Towards the Web of Things: Using DPWS to Bridge Isolated OSGi Platforms

Oliver Dohndorf, Jan Krüger and Heiko Krumm
TU Dortmund University
Dortmund, Germany
(oliver.dohndorf, jan.krueger)@tu-dortmund.de

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

***Abstract*—Bringing heterogeneous devices like industrial machines, home appliances, and wireless sensors into the Web assumes the usage of well-defined standards and protocols. Our approach combines the Web Service standard for devices DPWS with the embedded system and component management standard OSGi. It implements the specifications of OSGi Remote Services, as well as OASIS Discovery, Eventing, SOAP-over-UDP, and DPWS. Furthermore, runtime WSDL generation and interpretation is supported, as well as the presentation URL feature which automatically provides a web browser user interface for interactive device access. Thus, our approach is an appropriate and comprehensive basis for the seamless, flexible, and standard-compliant integration of things into the Web.**

***Keywords*-DPWS-OSGi integration; embedded devices.**

I. Introduction

Web Services as a most widespread implementation of the *service-oriented architecture (SOA)* provide a comfortable way of creating flexible service-oriented applications for the Web. The enhancement of this approach is presented by the *Devices Profile for Web Services (DPWS)* [1] which targets resource-constrained embedded devices. Therefore, the field of application is significantly wide and variable. In the meantime, DPWS has been published as a specification within the *OASIS Web Services Discovery and Web Services Devices Profile (WS-DD)* [2]. The native integration of DPWS into Windows Vista and Windows 7 is another convincing point in favor of DPWS.

The *OSGi* specification [3] created by the OSGi Alliance defines an open, modular, and scalable local service delivery platform. Running within a *Java Virtual Machine (JVM)*, OSGi offers an in-JVM SOA. Exceeding JVM boundaries and providing distributed solutions are the main purposes of the *OSGi Remote Services* specification introduced in the latest version of the *OSGi Service Compendium*. Furthermore, several OSGi framework implementations allow the usage on resource-constrained devices.

Combining these two technologies is a promising way to ensure a wide field of application in bringing heterogeneous devices into the Web and building modular, distributed, service-oriented solutions. In this paper, we present our implementation of DPWS-OSGi mutual integration [4] and the latest results of our current work. In particular, we focus not only on the federation of isolated OSGi frameworks but also on the integration of native DPWS devices. We hope that our approach will be the next step on the way to the Web of Things where versatile devices and objects are connected seamlessly to the Web providing for modular, flexible, and service-oriented solutions.

The paper is structured as follows: Section 2 provides the technical background of our approach and introduces DPWS and OSGi technologies. Related work is presented in Section 3. In Section 4 we define the key requirements for the solution which is introduced in detail in Section 5. We demonstrate the applicability of our approach by means of an example from the medical home care domain and present some noteworthy performance measurement results in Section 6. Section 7 sums up the paper.

II. Related Work

The *Universal Plug'n'Play (UPnP)* was the first specification of a service-oriented infrastructure for embedded application scenarios. DPWS, being inspired by UPnP, defines a minimal set of Web Service standards and specifications targeting the provision of Web Service based communication for embedded devices. The special attention is paid to secure message transmission, dynamic discovery, description, subscription, and event notification.

According to the specification, a DPWS device hosts several services which can be discovered and used by DPWS clients. A device sends "Hello" and "Bye" when joining and leaving a network, respectively. Searching for particular services is performed through sending a "Probe" message. Matching services respond with "Probe Match" messages. "Invocation" messages are aimed for performing the service usage. The eventing mechanism comprises subscribing for particular event types by sending a "Subscribe" message and informing the subscribed clients through a "Notification" message. The basic messaging within DPWS employs SOAP using HTTP and SOAP-over-UDP.

Among the existing implementations of the DPWS specification, the open source *Java Multi Edition DPWS Stack (JMEDS)*[1], is characterized by its modular extensible architecture and remarkable features. These include interpretation and generation of service descriptions (WSDL) at runtime, a web browser user interface accessible via the presentation URL, and a small footprint.

A service-oriented standardized way of managing the software lifecycle is one of the main aims of the OSGi technology [5]. OSGi provides for the integration of pre-built, collaborative components and caters for the reusability and maintenance costs issues through dynamic service provision and update mechanisms. The OSGi specification defines the OSGi framework which offers an execution platform for Java-based components, called *bundles*. The platform permits to install, uninstall, start, stop, and update bundles at runtime without restarting the entire system. Moreover, extending the bundle's class path with classes or additional resources is possible through attaching a corresponding *fragment bundle* to the *host bundle*. The functionality of the bundles is offered in the form of services in a publish-find-bind way. The services are registered under one or more interfaces within the *service registry* and can be found by other bundles when necessary. *Event Admin Service* provides a generic mechanism to subscribe for and receive events from the framework or other services.

The extension of the self-contained SOA environment provided by OSGi to a Distributed OSGi framework was firstly addressed in RFC 119 [6]. This resulted in the OGSi Remote Services specification introduced in the latest version of the OSGi Service Compendium [3]. The challenge of bridging isolated OSGi frameworks has been taken up by several researchers up to now. Apache CXF [7] provides the reference implementation of the RFC 119 specification.

R-OSGi [8], developed by ETH Zurich, provides support for sharing OSGi services over a network and allows a centralized OSGi application to be transparently distributed among different OSGi frameworks. For this purpose, the application must be manually factored into distributable components. Dynamic proxy generation at runtime and a distributed service registry serve the aim of transparency. For data transmission, R-OSGi uses a proprietary binary protocol over persistent TCP connections. For realization of the distributed service registry jSLP [9] is used, a Java implementation of the *Service Location Protocol (SLP)* [10]. The service discovery mechanism of R-OSGi is extendable by other protocol implementations.

The Device Access specification in [3] defines the generic automatic detection and attachment of existing network devices on an OSGi framework. Following these concepts, the DPWS Discovery Base Driver [11], developed within the ITEA ANSO project, implements the integration of DPWS devices and services into OSGi. With this solution, it is possible to discover and use DPWS devices and services without concerns about underlying communication protocols. Similarly, the UPnP Base Driver specified in the UPnP Device Service Specification [12] defines a generic bridge between UPnP and OSGi technologies. One of its implementations was developed within the DomoWare project [13].

[1]WS4D.org Stack developed by TU Dortmund University and MATERNA

III. Requirements

Designing distributed applications, where no homogeneous devices are present and no centrally managed infrastructure is available, is a highly complex issue. Enabling everyday devices to connect to the Internet as well as making them discoverable, linkable, and usable by means of common open standards is not sufficient. There is a strong need for modular, global solutions for applications which are based on reuse, seamless integration, and runtime composition of services provided by these devices. The self-contained SOA environment of OSGi extended to exceed JVM boundaries is a suitable technology for this purpose, whereas DPWS allows solutions for devices with constrained resources.

We define the following key requirements for DPWS-OSGi mutual integration:

- *Location transparency:* The usage of local and remote OSGi and DPWS services must not differ for clients. Remote services are to be accessed as if they resided the local framework.
- *Support of legacy services:* Providing services remotely should not require any modifications.
- *Fault transparency:* The communication faults specific for a distributed environment must be handled in the same way as the reliability aspect is addressed by OSGi.
- *Dynamics:* The continuous changes in the topology imply that services appear, disappear, or become temporarily unavailable all the time. These facts should not impose any restrictions and must be regarded as a norm.
- *Manageability:* Local clients – DPWS as well as OSGi – should be able to access only those services that are intended to be remotely available. On the contrary, a mechanism should be provided to integrate only whitelisted remote services into a local OSGi framework.
- *Compatibility:* We set a high value on the ability to federate different OSGi implementations. Therefore, only standard OSGi services can be used in the solution. Moreover, the solution must also be applicable to those OSGi implementations which are designed for resource-constrained devices.

IV. WS4D.org DPWS-OSGi Integration

Addressing the listed requirements we integrate DPWS and OSGi technologies mutually and realize transparent cross-platform access to native services relying on existing security mechanisms. Figure 1 gives an architectural overview of our approach. According to the conventional model of distributed

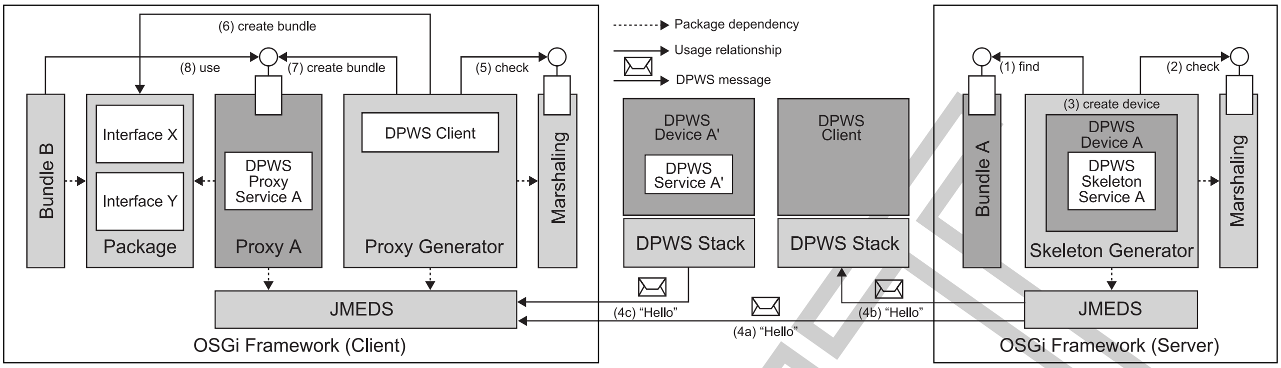

Figure 1. Skeleton and Proxy Generation

object systems [14], we distinguish between a client which uses and a server which provides a service. Acting as a server, the OSGI framework, on the right, intends to offer remote access to its OSGi service of the bundle *A*. At the same time, the native DPWS device *A'* provides access to its DPWS service *A'*. Bundle *B*, on the client OSGi framework is able to use these services. The native DPWS client has also the ability to use the service of the bundle *A* installed in the server OSGi framework. On finding the service (1), the *skeleton generator* bundle checks the availability of the required marshaling services (2). After that, it generates the DPWS device *A* hosting the corresponding DPWS skeleton service *A*. JMEDS which is wrapped as an OSGi bundle announces the presence of the DPWS device by sending a "Hello" message (4a) to the client OSGi framework and to the native DPWS client respectively (4b). On receiving a "Hello" message from the server OSGi framework (4a) or from the native DPWS device (4c), the JMEDS bundle on the client-side informs the *proxy generator* bundle. After checking the required marshaling services (5), the proxy generator generates a package bundle containing the required interfaces (6). It creates the DPWS proxy service *A* and proxy *A* bundle with the appropriate OSGi proxy service (7). Thus, bundle *B* is able to use the OSGi service provided by bundle *A* and the native DPWS service *A'* respectively (8).

A. Skeleton

In order to be remotely accessible, an OSGi service must be supplied with a corresponding skeleton object. Thus, the DPWS skeleton service unmarshals incoming requests, invokes the corresponding OSGi services, and marshals the responses. Skeleton objects are generated by the skeleton generator on demand. So, the issue of dynamics of OSGi services at runtime, i.e. appearance, disappearance and changes of services, is addressed. On arrival of services, the skeleton generator recognizes it and analyzes the input and output parameters of the declared actions. Being serializable to XML representation and backwards is a basic requirement for parameter types. The primitive types map directly to standard XML types, the OSGi specification, however, does not restrict the parameter types to the primitive ones. So, custom types need special marshaling services. In case they are not available, they can be looked up in an external bundle repository, downloaded, installed, and started automatically. Further, they are registered in the service registry and can be used later on. After that, the corresponding DPWS skeleton service is created and added to a DPWS device. One DPWS device is generated for each remotely accessible OSGi bundle. The declared and inherited methods of its OSGi services map directly to DPWS skeleton service's operations.

It is important that the usage of remote services does not differ from the usage of local services for a client. As a result, remote services are registered locally under the same interfaces and properties. A great challenge is to preserve the interface inheritance hierarchy, i.e. to declare all the methods in the proper interfaces. WSDL 1.1 which is used in DPWS, however, does not support the interface inheritance hierarchy natively. In order to handle this, an additional Java-specific auxiliary DPWS service is added to the device to provide information about the inheritance hierarchy and the mapping of actions to interfaces if needed. Moreover, an OSGi-specific DPWS service is provided, in order to expose information about the OSGi service properties. So, we do not embed the property information into the WSDL documents of the services, in order not to cause communication overhead on each property change. The information offered by the auxiliary services is useless for communication with native DPWS clients and is ignored in this case.

Finally, the generated DPWS device is started. In terms of distributed object systems, we speak about activation. JMEDS, acting as an object adapter, gets the generated DPWS device and sends a corresponding "Hello" message.

B. Proxy

For the purpose of location transparency, the proxy object which implements the same interface as the remote service is employed. It is registered and can be looked up in the local service registry. On receive of a "Hello" message, the proxy generator which holds a whitelist of external services to be offered locally creates the local proxy automatically. The generated proxy is an OSGi bundle offering the local OSGi service under the same interface. Similarly to the server-side skeleton, the encapsulated DPWS proxy service performs the required parameter marshaling and carries out the remote method invocation according to the forwarded method call.

Creating an OSGi proxy service requires the reconstruction of the corresponding Java interfaces, firstly. In case of connecting isolated OSGi frameworks, a Java-specific auxiliary service, provided by the server-side DPWS device, is involved. It is assumed that no interface hierarchy is present, if the service is not available. The properties of the service can be requested from the OSGi-specific auxiliary service. If it is not available, it is assumed that no additional properties are present. In case of integration of native DPWS devices, the existing rules for mapping WSDL to Java are used. By means of Java bytecode generation, the Java interface classes are created dynamically. We use the ASM library [15] which is suitable for embedded devices. In dynamic environments, no assumptions can be made in advance about the classes contained in a specific Java package. This is relevant for integrating remote OSGi services where the packages are fixed as well as for native DPWS services where the packages are deduced from the WSDL. 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.

C. Method Invocation

For OSGi clients there is no difference between the usage of local or remote OSGi services and native DPWS services respectively. The client retrieves an OSGi proxy service from the service registry and casts it to the appropriate interface. Equally, there is no difference for native DPWS clients in the usage of native DPWS services and OSGi services.

OSGi client using remote OSGi service 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 and instantiate the identified data types. SOAP requests received by JMEDS are forwarded to the DPWS skeleton service which calls the corresponding method of the actual OSGi service. If the method call fails and an exception occurs, a SOAP message containing the exception type and content is returned to the proxy.

OSGi client using native DPWS service In this case, the SOAP messages used for communication between proxy and a native service does not contain Java data types. Instead, the proxy service decides on the basis of predefined rules which Java data type is to be instantiated. Faults within the native service invocation are submitted within the fault element of the SOAP message. They are mapped to appropriate Java exceptions and thrown in the proxy service.

DPWS client using OSGi service The DPWS client communicates with the skeleton service representing an OSGi service. The exceptions are mapped to the SOAP message's fault element and can be processed by the client. As noted above, the SOAP message of the method invocation does not contain Java data type information, since a native client can not handle it.

In any case, the method invocation may fail due to network problems resulting in SOAP messages getting lost. In OSGi, the proxy service receives a SOAP fault and unregisters the service if retries do not succeed. It is important not to introduce a new fault model so that proxy and actual service do not differ in their behavior. The reaction of native clients regarding SOAP faults can not be influenced.

D. Remote Event Notification

The eventing mechanism in the OSGi specification is realized by means of the Event Admin Service. It provides a publish-subscribe model for communication within a single OSGi framework. To support remote notifications, events published by an OSGi service have to be forwarded to all OSGi frameworks that hold the corresponding proxy service. Thus, the *event converter* bundle provides an event converter service and registers it in the service registry. On events published within the OSGi framework, it is notified. A DPWS event converter device represents the OSGi framework and hosts a service to propagate the event across OSGi boundaries. DPWS clients subscribe to event types of interest and get notifications on changes. In case of integration of native DPWS devices, the DPWS proxy service subscribes by the native services directly.

The Event Admin Service notifies the event converter service on events published by a remote service. The event converter marshals the event's properties to XML representation and sends it as a DPWS notification. The native DPWS client can handle this notification straightaway. In order to reach the corresponding OSGi client, the corresponding OSGi event is reconstructed from the received DPWS message. Finally, the event converter bundle determines which handlers must be informed and notifies them.

In DPWS, it is allowed to subscribe to a specific service. The OSGi specification, in contrast, only recommends the indication of the event publisher, but does not prescribe it. Thus, it is not possible to assign a received event to its publisher and corresponding skeleton service definitely. To handle this, the remote event notification is performed by a central instance of the event converter within the OSGi framework. A filter is used to specify, which events are to be published remotely.

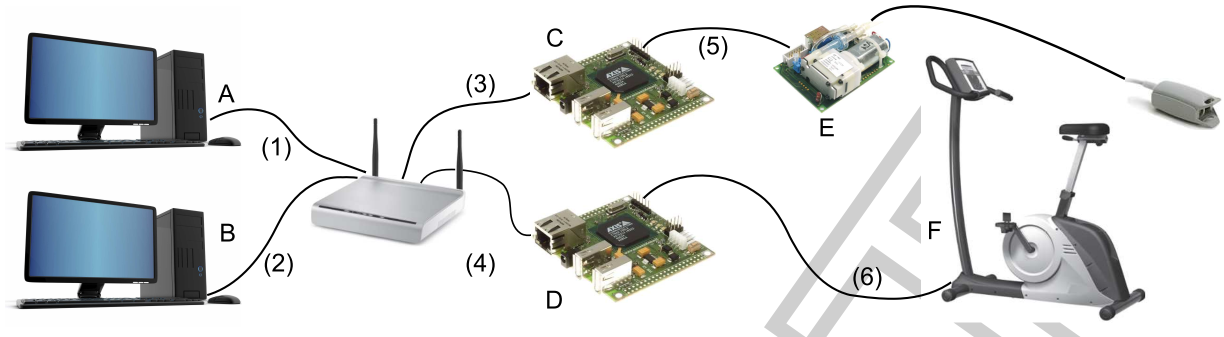

Figure 2. Structure of the Application Example

For DPWS services that offer evented operations, the event converter is not needed. The proxy subscribes to the operation itself and publishes received events as OSGi events in the local platform using the Event Admin Service. If native clients are interested in OSGi events, they simply subscribe DPWS events provided by the event converter.

V. Application Example

We have tested and evaluated our solution on several real world scenarios. To demonstrate its capabilities, we present a simplified example from the medical home care domain which was adapted from the more sophisticated scenario used within the research project OSAmI [16].

According to the scenario, a cardiac *patient* has to conduct a series of rehabilitation trainings at home which has to be telemedically supervised by a medical *supervisor*, e.g. a physician or a sports scientist. Therefore, the patient is equipped with a training device, a bicycle ergometer, and some medical sensors to monitor his vital signs. In this example, pulse rate and blood oxygen saturation are monitored. Furthermore, the patient is supplied with a so-called *home gateway*, which provides a platform for running the application to control the training device and medical sensors and handles the communication with the supervisor. During the training, the application supplies the patient with information about his health state and the training's progress. The training itself is divided into three phases: warm up, actual training, and cool down. For each phase, the supervisor defines the appropriate settings like the target ergometer load as well as upper and lower thresholds for the vital signs. To adjust these settings at runtime and to monitor the training session, the supervisor is supplied with the appropriate *supervisor application* which runs on the so-called *clinic gateway*. In our example, the home gateway and the clinic gateway are in the same LAN. Within the OSAmI project, we will implement this demonstrator as a real world solution with the gateway devices connected to the Internet. Communicating over the Internet will enforce the consideration of several additional security aspects, which are not regarded here for the sake of simplicity.

Figure 2 depicts the hardware involved in the scenario. Standard PCs with x86 Intel Core 2 Duo CPUs and 1 GB RAM each represent the home gateway (A) and the clinic gateway (B). The communication link between these two gateways is provided using a switched 100 MBit/s network connection (1,2). As an interface to integrate the medical sensor (E) and the bicycle ergometer (F), which both do not have a network connection interface, the Foxboards are used (C,D). Technically, the Foxboards are Linux-based embedded systems with a 100 Mhz Axis Etrax 32 bit RISC-based CPU, 32 MB RAM and 8 MB ROM [17]. On the one hand, the Foxboards are connected to the medical sensor using a serial 5V-TTL-level or to the ergometer with a RS-232 connection (5,6). On the other hand, the Foxboards are connected to the network using their on board network interfaces (3,4).

The software comprised in the scenario consists of a simple supervisor application and a home gateway application. Both are realized as OSGi bundles running in separated OSGi frameworks on the clinic (B) and the home gateway (A) respectively. The home gateway PC uses Microsoft Windows XP as an operating system, where as the clinic gateway PC uses Windows Vista. The Foxboards host native DPWS services to provide access to the functions of the medical sensor (C,E) and the ergometer device (D,F). These services are implemented using JMEDS and run inside a *Kilobyte Virtual Machine (KVM)*, a minimalist JVM for resource-constrained devices. Invoking an operation of the DPWS services (e.g. setting the target load of the ergometer device (D)) is translated into the appropriate command sequence for the serial communication which is transmitted to the device connected to the serial port. Vice versa, data received over the serial connection from the medical sensor or the ergometer is made available by means of the corresponding DPWS services (e.g. the result of a pulse measurement received from the medical sensor is published in the network as a DPWS event). The DPWS service to control the ergometer as well as the service to use the medical sensor are needed in the training control application running on the home gateway. For this purpose, they are integrated into the OSGi framework using our DPWS-OSGi integration. Hence, the usage of the DPWS service (e.g. setting the ergometer target load) does not differ for a client from the usage of a regular OSGi service. Also, the received DPWS events (e.g. pulse measurement results) are published within the OSGi framework as regular OSGi events. The application on the home gateway provides in turn some functionality to be used by the application running on the clinic gateway. In detail, a function to define the mentioned phase settings is provided, as well as alarm events which are fired if the patient's pulse rate exceeds the defined threshold levels. Here, the use of DPWS-OSGi integration allows to implement this function as a regular OSGi service which is registered in the home gateway framework. The alarm events can be realized as regular OSGi events which are propagated into the home gateway framework. According to the configuration of the bundles in the home gateway OSGi framework, this service is made available for remote use, and the OSGi events are propagated as DPWS events. The bundles in the clinic gateway OSGi framework, on the other hand, are configured to integrate this remote service into the local framework and to propagate the received remote events as OSGi events into the framework. Thus, the supervisor application can use the remote service to configure the training phases and remote alarm events to monitor the patient during the training.

The outcomes of measurement experiments in that scenario shall give a hint about realistic footprints and performance values. The generated proxy bundles are relatively small in size. For example, the proxy bundle of the service to configure the phase settings has a size of just 3 kB. In case of integrating the ergometer control service, a native DPWS service, the proxy has a size of 4.1 kB. The proxy bundles for remote service access were created at runtime within 107 ms. The generation of the skeleton for a remote OSGi service took only 1.1 ms. The average time for invoking a remote OSGi service, running on the home gateway, was 4.7 ms and the average time for invoking a native DPWS service, hosted on the embedded device, was 14.8 ms.

VI. Conclusion

In this paper, we have presented our current work concerning the mutual integration of the DPWS and OSGi technologies. The solution allows to connect isolated OSGi frameworks and supports the usage of native DPWS devices and services. In comparison to the DPWS Discovery Base Driver, our remote service integration is fully transparent and in contrast to R-OSGi relies only on broadly accepted open standards and avoids the transmission of Java bytecode across the network. As a result, the remote access to OSGi services is not limited to OSGi- or Java-based clients. The applicability of the approach was demonstrated by means of a real-world example from the field of home medical care.

Acknowledgment

This work is funded by the German Federal Ministry of Education and Research (BMBF) within the context of the European ITEA 2 project *Open Source Ambient Intelligence (OSAmI)* [16].

References

[1] S. Chan *et al.*, "Devices Profile for Web Services (DPWS) Specification," 2006.

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

[3] OSGi Alliance, "OSGi Service Platform Core Specification & Service Compendium – Release 4, Version 4.2," 2009.

[4] C. Fiehe, A. Litvina, I. Lück, F.-J. Stewing, O. Dohndorf, J. Krüger, and H. Krumm, "Location-Transparent Integration of Distributed OSGi Frameworks and Web Services," in *Proc. of the IEEE 23nd International Conference on Advanced Information Networking and Applications - Workshop SOCNE (AINA 2009)*, Bradford, UK, May 2009.

[5] G. Wütherich, N. Hartmann, B. Kolb, and M. Lübken, *Die OSGi Service Platform.* dpunkt.verlag GmbH, 2008.

[6] E. Newcomer *et al.*, "RFC 119 - Distributed OSGi," 2008.

[7] Apache Software Foundation, "Apache CXF RFC 119 Implementation," http://cxf.apache.org/distributed-osgi.html, 2009.

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

[9] "jSLP Documentation," http://jslp.sourceforge.net, 2008.

[10] E. Guttman *et al.*, "RFC 2608: Service Location Protocol, Version 2," 1999.

[11] 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.

[12] A. Bottaro, "RFP 72 - Extended Mapping for UPnP Discovery Transparency," April 2006.

[13] M. Demuru, F. Furfari, and S. Lenzi, "The Domoware UPnP Service for OSGi," http://domoware.isti.cnr.it, 2005.

[14] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001.

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

[16] OSAmI-D Consortium, "OSAmI Commons: Open Source Ambient Intelligence," 2009, http://www.osami-commons.org.

[17] Acme Systems Srl., "Foxboard LX832: a complete Linux system," 2009, http://foxlx.acmesystems.it/?id=4.