most cases only to the repository or to
specific devices in the infrastructure environment.

PROSE and Nanning use similar programming style whereas Nanning has to get a spe¬
cific service for crosscut. In PROSE, the service to be crosscut is automatically figured out
by the PROSE engine with the specified pointcut (line 10).
6.2.3 Distributed Infrastructure

Jadabs uses the JXME concept for communication. Figure 6.1 shows an example of how services are able to communicate with each other. Here, a general pipe is created and a listener registered on that pipe who will get notified of incoming messages.

6.2.4 Descriptor generation and deployment

As a fair amount of descriptors are needed in Jadabs, an automatism is required for their generation. Currently the generation of bundle descriptions (OBRs) was automated by a module added to the Maven project management development environment [11]. Once a service plugin is designed, the generation of the bundle dependency path is defined and can be described in its OBRs. The service plugin are described in their corresponding (OPD) files. They still need to be written manually as no transformation tool is available which could map the PSM into the corresponding architecture implementation.

With maven and the OBR generation module, the compilation and descriptor generation is performed in one step. Once the code is compiled and the descriptor generated, they can be deployed to the OSGi repository.

```
// compile, generate descriptor and install locally
> maven osgi:install
// deploy bundle and descriptor to repository
> maven osgi:deploy
```

When the new service plugin descriptor is published in the repository it can be installed on a Jadabs peer. The installation on a peer is performed either automatically or can be performed in a static way. In the automatic case the repository is monitored by Jadabs which allows it to announce new services to other peers. In the static case new services can be installed by using a shell command on the peer to explicitly install the new service by using the services UUID.

```
JADABS-SHELL> loadBundle jadabs:example-service:1.0.0:opd
```

6.3 Emulation and Monitoring

As the mobile environment consists of many different devices running the Jadabs architecture, the amount of service plugins and bundles used may be very large. Through the dynamic service composition, services may be composed in totally new ways which have not yet been thought of. To overcome this problem an emulation tool is required which allows us to test several scenarios. In addition to the testing, a monitoring feature was included to observe the mobile environment for the installed plugins and bundles.

Figure 6.2 shows the user interface of the emulation and monitoring tool. The peers which are online appear on the left side in a tree view. The graph view allows the user to see the connections between the peers. In an additional view new peers with different
public class GMPipeServiceTestActivator implements BundleActivator, Listener {

  GroupService groupService;
  Pipe groupPipe;

  /* implements BundleActivator.start */
  public void start(BundleContext bc) throws Exception {

    ServiceReference sref = bc.getServiceReference("ch.ethz.jadabs.jxme.services.GroupService");
    groupService = (GroupService)bc.getService(sref);

    // create Pipe
    groupPipe = groupService.createGroupPipe("gmpipeName", gmpipeID);

    // set listener
    groupService.listen(groupPipe, this);
    // send test string
    sendTestString("hallo world!");
  }

  /* implements BundleActivator.stop */
  public void stop(BundleContext bc) throws Exception {
  }

  public void sendTestString(String string) {
    Element[] elm = new Element[1];
    elm[0] = new Element("obstr", string, Message.JXTA_NAME_SPACE);
    Message msg = new Message(elm);

    try {
      groupService.send(groupPipe, msg);
    } catch (IOException ioe) {
    }
  }

  /* implements Listener.handleMessage */
  public void handleMessage(Message message, String listenerId) {
    System.out.println("got message: "+ message.toXMLString());
  }
}

Figure 6.1: PipeService communication
Chapter 6. Jadabs – Primer

configuration possibilities can be assembled. A peer configuration can then be started locally or on a remote machine to emulate a new node with different available plugins. Such a node is then able to influence other nodes in its environment and vice versa.

6.4 Digital Office Scenario

Key features of the proposed Jadabs platform were used in different single applications as presented in the previous chapter. The examples show the novelty of using Jadabs in such application scenarios for one or the other Jadabs functionality. In this digital office scenario we would like to show that Jadabs can be integrated into a comprehensive application scenario for mobile and infrastructure environments. A basic infrastructure scenario for the digital office has been developed, but still some parts need to be integrated. Part of it was performed in collaboration with Jianbo Xue who is about to build up a Personal Area Network or the so called PANAMA infrastructure.

In a digital office scenario the challenge is to integrate multiple devices and applications into one common platform. Application interactions are continuously changing depending on the users' daily businesses. Many different micro application scenarios can be applied in a complete digital office. In our scenario we concentrated on a location based instant messaging application.
For example a user who needs to be reachable anytime may choose a combination of different technologies. Nowadays SMS, email, and instant messenger are used. Each of these technologies is regularly used with an implicit context. An SMS is for example sent when we think the colleague is currently not available by email. An email may be sent when we assume the user is working on his desktop. All these examples are implicit assumptions because we do not really know the user’s current behaviour and location. In our digital office scenario, we focused on enabling instant messaging in mobile and infrastructure environments. A user is moving between these different environments and should be able to use the current instant messaging technology. Depending on the users device and application utilization, an instant message should be sent as an email, SMS, or to the instant messenger application.

When looking at this instant messenger scenario several infrastructure and user properties can be conducted:

- A user may work on different devices throughout the day. On the way to work he may use his PDA or mobile phone, at work use the desktop machine and back home relax on the sofa and surf via a Media Center. All the time he is connected in one or the other way.

- On these devices a user has different communication applications running. For example on the PDA and desktop machine, an email client can be used. A messenger application is installed on the Media Center and one with a proprietary protocol on the PDA, and finally the user is reachable over SMS on the mobile phone.

- A presence infrastructure is required which allows the user to register his or her current location or a representative. A representative is used when the user is not available at all or the user’s current device is not compatible to the presence infrastructure. For example a representative could be the email server which stores the instant messages until the user is online again.

SIP was chosen as a presence infrastructure as it already provides a large deployment for telephony and instant messaging. By using SIP we are able to locate the user or a representative who is responsible for the message. Beside SIP, as the infrastructure backbone, a mobile infrastructure is required which allows to route messages in non IP based infrastructures. It is also responsible for presence mechanisms and is called the Personal Area Network (PAN). Figure 6.3 shows the overall picture of the infrastructure. To simplify the digital office scenario we only look at the architecture from a sender using an instant messenger application. The user A sends on its instant messenger application an instant message to user B without knowing B’s location or used application. Similar adaptations to the ones explained below, would need to be performed in the opposite direction.

To enable the digital office scenario several parts of the already established infrastructure had to be extended. By using the adaptive Jadabs platform, the infrastructure as well as the mobile environment can be easily adapted for the new requirements. This requires the infrastructure nodes and the mobile devices to run their application on the Jadabs platform.
6.4.1 SIP - Infrastructure

The *SIP-Presence Server* in Figure 6.3 acts as a SIP presence service. This enables the users to register their current location on the server so as to be discoverable by other users. User A, or the PAN of A, registers a representative at the SIP server. In general the representative is the application itself. In our scenario a SIMPLE instant messenger of A would be registered with its SIP address and current IP address. This allows the other PAN-B to lookup the location of A and send a message.

For the digital office scenario we adapted the behaviour of the SIP server in a way which allows us to add additional messaging mechanisms like email and SMS. When the user switches devices and applications, the *SIP-Location Server* has to be notified to use another appropriate messaging technology. For example when a user switches from his desktop machine to the mobile phone a location update is sent to the SIP-Location to deregister the SMTP server and register the SMS-Gateway instead which is now responsible for the forwarding of messages to the mobile phone.

The SIP presence server and SIP location server are separated processes. The presence server is only used for signaling purposes and the location server for storing the location of the user. On a user lookup the SIP presence server queries the SIP location server to get
the IP address of the requested user SIP address. For the digital office scenario the lookup mechanism to the SIP location server now has to be adapted to integrate location based messenger technologies. In the digital office each user can specify the precedence of the used messaging technologies. By default, messages are always sent to the instant messenger. In case the user is not available the message can be forwarded to the email address or as an SMS. The adaptation on the SIP presence server therefore requires to change the lookup of the users IP to query a user preferences database and get a representative for a user. The message is then sent to this representative which is an SMTP server, SMS gateway, or a PAN representative. As the SIP presence server is running on the OSGi/PROSE container, an aspect can be deployed which adapts the Java NIST-SIP [103] server at run-time without intervening ongoing signaling requests.

The registration of the users is performed in three ways. First, the instant messenger registers itself on the presence server. Second, the mail client requires an additional plugin which is in charge of acting as a SIP user agent and registers the email application. Third, there is a gateway which is responsible for one or several PANs.

### 6.4.2 Personal Area Network (PAN)

The personal area network is a virtual concept used to connect devices in the users range. These devices can be interconnected by different network technologies like Bluetooth, IEEE 802.11 a/b/g, or normal LAN. Several properties can be conducted for PANs:

- A user is using different devices which he may carry partially with him like a mobile phone or a handheld. Some devices which he is using are also fixed and may run all the time or be switched off.

- A PAN may consist of devices which are separated over the infrastructure network and not in the users range. Such a PAN is then forming a virtual group of the users devices.

- In a PAN an application can be separated over different machines. For example a gateway may act as a proxy and provide functionality to the mobile phone which it would not be able to use otherwise.

- A PAN consists of a master device which is responsible for the coordination of devices and applications in the PAN. Usually the most powerful node will be the master; for example a desktop machine. Its is also possible that a mobile phone is the only device of a PAN when all the others are switched off. In this case the phone is the master. This master concept is related to the mobileIP [122] infrastructure where a home agent forwards IP requests to the mobile nodes located with the users.

By using the JXTA/JXME infrastructure the PAN's infrastructure can be implemented using JXTA's peer group concept. As peer groups are distributed on the overlay network
the above mentioned properties can be accomplished. The PAN's master is in charge for detecting changes which correspond to the PAN devices.

To enable the PAN with the proposed digital office scenario three possible different adaptive functionalities are provided.

1. The masters which have registered for the SIP service infrastructure themselves changes are now provided with additional services. In case the masters match a new service it is downloaded and activated. Additionally, the service descriptions are made available to the devices in the PAN. This allows the PAN devices to react to changes in the SIP infrastructure. For the digital office scenario the masters are provided with the capability of bridging JXME instant messages to SIMPLE messages which are downloaded and activated on the OSGi/Nanning container. Furthermore, an instant messenger (IM) service plugin is provided to the mobile phones which allows them to receive instant messages over the JXME infrastructure. Therefore, the JXME-IM to SIMPLE is required on the master to adapt the messages. With the mOSGi/CLDC container on the mobile phone the IM service plugin can be downloaded and activated.

2. To adapt the email client on the Laptop in PAN-A a .NET service bundle has to be activated. The master gives the service description to the Laptop which in turn downloads the appropriate .NET bundle. With the service bundle the email client updates the email application on the location server.

3. The last adaptation enables the master to detect when users leave a room with their mobile phone. First, the sensor needs to be adapted to be able to react to this new sensor detection schema. The sensor then sends the information to the master. Second, the master needs to be adapted so that it can react to new events from the sensor and do the proper registration on the location server.

6.4.3 Discussion

The Jadabs platform was mainly built for mobile environments and to be integrated into existing infrastructures. To cope with changing requirements in the mobile environment as well as in the infrastructure environment a new adaptation platform is required. With Jadabs we are able to adapt both in mobile environments and infrastructure based devices accordingly.

As presented in the digital office scenario a new application requirement involves not only changes in the mobile environment but also changes in the infrastructure based environment. With Jadabs, changes can be easily propagated through the JXTA/JXME network. The infrastructure nodes are addressed directly so as to adapt to the new behaviour. PANs are registered in an update JXTA/JXME peer group which allows them to notify all PANs so that they can match their extension-points. The required service plugins and bundles are then resolved automatically.

Jadabs' small footprint requirements enables new application scenarios in infrastructure based environments. Further explorations have to be examined.
Part of the proposed digital office scenario was already implemented by others. For example, many email providers allow users to forward emails to SMS. In our scenario, we are able to extend the instant messaging application to a transparent messaging system where the user gets the messages in his currently used application independent of his location. Furthermore, by using Jadabs as the connecting middleware platform, adaptation can be performed smoothly and transparently to the user’s applications.
Seite Leer / Blank leaf
Chapter 7

Related Work

Our work was influenced by research carried out in many different fields of Computer Science. In this chapter we present previous and related work in component models and platforms, mobile infrastructures, and dynamic AOP solutions.

7.1 Component Models and Platforms

The proposed pervasive middleware architecture uses concepts from different component models and platforms which have evolved over time. Interprocess communication (IPC) were the first procedural interactions over process boundaries. In 1984 Birrel and Nelson introduced the concept of remote procedure calls [17]. This allows to use stubs at the ends of the local callee and the remote caller. The distributed computing environment (DCE), from the Open Software Foundation, is the most prominent service implementing remote procedure calls across heterogeneous platforms.

After the RPC idea, the object invocation mechanism was introduced using interface descriptions or an IDL. Basically two standards have evolved by OMG which is used as the CORBA IDL and by Microsoft with the COM IDL. There is no IDL for Java or CLR where the information is kept as metadata.

Short after component-based development (CBD) [13] and then Web service [7, 15] were proposed for enterprise information systems. CBD platforms and technologies, such as CORBA Components, Sun’s Enterprise Java Beans (EJB), and Microsoft’s COM+/ .NET, have become de facto standards for systems development.

7.1.1 Enterprise Platforms

Four different platforms are introduced which are used for enterprise applications: J2EE, Spring, Metro, and Web services. Whereas J2EE is a leading enterprise platform, Spring and Metro are targeting a lightweight architecture. Lightweight in the sense that at runtime only the required components are available. This does not mean the architecture can also
be used in small resource constraint devices. Nevertheless, all four platforms are using concepts which have been used again in the proposed Jadabs architecture.

**J2EE** A J2EE platform [154] has become a de facto standard for Java enterprise component platforms proposed by Sun Microsystems. Enterprise Java Beans (EJB) [40] are server-side components that execute a specific business logic. Components built in the EJB specification are interprocess components that stay in multiple address spaces as distributed objects. These components are used as transactional business objects that are accessed as remote objects. EJB is therefore a framework used to write distributed programs. The components implement a set of Java interfaces from the EJB API which allows them to run the components on any EJB compatible container. This container provides services for the EJB components like transactions, security, and persistency. An EJB container then configures services to match the needs of contained beans. These needs are expressed declaratively in an EJB’s deployment descriptor.

An EJB deployment descriptor is responsible for hosting EJB beans. There are four kinds of EJB beans – stateless session, stateful session, entity, and message-driven beans. Session and entity beans share the design of the EJB object and the EJB home interface. The home interface is used to create an instance of a bean in the container and returns an EJB object. The EJB container is responsible for the lifecycle of the hosted beans.

Deployed beans are contextually composed with services provided through the container. Contextual composition works by placing a hull around bean instances and intercepting communication from and to that instance. The hull provides proxies which wraps the instances. The same proxy mechanism is used in the Jadabs segmented container. When a service is registered in Jadabs, a proxy is generated first and the proxy gets registered in OSGi instead of the real instance of the service. This allows to intercept the service at a later time with new functional behaviour.

**Spring** The Spring framework [156] Enterprise Services Architecture tries to solve a major drawback when using J2EE. In J2EE each bean needs to be implemented with the EJB API and must be provided with a set of not less than five descriptors and classes. Spring is less invasive than J2EE as it only requires the container classes to conform to the JavaBeans naming convention. JavaBeans have been proposed by Sun Microsystems as a means to compose GUI components. JavaBeans are connection-oriented to facilitate the link between event source and event sinks. Compared to EJBs, JavaBeans do not support remote invocation. Spring uses the JavaBeans to connect Plain Old Java Objects (POJO) together. This allows it to develop and use libraries independent of a container API as proposed by J2EE with the EJB API. Spring requires deployment descriptors to wire the different JavaBeans together. To extend JavaBeans with services like persistency, security, or transaction, the beans are combined with such third-party libraries. Additional descriptors define the interception points of the JavaBeans which then extend the beans with AOP functionality. The Spring framework implements the AOP interception mechanism by using Java’s dynamic proxy approach.
The mechanism of extending the application with additional middleware libraries has also been used in Jadabs. In Spring as well as in Jadabs the premise is to instantiate only those libraries which are required. Whereas in the J2EE approach, the EJB services are available at anytime.

**Metro** Finally, Metro [99] propagates a very stable container architecture which is able to load and unload components or so-called artifacts. Artifacts can be any resource like JARs, or images which are available from a centralized repository. Metro uses a subproject Transit to load artifacts and its dependencies in different class-loaders. So that artifacts can be unloaded when they are not needed. A dedicated class-loading mechanism is an equally important key issue for Jadabs. Therefore, Jadabs uses a service composition framework on top of OSGi to load its bundles in different class-loaders. Jadabs uses a peer-to-peer repository infrastructure which allows it to extend other nodes with resources available in the peer-to-peer infrastructure.

**Web Services** Alonso et al. have stated in the book about Web services [7]:

> Web services are likely to play a crucial role both in the short and in the long term. [...] In the long term, Web service could become the basis for a seamless and almost completely automated infrastructure for electronic commerce and wide area, cross-enterprise application integration.

A key benefit of Web services is its interoperability across clients and servers. The standard protocols define how different platforms are able to interact with each other. With the emergence of XML and SOAP, standardization has evolved the remote object invocation. Web services consist of standards like XML, SOAP, WSDL, UDDI and advanced the object invocation mechanism for heterogeneous infrastructure based environments. Different boards W3C [180], OASIS [104], WS-I [181], and Liberty Alliance [94] are working on standards for the Web service stack which consists of a set of different protocols relying on each other. The WS-I [181] standardization body defines profiles of sets of Web service standards. On top of this basic interoperability protocol stack, new languages and specifications defining the composition of services to form business processes have emerged, such as Business Process Execution Language for Web Services (BPEL4WS) [23] and Web Service Choreography Interface (WSCI) [182].

To describe services in a uniform way, Web Service Description Language (WSDL) describes the services in an XML document. The service is then published through the Universal Description, Discovery and Integration (UDDI) mechanism. The Web-based distributed directory enables businesses to list their services on the Internet and discover other business services. In many cases only the WSDL document is published on a Web server and the URL to the WSDL is known to the client using the service. Finally, SOAP defines standards for XML messaging and the mapping of data types so that applications adhering to the standards can communicate with each other.
The communication interoperability advantage of the Web services concept has also been taken over for Jadabs. In Jadabs the JXTA messaging protocol is used to send messages between different platforms. On top of the JXTA messaging protocol Web services invocations can be used as shown in a previous example (section 5.5).

### 7.1.2 Distributed & P2P Infrastructures

Whereas enterprise information systems have emerged to a large community with standardization bodies, the technology and platform diversity in mobile environments is apparently changing too fast for standardization. A few de facto standards have crystallised and are now presented in the context of the proposed adaptive pervasive middleware architecture.

**Jini** [12] is a distributed service platform defined by Sun. It is based on Java and enables a platform to dynamically load code from the network. A service requestor is able to look up services in a centralized registry. The registry is known in advance or uses broadcast to announce its presence. The following is a summary of Jini’s service-oriented characteristics:

- **Service description.** A service is described with its interface and a set of attributes.

- **Service publication.** Services are published through the register method. This puts the service together with the service description in the service registry which grants a lease for the registration. Service leases expire automatically when they are not renewed. The service provider includes a reference to the service object which is copied into the registry.

- **Service discovery.** Service discovery is done through a method named lookup. The service user sends a filter to the service registry which evaluates the filter against the available service attributes. On a match the service object is returned to the service user.

Several ideas were taken over for Jadabs and were adapted to meet the mobile environment requirements.

The service lookup mechanism has been extended to be able to query not only a dedicated service registry node but to query any node in its mobile environment for services. In our architecture the service discovery, publication, and registry is all performed by the service manager. The filters which are sent to remote nodes are assembled automatically depending on the activated services. Therefore, no explicit lookup for a specific service is required.

The lease mechanism was simplified to allow the use of services as long as the node providing the service is in the others communication range. Through the adaptive architecture an extended lease mechanism could be plugged in when required by a specific service.

Once a Jini process restarts, all required remote services have to be downloaded again. In our solution the remote service code can be made persistent and allows a faster restart of the application.
A major drawback of Jini is its tight coupling with Java and RMI. This complicates the use of Jini services for other platforms than Java. Jadabs communication is based on JXTA's message protocol which is XML based.

**JXTA** A P2P network was proposed by Sun Microsystems as JXTA [24, 81]. JXTA is a P2P interoperability framework which incorporates a number of protocols. The P2P framework is specified through protocols which allows to implement them in different systems, e.g., C, Perl, Python, Java. This makes the JXTA protocols a versatile framework used in heterogeneous environments.

In JXTA each peer is represented on a virtual network so that other peers do not have to know about the underlying technologies and standards of the peers they communicate with (Figure 4.6). Each of the virtual peers is mapped to a physical node and all peers have to deal with the JXTA protocols rather than with different protocols on different peers. In this way, the physical peers are not actually connected directly but through the virtual peer that represents them.

The protocols can be separated into the following categories.

**Core.** Peer Endpoint Protocol, Rendezvous Protocol, and Peer Resolver Protocol define in which way peers connect to the network and find each other.

**Services.** Pipe Binding Protocol, Peer Information Protocol, and Peer Discovery Protocol; provide special services for the communication over a pipe or channel, the query of other peers, and the locating of them.

A main disadvantage considering the service mechanism is JXTA's lack of a proper container model. Once new services are found by their advertisements, a download of service code has not been specified. In our proposed solution we take advantage of the small footprint implementation of JXME and use it in our adaptive middleware architecture.

**P-Grid** The P-Grid [1, 5] was introduced as a structured peer-to-peer network for unstructured networks, which are decentralized and support internal networks. P-Grid is comparable to FreeNet [32] in that both can operate without any global knowledge. Systems like Chord [39], CAN [138], and OceanStore [139] need to establish a distributed index which require some form of central coordination.

By using P-Grid instead of JXTA/JXME, Jadabs would benefit from two properties which are supported by the P-Grid: *semantical search and trust with security*. First, by using the reputation-based trust management of the P-Grid [4] the current centralized knowledge of trust in Jadabs can be extended with a decentralized trust establishing platform. For P-Grid a decentralized public key infrastructure [3] was introduced as a basis for higher-level security service. Second, semantic search following the concept of GridVine [2] was implemented on the P-Grid. The Jadabs descriptor could be described in Resource Description Framework (RDF) files and published on the GridVine. This would allow to query the whole P-Grid for matching services.
While P-Grid enables a scalable decentralized peer-to-peer system it still remains open how a service discovery mechanism can be implemented upon it. Such a service discovery mechanism would be able to use the semantical infrastructure of P-Grid but location based discovery has not yet been tackled. Location based discovery is needed to distinguish mobile environments from infrastructure environments and provide location dependent services. Furthermore, a notification mechanism would be needed which allows the nodes to be notified about leaving and joining nodes in a specific location to request a service discovery upon new joining nodes. It is also an open issue how well the P-Grid concept can be implemented for resource constraint devices like a mobile phone or a handheld.

7.2 Mobile Infrastructures

GAIA [140] is a middleware infrastructure for active spaces. The system focuses on the management of active space resources and provides location, context and event services. GAIA is based on GaiaOS [25] a component based meta-operating system, that runs on top of existing systems, such as Windows2000, WindowsCE, and Solaris. GaiaOS uses dynamic TAO [141] a CORBA compliant reflective ORB that supports runtime configuration. Applications on smaller nodes require the CORBA infrastructure based on more powerful machines. The GAIA Microserver [27] which runs on mobile phones therefore requires a proxy in the GAIA infrastructure.

Olympus [137] was introduced as a high level programming concept for active spaces with GAIA. Its main goal is to automatically choose the best execution environment for applications with respect to the current context, resources available and user preferences. Application and resources are represented as entities. Entities need to know what kind of interfaces they support and require to interact with one another. To have a common understanding of the entities they are defined using an ontology. Once a user moves between different active spaces the previously used applications and available resources in the new active space are semantically matched.

Several of the above work has lead to an implementation of an automatic configuration service for distributed component systems as presented in [87, 88, 89]. The implementation presents new experimental results for an integrated architecture which supports: (i) automatic configuration of component-based applications, (ii) intelligent and dynamic placement of applications, (iii) dynamic resource management, (iv) push and pull component distribution, and (v) dynamic reconfiguration. An automatic configuration service was developed in which different kinds of prerequisite descriptions can be utilized. For validation the Simple Prerequisite Description Format (SPDF) was proposed. The SPDF is comparable to the Jadabs model which also allows to define prerequisite by using the extension mechanism. The implementation is based on J2EE and therefore needs more powerful nodes than what are required by the Jadabs container implementations.

CORBA-EC [151, 153] was introduced as a lightweight distributed reflective component model based on CORBA. It imposes a peer-to-peer network model in which the whole
network act as a repository to manage and assign the whole set of resources: components, CPU cycles, memory, and so on. Thus, application deployment is automatically and adaptively performed at run-time. Resources are described in an IDL and properties in XML files. This allows the resource manager in the CORBA-LLC container to aggregate new components to the system. The application domains of CORBA-LLC deal with CSCW and Grid Computing. In such infrastructures it solves the stated requirements [152]: 
- lightweight, 
- heterogeneous, collaborating peers over a P2P network, scalable and fault-tolerant, 
- seamless integration of new components, automatic component dependency management, and 
- one component model.

The last requirement stated as integration of tiny devices has to our knowledge not been addressed. Even though the implementation is supposed to be lightweight, no implementation is so far available for small devices as it is tackled by Jadabs. Whereas the CORBA-LLC model has similarities to the Jadabs PSM, our container implementations allow a finer grained adaptation.

**Prism** [100] is a software architecture which provides programming language-level constructs to implement components, connectors, configurations and events. The middleware is assembled before deployment according to the required components. So far no run-time changes can be performed as are required in a dynamic reconfigurable environment.

**Midas** [130, 131] is a dynamic middleware which combines the dynamic AOP implementation PROSE [129] with the Jini infrastructure. By using Jini the infrastructure is bound to Java environments only. Additionally, it requires a centralized lookup server which is not always available in an ad hoc environment. Furthermore, PROSE requires a Java Virtual Machine supporting the JVM Debugger Interface which is not available for all Virtual Machines in small devices. In our Jadabs architecture we use the proxy concept which is available in the J2ME/CDC implementation. As our distributed infrastructure depends on XML protocols we are not restricted to Java environments.

**ReMMoC** [62, 63] is a reconfigurable reflective middleware which supports mobile application development. It uses OpenCOM [33] as its underlying component technology. OpenCOM is implemented in C++ and available for a few platforms. An upcoming new release should support more platforms based on XPCOM [101]. ReMMoC also proposes a container model for mobile devices and is based on C++. Our platform contains additional an adaptation mechanism and a dynamic service composition mechanism. Furthermore, we propose a distributed infrastructure based on JXTA/JXME protocols which is independent of a network technology or an operating system.

### 7.3 Dynamic AOP Solutions

In this section we first give an overview of dynamic AOP implementations in Java and compare them to our solution. Later on we show the differences to .NET solutions.
7.3.1 dAOP in Java

JAsCo [31, 164] is an aspect-oriented programming language originally tailored for the component-based field in particular the Java Beans component model. The language introduces two concepts: aspect beans and connectors. The aspect bean defines the behaviour and a specification of the hook. This makes the hook context independent and reusable. The connector is used to deploy one or more hooks within a specific context; the so called traps. On occurrence of such a trap the appropriate aspect bean is executed. These traps are inserted before the execution is started and therefore have to be known in advance. To allow the addition of any possible crosscut at run-time all method entries and exits and field changes need such a trap functionality.

The JAsCo community also provides a JAsCo HotSwap solution which inserts and removes traps at run-time [35]. A similar approach to insert traps has already been used in the previous version but with a pre-processor at load-time of the classes. The combination of pre-processor and HotSwap at run-time seems a promising approach for one or the other situation.

A new technique called Jutta [173] allows to cache aspect interpreter evaluations to enhance the join-point filter execution. Such evaluations have to be performed for every reached join-point to evaluate the appropriate aspect advice to call. By using a just-in-time combined aspect compilation (Jutta) the evaluation has to be performed only once to generate a combined hook behaviour code fragment by using Javassist [28]. Such an interpreter evaluation mechanism could be used inside our AOP Engine to enhance the filtering.

AspectWerkz [21] is an open source dynamic AOP project. Currently two models for aspect components are supported. The pointcuts are defined in an XML file. With the Metadata (JSR-175) in Java 1.5 the pointcuts can be defined directly in the aspect as metadata.

The weaving in AspectWerkz can be conceptionally divided into two parts: weaving of pointcuts and weaving of the aspects which are using these pointcuts. The distinction is done because the pointcuts have to be woven no later than at load-time. Once the join-points of the pointcuts are set in the code they cannot be unwoven anymore but only disabled. It is therefore also not possible to set join-points in already loaded classes. The full dynamic comes in with the aspects. As pointcuts are independent of their using aspects, aspects can be attached or detached from pointcuts at run-time.

AspectWerkz is currently working on a solution for dynamic pointcuts. Once AspectWerkz is fully dynamic it supports a reach set of application scenarios. Applications can be woven at compile time to reduce the time overhead at load-time and specially at run-time. As many aspects are known at compile-time it wise to weave them before startup. For applications which need a dynamic behaviour as propagated in a mobile environment, new behaviour can be added as required without a restart of the application.

JAC [120, 121] is a Java Aspect Component framework to build aspect-oriented distributed applications. Two levels of aspect-oriented programming are distinguished:

- The programming level is similar to AspectJ where new aspects, new pointcuts, and
new wrappers are implemented.

- The configuration level is used to customize existing aspects. By using a configuration language it is possible to reconfigure the system with predefined aspects.

The join-points used by JAC have to be established at compile-time or load-time of the class. Any dynamic changes of aspects can be done at the pointcut level of the already established join-points.

The RtJAC [49] project overcomes the problem of introducing new join-points at runtime whereas the loading mechanism of JAC needs to be changed.

PROSE [128, 129, 132] is a just-in-time weaver of aspects which allows to weave an application at run-time. Pointcuts and aspect advice are defined in an aspect class. Such an aspect can be woven at run-time which allows the PROSE engine to generate the appropriate joints from the pointcuts. The PROSE engine supports join-points on method entry, exit, replace, exception throw, field set and get.

In one version PROSE uses the JVMDI which is the lowest layer of the Java Platform Debugger Architecture (JPDA). By using the debugger interface the join-points can be set as breakpoints. When a breakpoint is reached the PROSE engine will execute the appropriate aspect advice. In a second version of PROSE join-points are inserted for every method entry, exit and field modification. The so called join-point stubs are evaluated in the PROSE engine to call an aspect advice if the respective join-point is defined. The insertion of the join-point stubs is done at JIT compilation time by using the IBM Jikes Research Virtual Machine.

In our dAOP Engine we do not have to change the CLR JIT mechanism but we trigger each JIT compilation. This allows us to insert a join-point stub on a later time. It is comparable with setting a breakpoint at the specified location as introduced in the JVMDI version of PROSE except that the breakpoint consists of a join-point stub. This approach has to be taken as the profiler of .NET is not allowed to execute managed code directly.

Woole [145] is a just-in-time aspect weaver by Sato et al.. A previous implementation [29] of Wool was only based on the the HotSwap [43, 78] mechanism introduced in the JVM 1.4. The HotSwap mechanism in the JVM allows to replace binary compatible classes (see chapter 13) at run-time which replaces a method body with a woven message body. The woven message body may therefore contain a before or after aspect advice. This mechanism also allows to replace a method body with only the aspect advice which would be analogous to the the around advice in AspectJ. This first implementation of wool had the drawback that even for a simple logging advice in the beginning of a method the whole class had to be replaced which had consequences to the run-time performance of the application at the time of weaving.

The second approach in Wool took advantage of breakpoint execution in the Java Debugger Interface. A breakpoint which represents a join-point can be set in the running application. As soon as a breakpoint occurs Wool executes the aspect advice. By having
breakpoint execution and method replacement in one system it was possible to choose the right behaviour depending on performance issues or system decisions. Currently the decision for one or the other solution is done by hand but could be automated at the system itself.

As Wool uses the highest layer of the Java Platform Debugger Interface the time overhead is significantly higher than in PROSE which uses a direct approach with JVMDI.

7.3.2 dAOP in .NET

CLAW [34, 91], Cross Language Aspect Weaver, is a run-time extension to the CLR which allows to weave aspects at JIT compile time. At JIT time dynamic proxies are generated for join-points which are defined in an external XML file with pointcuts. The dynamic proxies can then be used to dispatch to the aspect advice. In the implementation of CLAW the profiler was used. Unfortunately a re-JIT to redefine new join-point is not possible due to a buggy method in the CLR. Even though CLAW was presented as a demo at AOSD 2002 [91] the implementation unfortunately is not available. Our work is based on John Lam’s work and extends his work into an integrated solution which allows to remove inserted aspect again. Furthermore, we are able to insert traps in methods which have already been compiled to native. By setting the re-JIT flag we are able to recompile the method again on the next invocation. By an address calculation we are able to circumvent the buggy method in the CLR which would result in a re-JIT.

JAsCo.NET [174] The JAsCo community is also working on the JAsCo.NET implementation. The .NET solution provides support to specify aspects on specific Web service interactions [174]. This approach requires a new Web service component model. The JAsCo.NET Web service has built in traps, which enable run-time aspect application and removal. As not all third party Web service vendor have JAsCo.NET enabled services, regular Web services are transformed into JAsCo.NET Web services at startup.

Weave.NET [90] describes a solution to weave aspects at load-time in a language neutral way. As .NET is based on the Common Type System (CTS) any language which matches the CTS can be woven with some other language. The pointcuts are defined in separate XML file which are combined with the aspect code to weave them with the target assembly. As weaving is done on the assembly level no source code is needed for cross-language weaving. As Weave.Net weaves the aspects at load-time directly into the code without a trap mechanism, the aspects cannot be removed at run-time.

Loom.NET presents an AOP solution to weave aspect-code and component-code and uses the mechanisms of the Common Type System in .NET. Their approach is therefore also not restricted to one language. The first implementation [148] weaves aspects statically whereas in a second approach they allow dynamic weaving [149]. The pointcuts are defined as attributes in the aspect and woven at instantiation time of the target class. As long as the instances are used the aspects in that instance are executed.
Other AOP solutions are explained only shortly as they are either static or in a pre-prototype stage. CAMEO [133] is a static aspect weaver which hooks into the C# compilation phase. The aspect weaver uses the abstract graph of the Shared Source Common Language Infrastructure (SSCLI) to weave the aspect code. The Shared Source Common Language Infrastructure (SSCLI) is also known as the open source implementation of the CLI called Rotor. Eos [134, 135] is a static AOP approach similar to AspectJ but added support for first-class aspect instances and instance-level advising. AopDotNetAddIn [9] is an AspectJ like language which can be plugged into the .NET infrastructure. AOP.NET [10] describes a similar dynamic AOP approach but they propose to use the profiler to replace the relative virtual (RVA) at run-time to call a proxy method. However, no implementation could be found so far.
Seite Leer / Blank leaf
Chapter 8

Conclusion and Outlook

8.1 Conclusion

This thesis started by illustrating why a new pervasive middleware architecture is needed for mobile environments. So far, no middleware is available which is able to provide a solution for the inherent heterogeneous and resource constraint requirements in mobile environments. Jadabs, our new pervasive middleware, copes with the dynamics and adaptive behaviour of mobile environments. First, a container model was defined whereon services in a mobile environment can be described. Second, the model is mapped to different middleware architecture implementations which can be run on devices ranging from mobile phones over handhelds to gateways and laptops. A core advantage of the architecture is its ability to adapt to new requirements. Adaptation is performed on different layers of the architecture but always with the goal to adapt applications transparently at run-time. A two way approach was used to implement the Jadabs model thereby defining both a segmented and adaptive path as well as a monolithic and dynamic approach. Finally, the model can be used for dynamic service composition in mobile environments where nodes leave and join.

Jadabs-PSM  The Jadabs platform specific model (PSM) is defined as part of the model driven architecture (MDA) approach to separate the application logic from the underlying platform technologies. The proposed Jadabs-PSM then defines both the service model and the container types for a pervasive middleware architecture. The containers can be categorized into a segmented container for more resource capable platforms and a monolithic container for resource constraint devices. Furthermore, the service model is mapped to XML descriptors for metadata processing as needed for the dynamic service composition.

Jadabs middleware platform  The proposed two container types were implemented for several platforms. The containers are based on the OSGi service-oriented architecture which can be implemented in two ways, once as segmented container and once as a monolithic container. The adaptation capabilities are provided for the monolithic container approach by the service manager which dynamically loads and unloads service plugins and
allows other services to be adapted. Much more efficient are the adaptation capabilities of the container itself by using Nanning or an underlying run-time adaptation possibility like PROSE. For communication the JXTA/JXME messaging protocols is the best solution in mobile infrastructures. By using a common peer-to-peer protocol specification, the communication can also be used for mobile and infrastructure environments.

**Dynamic service composition** Dynamic service composition is an important concept to autonomously bind to new services in mobile environments. Services are described with their metadata. This allows the service manager of the container to compose services even though they have not been designed to be used in distributed environments. The autonomic behaviour of the service manager allows it to react to changes in the environment transparent to the application.

The proposed Jadabs solution is a unique platform for mobile environments and small resource constraint devices. Other mobile platforms and infrastructures like Jini, GAIA, P2P systems, or enterprise platforms only provide partial solutions for a pervasive middleware. The concept proposed and developed has in Jadabs mainly been applied to mobile environments. Nevertheless, parts of these concepts and implementations have shown to be applicable also to infrastructure environments. In the digital office scenario a hybrid mobile and infrastructure based environment needs to be supported. As is shown, projects like JXTA, OSGi, and PROSE are already proven solutions for infrastructure environments. The combination of these projects with the dynamic service composition capabilities of Jadabs would make Jadabs also applicable for different infrastructure scenarios as proposed in the outlook.

As a concluding remark, with Jadabs we provide an enduring solution in different ways. The proposed middleware platform is a contribution to the need of a middleware for pervasive computing. Device capabilities will change and there will be a shift of capabilities upward but with new even smaller devices on the low end. Mobile phones will soon have the resource capabilities of handhelds and handhelds the capabilities of laptops. On the other hand, new technologies will lead to smaller devices which may have the resource capabilities of today's mobile phones. A middleware platform is therefore needed which fills this gap. Jadabs would also provide an enduring solution for many scenarios in enterprise platforms. Apart from additional scenarios described in the outlook, there is pressing need for service composable platforms as proposed for eclipse [46], the recently proposed Java module system [162], or the Spring framework [156]. All of these solutions compose services at load-time and in a predefined way. The dynamic service manager of Jadabs adds to the load-time composing also a dynamic and transparent way to recompose at run-time services.
8.2 Outlook

With Jadabs this thesis provides a pervasive middleware architecture for several platforms. Its implementation can be extended and improved to gain in performance and functionality completeness. Conceptually, progress in three directions is favoured.

Secure service activation Security is by all means a very important aspect in mobile environments where new functionality can be loaded and activated. A first step in this direction has already been done by providing two prerequisites. First, the platform itself is able to execute code in a sandbox model disallowing the execution of un-secure code. Second, downloadable plugins and bundles are signed by a trusted repository. A centralized security model was chosen which allows to check the signature before installation and activation.

A reputation-based trust management and a public key infrastructure for decentralized peer-to-peer systems was introduced by Aberer et al. [3, 4]. Jadabs would greatly benefit from such a decentralized security mechanism so that transparent adaptation to the user would be possible.

Context and User Preferences The proposed adaptation mechanism focused on adaptation of services with available services in the devices environment. If more than one services with the same extensions are available all of this services are installed and activated. It is up to the service user or the available AOP functionality which service is used. User preferences like for example using the cheapest connection for a specific service can not yet be specified. To enable such context dependent adaptations the Jadabs model and the service manager need to be extended. Extensions and extension-points need to support context information. The service manager has then to evaluate according to the given users preferences, the best service composition. Further investigations are needed to include an ontology and context infrastructure comparable to [137] and which can be implemented for small devices.

Distributed service composition A very interesting further step would be to integrate the service composition into a distributed service composition mechanism. Currently a local decision is made about which services have to be downloaded and activated. A distributed decision making mechanism could allow to create a whole service composition chain. A required service on a device could therefore result in dynamically building up a new service composition chain over different devices. This concept could be used in the PANAMA project which is building up a personal area network. In a PAN, applications follow the user to its current using device. As not all devices are capable of running all applications, the application needs to be divided into tasks which can be run on the small device and tasks for more powerful devices. The application is then composed of tasks distributed over the PAN’s devices. This distribution would require a distributed service composition mechanism which allows it to connect different applications tasks into one distributed application.
Middleware platform for infrastructure environment  The small footprint of Jadabs and its dynamic adaptive behaviour makes it an interesting platform for infrastructure environments and enterprise applications. In the proposed digital office scenario Jadabs is used to run the SIP server infrastructure and PAN gateways. For communication between these infrastructure devices, JXTA is used. Further evaluations are needed to show the feasibility of using Jadabs for enterprise platforms. It would be interesting to use Jadabs in the following two architectures: grid computing as in [16, 51, 119, 167], and agents [177]. Thompson [168] has already stated that architectures for agents and grids are very different at first look, but we can learn from comparing them that they have implications on agents, grids, Web service, pervasive computing, and middleware. We think Jadabs could play an important role as a basic middleware for such systems. Jadabs is lightweight and provides a dynamic mechanism to which additional functionality can be added. New computational code could be easily distributed over a Jadabs-Grid. Agents could be run on the dynamic Jadabs platform where they can freely move.
Bibliography


[71] Roberto Horowitz and Pravin Varaiya. Control design of an automated highway system.


Curriculum Vitae

Education

1991 – 1996  Gymnasium in Heerbrugg, Typus C
1996 – 2001  Computer science studies, ETH
08/2001     Dipl. Informatik-Ingenieur ETH
08/2002 – 2005 PhD at Institute for Pervasive Computing ETH

Work Experience

02/1999 – 04/1999  Internship at BMPI AG
05/1999 – 07/1999  Internship at Ergon Informatik
2000/2001         Teaching assistant at ETH
10/2001 – 11/2001 Temporary job at BMPI AG
03/2002 – 08/2002  Job at BMPI AG
08/2002 – 2005     Research and teaching assistant at ETH