Location-Transparent Integration of Distributed OSGi Frameworks and Web Services

Christoph Fiehe, Anna Litvina, Ingo Lück
Oliver Dohndorf, Jens Kattwinkel and
Franz-Josef Stewing
MATERNA Information & Communications
(christoph.fiehe, anna.litvina)@materna.de

Jan Krüger and Heiko Krumm
TU Dortmund University
(krueger, krumm)@ls4.cs.uni-dortmund.de

Abstract

The OSGi Alliance defines an open, modular, and scalable service delivery platform. The DPWS specification standardizes the process of consuming and exposing Web Services in a lightweight footprint. In our work, we provide a solution for the mutual integration of OSGi and DPWS. The approach adopts the mechanisms of distributed object systems. It employs OSGi-based service proxies and service skeletons.

1. Introduction

The *service-oriented architecture (SOA)* is an emerging architectural style to achieve high interoperability of heterogeneous software components and systems. Web Services implement this principle and provide a convenient way of creating flexible service-oriented solutions.

The *Devices Profile for Web Services (DPWS)* [5] enhances this approach by standardizing the process of consuming and exposing Web Services in a lightweight footprint. DPWS targets resource-constrained devices explicitly so that it can be applied to a variety of embedded devices used widely in homes and enterprises. A key concept is the creation of a distributed SOA within heterogeneous environments. The best evidence of its importance and its future prospects is the foundation of the *OASIS Web Services Discovery and Web Services Devices Profile (WS-DD)* Technical Committee [9] in the year 2008. Furthermore, DPWS is natively integrated into Windows Vista.

OSGi runs in a *Java Virtual Machine (JVM)* and offers an intra-JVM SOA. It defines an open, modular, and scalable local service delivery platform. The specification [11] is created by the OSGi Alliance, a non-profit consortium of ICT companies and research organizations, which promotes a process to assure interoperability of applications and services based on OSGi technology. This technology will surely benefit from exceeding JVM boundaries and merging it with distributed SOA environments. In this paper, we present a solution to this problem that connects OSGi with DPWS and vice versa. In order to ensure interoperability and a wide field of application, the principles of both technologies must be preserved.

This paper is structured as follows: Sections 2 and 3 give a short overview of the OSGi and DPWS technology. Section 4 introduces the principle of distributed object systems. In Section 5 we outline the related work briefly and present in Section 6 our requirements. Section 7 describes the architecture and the design principles of our approach which are illustrated by an application example in Section 8. Finally, Section 9 concludes the paper.

2. DPWS

DPWS defines a minimal set of standards and specifications in order to provide Web Service based communication for embedded devices. It identifies a core set of Web Service specifications comprising the following areas: secure message transmission, dynamic discovery, description, subscription, and event notification.

According to the DPWS specification, a client can discover and use services which are hosted by DPWS devices. The discovery process implies sending ”Hello” and ”Bye” messages respectively, when a DPWS device joins or prepares to leave a network. A client initiates a search for particular services through ”Probe” messages. Matching services answer with corresponding ”Probe Match” messages. Data transmission within DPWS is carried out on the basis of SOAP using HTTP as well as SOAP-over-UDP. The actual usage of services is performed by means of ”Invocation” messages. In order to receive notifications from a service on some event type, a client can register its interest by sending a ”Subscribe” message. When the event occurs, the client is informed through a ”Notification” message.

Several implementations of the DPWS specification exist already. The open source *WS4D.org Java Multi Edition DPWS Stack (JMEDS)* [15], developed by TU Dortmund University and MATERNA, is characterized by its modular extensible architecture and features like interpretation/generation of service descriptions (WSDL) at runtime, a generic web-based user interface, and a small footprint.

3. OSGi

The OSGi technology provides a service-oriented standardized way of managing the software lifecycle [14]. Furthermore, the technology caters for the integration of the pre-built reusable and collaborative components, reducing maintenance costs by delivering and updating provided services dynamically. The core of the OSGi specification comprises the OSGi framework which provides an execution platform for Java-based components, called *bundles*. The platform allows to install, uninstall, start, stop, and update bundles at runtime without restarting the entire system. Attaching a *fragment bundle* to its *host bundle* provides the ability to extend the host bundle's class path with classes or additional resources. Bundles offer their functionality in the form of services by means of a publish-find-bind mechanism. For this purpose, a *service registry* is used where bundles can register their services under one or more interfaces and search for other services. Moreover, the OSGi framework contains a *service tracker* which notifies registered listeners about service registration changes. A generic mechanism to subscribe and receive events from services or the framework itself is provided through the *Event Admin Service*.

4. Distributed Object Systems

A *distributed system* is a collection of independent components which appears to the users as a single coherent system. The object-oriented model for a distributed system is based on the model supported by object-oriented programming languages. *Distributed object systems* [13] provide *remote method invocation* for the purpose of transparent object sharing. It means that the actual location of the *distributed object* is transparent to the client. A key principle is the separation between an object's implementation and its interfaces. The object encapsulates its state and offers access only through methods made available by its interfaces.

A typical distributed object architecture is presented in Figure 1. When a client binds to a distributed object, an implementation of the object's interfaces, called a *proxy* object, is created. It marshals the input parameters and sends the invocation request to the *object server*. At the object server, the request is dispatched to an *object adapter* which activates the object. This means that the object is brought into the server's address space, that its local interfaces are

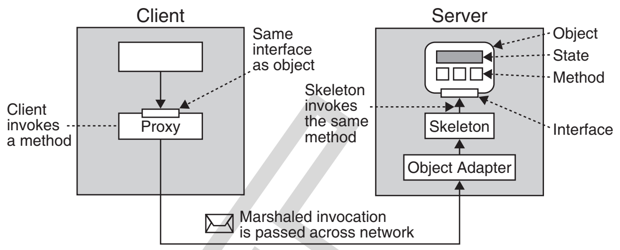

Figure 1. Distributed Object

activated, and that threads for remote method invocations are created. Subsequently, the request is forwarded to a *skeleton* object which unmarshals the input parameters, invokes the method of the object, and marshals the output parameters. Finally, the invocation response is sent to the calling proxy object which unmarshals the output parameters and returns them to the callee.

5. Related Work

The federation of distributed OSGi frameworks has been studied in only a few approaches so far. One of them is the open source project R-OSGi [12] that realizes the federation of OSGi frameworks through a middleware layer. It offers remote access to OSGi services as well as remote event notification. Another project is Nyota [1] that provides a lightweight solution to this problem based on Web Service technology. The DPWS Discovery Base Driver [2, 3] provides remote access to OSGi services through DPWS. These three approaches are outlined below.

R-OSGi [12], developed by ETH Zurich, is a middleware platform running on top of any OSGi framework. It provides support for sharing OSGi services over a network and allows a centralized OSGi application to be transparently distributed among different OSGi frameworks. The goal of transparency is achieved through dynamic proxy generation at runtime and a distributed service registry. For this purpose, R-OSGi creates proxy services on the fly which forward method calls to the corresponding remote services. For data transmission, R-OSGi uses a proprietary binary protocol over persistent TCP connections. The distributed service registry is realized by means of jSLP, a Java implementation of the *Service Location Protocol (SLP)* [7]. The service discovery mechanism of R-OSGi can be extended by other protocol implementations.

Nyota [1] exposes OSGi services through Web Service endpoints. This approach depends on the buddy class loading mechanism which is an extension of the Equinox OSGi implementation. In order to make an OSGi service remotely accessible, its registration properties must contain specific information about the transport protocol and the Web Service endpoint location. Nyota tracks the services being registered or unregistered and announces their appearance or disappearance. For data transmission, the two Web Service protocol implementations Hessian and XFire are supported. In order to use a remote service in another OSGi framework, a client has to create and register a proxy service first. For this purpose, the service's interface classes must be available in the local OSGi framework. The protocol-specific proxy generation as well as the service endpoint location are managed by an additional configuration bundle. By means of an optional service discovery mechanism the proxy generation can be performed automatically in the background without any manual intervention.

The DPWS Discovery Base Driver [3] developed in the ITEA ANSO project is comparable to the UPnP Base Driver [6] implemented in the Domoware project. Instead of UPnP, the DPWS Discovery Base Driver integrates DPWS into OSGi. This approach is described in the RFP 86 [2] which was contributed to and accepted by the OSGi Alliance. The DPWS Discovery Base Driver exposes and advertises OSGi services as DPWS services which can be discovered and used by DPWS clients without detailed knowledge of the underlying communication protocols. Local and remote DPWS devices can be found by means of OSGi standard mechanisms.

6. OSGi in a Distributed SOA Environment

R-OSGi offers a solution for a federation of distributed R-OSGi frameworks. It cannot embed OSGi services into a distributed SOA environment based on open standards. Therefore, the interoperability and the field of application is limited. Nyota is tailored to the Equinox OSGi implementation and depends on its buddy class loading mechanism. It cannot be executed on another OSGi framework which does not support this mechanism. The DPWS Discovery Base Driver exposes OSGi services as DPWS services, its goal is not to transparently integrate DPWS services in OSGi to achieve a federation of OSGi frameworks.

To merge distributed SOA environments with OSGi, remote access to OSGi services must be offered by means of vendor-neutral and broadly accepted standards. DPWS is a promising approach of bridging the gap between a distributed and a self-contained SOA environment provided by OSGi. The solution must fulfill the following key requirements which are similar to those defined in [8]:

- *Location transparency:* For a client, there must be no distinction between the usage of local and remote OSGi services. They should be accessed in the same manner as if they were present in the local OSGi framework.
- *Support of legacy services:* Existing OSGi services should not be modified necessarily in order to provide remote access. The OSGi programming model must be preserved and should not be subject to any adjustments.
- *Fault transparency:* Due to the unreliable nature of data transmission in distributed environments, the introduction of a new fault model must be avoided. Communication faults must be handled only by resorting to standard OSGi functions.
- *Dynamics:* It must be taken into consideration that the surrounding environment is continuously changing. Services appear or disappear and may be temporarily unavailable. These incidences are the rule rather than an exception and must not lead to misbehavior.
- *Manageability:* Only the information about those OSGi services which should be remotely accessible must be exposed. It has to be avoided that every OSGi service can be accessed by external clients or that every remote OSGi service is integrated in the OSGi framework.
- *Compatibility:* There are OSGi implementations which can be executed on very resource-constrained devices. These limitations must be considered so as not to restrict the applicability of the solution. Furthermore, the solution must not be tailored to a specific OSGi implementation. Therefore, it can only be resorted to OSGi standard services.

7. OSGi-DPWS Integration

OSGi provides a component model, but does not directly address distributed systems. By means of DPWS it is extended by a set of bundles and services in order to realize transparent cross-platform access to OSGi services. The solution relies on existing OSGi and DPWS security mechanisms. An architectural overview of our approach is given in Figure 2. Dashed arrows represent package dependencies, whereas solid arrows signify usage relationships. We adapt the principle of distributed object systems, described in Section 4. According to this principle, we distinguish between two roles: client and server. On the right, the figure shows an OSGi framework acting as a server and intending to offer remote access to the OSGi service of bundle *A*. Bundle *B*, installed in the left-side OSGi framework, intends to use this service. On server-side, the *skeleton generator* bundle finds the service (1) and checks whether appropriate marshaling services are available (2). It creates the corresponding DPWS device *A* which hosts the DPWS skeleton service *A* (3). By means of JMEDS, wrapped as a bundle, the availability of the DPWS device is announced through a "Hello" message (4). This message is received by the JMEDS bundle in the client OSGi framework which informs the *proxy generator* bundle. If appropriate marshaling services are available (5), it creates a package bundle which contains the required interfaces (6). Furthermore, the DPWS proxy service *A* and the proxy *A* bundle with a corresponding OSGi proxy service are generated (7). Afterwards, bundle *B* can use the OSGi service offered by bundle *A* (8).

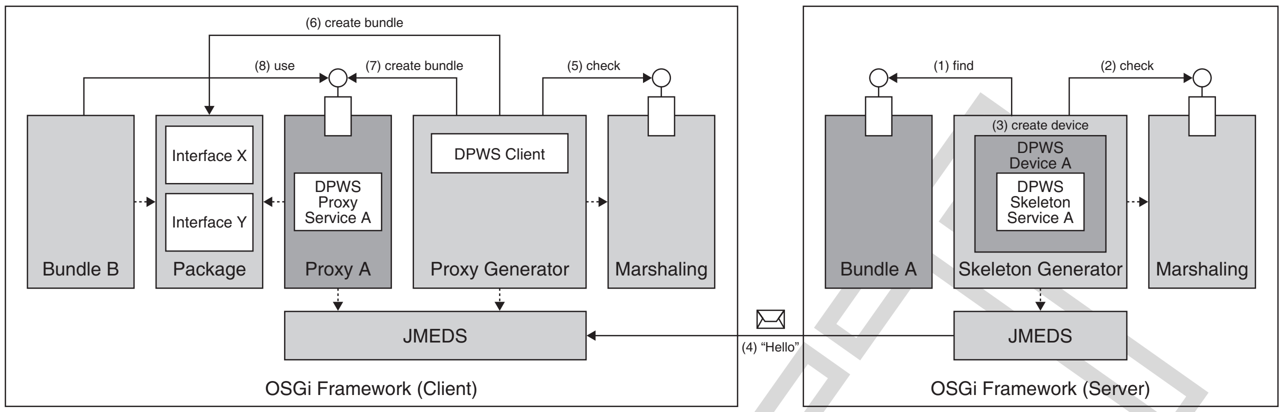

Figure 2. Skeleton and Proxy Generation

7.1. Skeleton

Skeleton objects provide the connection between OSGi and DPWS. An OSGi service has to be accompanied by a specific skeleton object in order to be remotely accessible. The skeleton unmarshals incoming requests, invokes the OSGi services, and marshals the responses. This functionality is realized by a DPWS skeleton service.

To handle appearance, disappearance, and changes of OSGi services at runtime, skeleton objects are generated by the skeleton generator (size: 89 KB) dynamically on demand. For this purpose, the service tracker is extended. It manages the list of services to be offered remotely and triggers the skeleton generator which analyzes the input and output parameters of the declared actions. All parameter types have to be serializable to an XML representation and vice versa. The primitive types map directly to standard XML types, but the OSGi specification does not limit the parameter types at all. Custom types, therefore, need special marshaling services. If they are currently not available, they can looked up in an external bundle repository. Those marshaling bundles which are found are automatically downloaded, installed, and activated. The marshaling services are registered in the service registry and can be used for subsequent calls. In the next step, the corresponding DPWS skeleton service is generated and added to a DPWS device. We propose a one-to-one mapping which means that one bundle equates to one DPWS device. The declared and inherited methods of the OSGi service correspond directly to operations of the DPWS skeleton service.

To preserve the OSGi programming model, the usage of remote services should be identical to local services. Therefore, remote services must be registered locally under the same interfaces and properties. It is essential that the interface inheritance hierarchy is maintained and that all methods are declared in the proper interfaces. DPWS prescribes the usage of WSDL 1.1 which does not support interface inheritance innately. To solve this problem, the skeleton generator adds an additional Java-specific auxiliary Web Service to the DPWS device which exposes information about the underlying inheritance hierarchy and the mapping from actions to interfaces. This auxiliary service is only available if the OSGi service is based on an inheritance hierarchy. To indicate under which interfaces the service is registered, the port type is extended by the namespace and attribute `osgi:registered`.

In order to separate the concerns "Java type structure" and "OSGi service properties", information about the OSGi service properties can be retrieved via an OSGi-specific auxiliary Web Service. We use this approach and do not embed the property information into the WSDL documents of the services, because in this case each property change would lead to WSDL changes and cause communication overhead.

Finally, the generated DPWS device has to be started. This step corresponds to the activation in terms of distributed object systems. The DPWS device is passed to JMEDS which acts as the object adapter and transmits a "Hello" message.

7.2. Proxy

In order to provide location transparency, the usage of local and remote OSGi services must not differ. For this purpose, the proxy object is employed which implements the same interface as the remote service. This proxy service is registered in the local service registry.

The proxy generation is performed automatically when a "Hello" message is received. The proxy generator (size: 95 KB) manages the list of external services to be offered locally. In case of an appropriate "Hello", the generator analyzes the input and output parameters of the service and creates the local proxy. The proxy is an OSGi bundle which offers a local service with the same interface. Internally, it encapsulates a corresponding DPWS proxy service and performs the parameter marshaling in the same way as the skeleton on server-side. The proxy bundle registers its proxy service in the local service registry. It forwards all method calls to the encapsulated DPWS proxy service which finally carries out the remote method invocation.

To create an OSGi proxy service, its Java interfaces must be reconstructed first. For this purpose, the Java-specific auxiliary Web Service, provided by the DPWS device on server-side, is used. If it is unavailable, no interface inheritance is considered. The Java interface classes are created dynamically by means of Java bytecode generation. We use the ASM library [4] which is feasible even for resource-constrained devices.

In dynamic environments, no assumptions can be made in advance about the classes contained in a specific Java package. Therefore, every Java package is represented as a single bundle which hosts its interfaces as attached fragment bundles. The package is exported by the host bundle and imported by the proxy bundle. Thus, the Java interface class objects are available in the OSGi framework and can be used by other services. The attribute `osgi:registered`, described in Section 7.1, indicates under which interface(s) the OSGi proxy service must be registered. The service's properties can be retrieved by sending a request to the OSGi-specific auxiliary Web Service. If it is not available, no additional properties are considered. Hence, the OSGi proxy service can be registered in the service registry providing transparent access to the actual service.

7.3. Method Invocation

For a client, there is no difference between the usage of local and remote OSGi services. The client retrieves an OSGi proxy service from the service registry and casts it to the appropriate interface. The proxy service communicates with a corresponding skeleton service via SOAP messages. The skeleton passes method calls to the remote OSGi service. In this context, two aspects are of special interest: late binding and fault handling.

To support late binding, SOAP messages contain the denotations of the Java data types. The proxy and the skeleton interpret them in order to identify the data types which have to be instantiated. When JMEDS receives a SOAP request, it is forwarded to the DPWS skeleton service which calls the method of the actual OSGi service. This method call may fail and throw an exception. The skeleton represents the exception in a SOAP message containing a fault code and a description. The proxy service receives this message and decides whether to throw an exception or to unregister the service. It is important not to introduce a new fault model so that proxy and actual service do not differ in their behavior. A fault can be ascribed to the service itself or the underlying infrastructure. Faults that are not caused by the service itself are handled transparently. If retries do not succeed, the service is unregistered eventually.

7.4. Remote Event Notification

Events are typically notifications of significant changes in state that enable those interested in these changes to react accordingly. OSGi allows the development of applications based on an *event-driven architecture*. For this purpose, the Event Admin Service provides an inter-bundle communication mechanism which is based on a publish and subscribe model. In order to support remote notification, events published by an OSGi service must be transmitted to all OSGi frameworks holding a corresponding proxy service. For this purpose, the *event converter* (size: 42 KB) is provided, which registers an event converter service in the service registry. It is notified when an event is published within the OSGi framework. The bundle creates a DPWS event converter device representing the OSGi framework and hosting a service which propagates this event across OSGi boundaries. Thus, DPWS clients can subscribe to event types of relevance and are notified on subsequent changes. After instantiation, the service searches for other event converter services. It sends out a "Probe" message and subscribes automatically if an appropriate service is found.

When an event is published by a remote service, the Event Admin Service notifies the event converter service. It marshals the event's properties to an XML representation and transmits it via DPWS in the form of a notification message. The client reconstructs the actual OSGi event object from the received message. To publish this event in the OSGi framework, the event converter bundle makes use of the Event Admin Service. It determines which handlers must be informed and notifies them afterwards.

Remote event notification is realized by a central instance within the OSGi framework, the event converter bundle. DPWS, however, allows the subscription of clients to a specific service in order to receive events published by this service. This principle cannot be applied to OSGi services, because the specification only recommends the indication of the event publisher, but does not prescribe it. Therefore, an event received by an event handler cannot be certainly ascribed to its publisher and its corresponding skeleton service. The usage of a central instance is the only possibility to support remote event notification in a transparent manner. In order to avoid converting every event and sending it as a remote event to the subscribers, a filter determines only those which have to be transmitted.

8. Application Example

We evaluated our approach on several real world scenarios. To illustrate its applicability we chose a scenario from the home automation domain which comprises different device types. It demonstrates that not only a federation of OSGi frameworks, but also an integration of native DPWS services is feasible. Figure 3 depicts this example schematically.

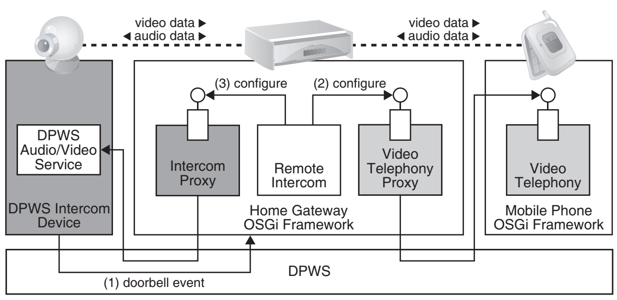

Figure 3. Example Scenario

The intercom system includes a doorbell and an audio/video module which allows two-way audio communication and one-way video transmission between a house resident and its visitor. This functionality is provided by the DPWS audio/video service of the DPWS intercom device. Furthermore, the scenario comprises two OSGi-based devices: the home gateway hosting the remote intercom application and the mobile phone providing the video telephony service. The home gateway plays the key role in the communication between the DPWS intercom device and the mobile phone. It offers access to the remote services by means of proxies generated by the proxy generator bundle, not shown in the figure. When a visitor rings the doorbell, the DPWS intercom device informs the home gateway via a doorbell event (1). Subsequently, the remote intercom application configures the video telephony service (2) and the DPWS audio/video service (3) to establish a connection with each other. Thus, audio and video data are not transmitted via DPWS, but out-of-band.

For evaluation purposes, we used three Athlon 64 X2 3800+ machines each equipped with 2 GB of main memory, Sun's JVM in version 1.6.0-b11, and Eclipse's Equinox OSGi framework in version 3.4.0. The machines simulating the intercom system, the home gateway and the mobile phone were connected by a 100 MBit/s Fast Ethernet. In this scenario, the DPWS skeleton service of the OSGi video telephony service was created in 1.1 ms on average. The generation and installation of the OSGi proxy service took 107.1 ms measured from the reception time of the "Hello" message. The duration of a remote method invocation including data transmission and processing time averaged 4.7 ms.

9. Conclusion

In this paper, we have outlined our work concerning the federation of multiple OSGi frameworks by means of DPWS acting as a bridge between them. Our solution integrates transparently OSGi into DPWS and vice versa, supports legacy OSGi services, and handles the problem of unreliable data transmission in distributed environments. We also consider that services can appear, disappear, and change unpredictably at runtime. The applicability of our approach has been demonstrated with a real-world scenario in the field of home automation.

Particularly in comparison with R-OSGi, our solution has the advantage that it relies only on broadly accepted open standards and avoids the transmission of Java bytecode across the network. As a result, remote access of OSGi services is not limited to OSGi- or Java-based clients. The gap between a self-contained SOA environment provided by OSGi and a distributed SOA environment based on Web Service technology could be successfully closed.

The work is funded by the German *Federal Ministry of Education and Research (BMBF)* in the context of the European ITEA 2 project *Open Source AMbient Intelligence (OSAMI)* [10].

References

[1] Nyota Online Documentation. http://eclipse.compeople.eu/wiki/index.php/Nyota:Main, 2007.

[2] A. Bottaro et al. RFP 86 - DPWS Discovery Base Driver, OSGi Alliance, 2007.

[3] A. Bottaro et al. Dynamic Web Services on a Home Service Platform. In *Proc. of the 22nd Int. Conf. on Advanced Info. Networking and Applications*. IEEE Computer Society, 2008.

[4] E. Bruneton et al. ASM: A Code Manipulation Tool to Implement Adaptable Systems. In *Adaptable and Extensible Component Systems*, 2002.

[5] S. Chan et al. Devices Profile for Web Services (DPWS) Specification, Microsoft Corporation, 2006.

[6] M. Demuru et al. The Domoware UPnP Service for OSGi, 2004. http://domoware.isti.cnr.it.

[7] E. Guttman et al. RFC 2608: Service Location Protocol, Version 2, IETF, 1999.

[8] E. Newcomer et al. RFP 119 - Distributed OSGi, OSGi Alliance, 2008.

[9] OASIS WS-DD Technical Committee. Web Services Discovery and Web Services Devices Profile (WS-DD), 2008. http://www.oasis-open.org/committees/ws-dd/.

[10] OSAMI-D Consortium. OSAMI Commons: Open Source AMbient Intelligence, 2008. http://en.wikipedia.org/wiki/OSAMI-D.

[11] OSGi Alliance. OSGi Service Platform Core Specification – Release 4, Version 4.1, 2007.

[12] J. S. Rellermeyer et al. R-OSGi: Distributed Applications through Software Modularization. In *Proc. of the ACM/IFIP/USENIX 8th Int. Middleware Conf.*, 2007.

[13] A. S. Tanenbaum et al. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001.

[14] G. Wütherich et al. *Die OSGi Service Platform*. dpunkt.verlag GmbH, 2008.

[15] E. Zeeb et al. WS4D: SOA-Toolkits Making Embedded Systems Ready for Web Services. In *Proc. on the 2nd Int. Workshop on Open Source Software and Product Lines*, 2007.