Network-centric Middleware supporting dynamic Web Service Deployment on heterogeneous Embedded Systems

Jens Schmutzler¹, Ulrich Bieker², Christian Wietfeld¹

¹Communication Networks Institute, University of Technologies Dortmund, Otto Hahn Straße 6, 44227 Dortmund, Germany, {jens.schmutzler, christian.wietfeld}@tu-dortmund.de
²PRO DV Software AG, Hauert 6, 44227 Dortmund, Germany, ulrich.bieker@prodv.de

Abstract
The market for embedded systems performing dedicated tasks is growing in large numbers in recent years. They are utilized in manifold application areas for different use case scenarios. Three prevailing fields are pervasive healthcare, homeland security applications, and environmental mitigation management including energy conserving applications. The increasing demands for remotely manageable monitoring purposes ask for flexible networking concepts and interoperability of the involved devices with standard Web Services. Furthermore due to the prevailing platform heterogeneity of the embedded systems market, software development on these platforms became a very complex and time consuming field of work. This paper describes an efficient way to counter these new requirements by utilizing the Device Profile for Web Services (DPWS) framework on top of the OSGi platform.

Keywords
Web Services, Middleware, Embedded Systems, DPWS, OSGi

1 Introduction

The MORE project [MORE] is an ongoing research project funded by the European Union in the 6th Framework Programme which focuses on easing the development of services on embedded platforms. It aims to provide a middleware for embedded systems enabling application developers to deploy services remotely. Reference use cases in the pervasive healthcare and environmental mitigation management domain allowed us to derive real world requirements and serve as validation scenarios during the early deployment phase of the project. In order to keep the design as generic as possible, no assumptions are being made on the capabilities of the communication links. Hence the middleware encompasses different deployment scenarios varying from personal area to wide-area incl. cellular 2/3G networks.

We start with an introduction of the MORE use case scenarios in the following section and highlight major technical requirements of the MORE Middleware. Section 3 introduces the architecture of the MORE middleware and its integrated service environment. In section 4 we will take a look at the core technologies utilized in MORE in order to support Web Services and their remote deployment. Section 5 summarizes early experiences with the first MORE middleware core prototype running on an embedded platform followed by the conclusions and an outlook on future research topics.

2 MORE Use Case Scenarios

The MORE middleware architecture is validated by two independent use case scenarios in which real end users are involved. During the early stages of the MORE project all use case
requirements were gathered and compiled into generic middleware requirements. These requirements were the basis for the specification of the middleware architecture, the technical test cases, and validation scenarios to test a fully deployed MORE middleware at the end user’s site. All scenarios present current research areas in H2M or M2M communication.

2.1 Environmental Mitigation Management

The Environmental Mitigation Management scenario deals with continuous and discontinuous measurements for forestry applications. At present it is quite costly to conduct these measurements and to collect all data, as to this day these measurements are performed manually. An automated and wireless transfer of all measurement data would be a major progress for forestry departments and ecological associations. This involves the deployment of remote sensors and gateways in order to monitor and transmit all relevant environmental information to the respective departments or associations. In addition to the economic advantages of such a remote monitoring system, it could also be used as real-time early detection system for ecological disasters like forest fires. Figure: 1 shows the basic setup of an early prototype implementation of a gas monitoring system for risk assessment during fire fighter operations. The prototype demonstrates the collaboration of Web Services and smallest embedded devices in a Java-based environment, utilizing a GSM/GPRS link for communication between the Web Service and the client.

In this setup a gas sensor is connected via a serial link to a GSM/GPRS wireless module which also supports localization functions through GPS. Moreover the wireless module offers a J2ME environment following the CLDC – MIDP 2.0 specification. The Web Service running on the wireless module publishes its service description to a server in the enterprise domain. A service proxy is used in order to overcome possible shortcomings due to NAT depending on the mobile network provider’s infrastructure. A client can search for a service within the enterprise domain and retrieves all necessary service descriptions through the server. The user can subscribe to an area of interest on a geographical map and retrieves from now on all valuable information from sensors located within the subscribed area. The prototype offers both continuous monitoring capabilities and discontinuous alarm notifications in case a previously defined threshold is exceeded at the gas sensor’s site.

This rather simple demonstrator application shows the general potential of wirelessly connected Web Services. Obviously the type of sensor can be exchanged in order to move towards different application domains. Possible examples would be energy conserving or homeland security applications which involve a service environment set around home automation appliances. Typically the developer then needs to implement a new service
satisfying the requirements of the new domain and afterwards has to deploy it on the wireless module. Here the MORE middleware supports development in form of predefined services for common functionality and a generic API helping to reduce the development time. Additionally MORE offers mechanisms for remote service deployment.

### 2.2 Pervasive Healthcare

The Pervasive Healthcare scenario focuses on continuous and discontinuous measurements of blood sugar values of diabetic patients. The adjustment to an appropriate insulin dose could be made far more efficient and convenient for the patient if mobile Web Services would be utilized. Ideally the measurements should be reviewed by the doctor while the patient is living and working in his/her regular environment. Adjustments to the insulin dose could be made based on the remote measurements. Similar to the Environmental Mitigation Management scenario the patient’s mobile phone can be utilized to connect to a blood sugar measurement sensor and to transmit all measurements to a hospital server where the doctor on duty can access and analyze them. In case of hypoglycaemia (low blood sugar level) patients can suffer severe damages and might be unable to set up an emergency call on their own. Thus having an alarm notification service constantly watching the blood sugar level would be a great benefit for diabetes patients.

Even though most technical requirements in this scenario are similar to the Environmental Mitigation Management scenario, the main emphasis on middleware functionality is kind of different. Prevailing points in this scenario are set around reliability and security issues of the chosen communication channel and additionally around dynamicity and mobility of all involved end-users. In order to respect the mobility of the diabetes patients the middleware has to handle interface handovers depending on the currently available network types and depending on customized interface metrics. The dynamicity of this scenario refers to the possible churn rate of mobile devices and their services. Service discovery mechanisms ensure that all information regarding a joining or leaving device is properly propagated within the entire network. All these requirements are met by the MORE middleware by offering a base level of generic services which help to counter those issues in an efficient way from the developer’s perspective.

### 3 Middleware Architecture – The SOA Approach

The core of the MORE middleware follows the paradigms of service oriented architectures embedded into a classical layered approach which is illustrated in Figure 2 [Michaelis 2007].
Four layers are identified: The Application Layer on top, the MORE Middleware Layer, the Operating System Layer and the Hardware Layer at the bottom. The topmost layer features the application developer’s view using the MORE middleware. Use case specific applications can rely on the interfaces provided by the middleware which abstracts the functionality from the underlying platform. The lower layers, operating system and hardware, allow the middleware to access specific functionality like setting up communication links and managing the hardware resources.

The second part of the MORE architecture concept adapts a service oriented approach for embedded devices targeted in MORE. The middleware provides its functionality to the application developer and between single middleware components through messaging and API function calls. One major objective in this regard was to define a middleware specification based on a modular concept to enable high reusability. By abstracting generic middleware functionality into services, these services can be utilized in various application scenarios.

Each service sends and receives messages through queues that are linked to connectors. Implementation of at least one pair of connectors is mandatory for each service that wants to communicate with another. Listeners are the input connectors used to receive messages from other services and put these messages in input queues. Notifiers are output connectors, used to get messages from output queues and send these messages to other services. Since standard Web Services introduce quite a lot of overhead on the communication channel, the concept of exchangeable connectors was introduced in order to provide communication means specifically tailored for embedded devices [Wolff 2007].

The MORE middleware will provide a set of enabling services offering common functionality as well as abstract services, which can be used as templates for easier implementation of application specific functionality. For the application developer or the operator, who runs a selection of MORE services to satisfy the demands of specific scenarios, the complete set of MORE services offers a toolkit with several building blocks. A high level of independence was a key principle when specifying the services, which in turn eases the selection process of which services need to be installed on a device.
Web Services for Embedded Systems

The runtime of the MORE middleware is based on the Device Profile for Web Services specification [DPWS]. DPWS identifies a minimal set of Web Service specifications tailored towards the needs and capabilities of embedded devices in order to allow for a base level of interoperability between devices and standard Web Services. The Java-based DPWS stack utilized for the MORE middleware currently supports the WS-Addressing, WS-Discovery, WS-Transfer, and WS-Eventing specifications. The support of WS-Eventing implements a publish/subscribe mechanism which allows devices to subscribe to asynchronous messages (events) triggered by a given user defined service and hence satisfies one of the major requirements of the validation scenarios for remotely manageable monitoring services.

All services implemented on top of the MORE middleware underlie the typical Web Service development process. Developing a service starts by defining its WSDL description which carries information about involved operations and data types used as parameters and return values of the operations [WSDL]. The DPWS4J [DPWS4J] stack generates all necessary interfaces and proxies from the WSDL definitions which enable the developer to concentrate on the service logic on the server side and on the service invocation or eventually the event handler which handles incoming events the user has subscribed to on the client side.

In addition to simplify the development process of services, MORE also considers a remotely manageable service lifecycle. This becomes crucial in scenarios like the environmental mitigation management, where it is not feasible to locally maintain a lot of sensor nodes which might be even hard to access physically. The approach followed in MORE is to utilize the OSGi platform for managing the MORE middleware and the user services.

OSGi technology [OSGi] provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle which enhances the value of a wide range of devices that use the Java™ platform. OSGi is located inside a Java virtual machine, but with the capability to access functions of the underlying system via JNI. The two main concepts of OSGi are the OSGi framework and the OSGi bundles.

OSGi bundles are the behavioural components of OSGi. They can contain applications or libraries that can share functionalities via services to other bundles that are located on the same device or anywhere else. The functionality of a bundle is clustered inside a JAR file with a description of the provided services. Additionally OSGi specifies a set of standardized bundles for the development of proprietary applications such as the “logging”, “configuration”, “http”, “device discovery”, and “UPnP” etc.

The main control unit of OSGi is the so-called OSGi framework. It is responsible for managing the life-cycle of a bundle such as installing a bundle, checking dependencies towards other bundles, starting and stopping a bundle and the corresponding services, replacing and updating a bundle, and removing a bundle from the framework. Besides these core management functionalities the framework also includes a local service registry for services of the installed and started bundles and security mechanisms that separate each bundle from others.

We implemented an OSGi compliant bundle of our DPWS stack in order to combine the benefits of both DPWS and OSGi. The implementation of the bundle has been successfully tested with both the Felix OSGi R4 platform from Apache [Felix] and the Equinox OSGi R4 platform from Eclipse [Equinox]. The use of OSGi has no major influence on the amount of code to be written by the developer as the Eclipse platform [Eclipse] inherently supports OSGi. The developer has to implement a service activator class in order to hook the DPWS-based service as a bundle to the OSGi platform. This enables dynamic Web Service deployment as OSGi supports to install, uninstall, start, stop, and update a bundle during runtime without restarting the Java virtual machine. Being installed on a device, the MORE Runtime bundle is responsible for inspecting a predefined bundle directory. All MORE Service bundles located in this directory are automatically detected and added to a DPWS registry. As soon as they are registered, they
provide full support of the DPWS stack and are able to be discovered, exchange Meta-Data, and to be invoked remotely. Service updates can be distributed as OSGi bundles which are placed into the OSGi bundle directory and then started in order to complete service deployment. Following this approach eases the process of DPWS service management on embedded devices significantly.

Figure 3: MORE Services and DPWS Runtime managed by OSGi

A few more changes have been introduced to the standard DPWS4J stack in order to simplify the development process. The main difference lies in the development of the server. While in DPWS4J it is necessary to develop a server class implementing a server loop, the MORE intermediate prototype already provides the server embedded into the MORE Core Runtime bundle. As previously explained the MORE Core Runtime bundle is responsible for inspecting a given directory and launch all present MORE Services bundles. Opposing to the standard DPWS4J stack the MORE Core Runtime bundle provides a core server where all MORE services will be added. Compared to standard DPWS4J service development procedure, the process is simplified, as the development of a server class is not needed anymore. Using this simplified procedure, a MORE service developer can focus on implementing the necessary OSGi service activator classes and the service logic itself. The Eclipse PDE Wizard creates a sketch of the service activator classes but it still has to perform the creation of a DPWS device model. A device model defines a set of service classes that a device of this particular model can host. After its creation the service class can be added to the device model.

Exporting the created MORE services as bundle is the last step of the Service implementation process. The Eclipse PDE tools provide a convenient export wizard to perform this task. Moreover, it allows configuring the target platform of the bundles which enables their deployment on many different environments. Using the PDE manifest editor export wizard of the Eclipse IDE, the developer finally obtains a fully deployable bundle compacted as a JAR file representing a MORE service.

5 First Intermediate Prototype Experiences

In order to validate the feasibility of our DPWS-based MORE Middleware approach we ran some early tests with our Intermediate Prototype on a Gumstix Verdex XL6P [Gumstix] which is one of the embedded MORE target platforms.

The Gumstix is a full function open source computer which is available in various configurations and can be individually extended through expansion boards. The motherboard has two connectors for expansion boards on both sides. This allows the use of up to two expansion boards at the same time leading to a sandwich like hardware configuration. Our prototype platform is based on an XScale PXA270 CPU (ARM 5) with 128MB RAM and 32MB ROM.
Two expansion boards (netwifimicroSD-eu and console-vx) have been attached to the motherboard offering several hardware interfaces. The Gumstix serves as a gateway between connected sensors and Web Services. Ethernet or WiFi 802.11b/g are used as Web Service interfaces, whereas USB, a serial port, GPIO, or the I²C Bus can be used as interfaces to the sensor.

In order to enable Java on the Gumstix the latest GNU Classpath 0.97.1 [Classpath] and JamVM 1.5.1 [JamVM] have been cross compiled for the ARM platform and installed on the Gumstix. The DPWS stack utilized for the MORE middleware is J2ME CDC or J2SE 1.4(+) compliant. In order to test proper functionality of the DPWS stack regarding the given platform constraints a test application has been implemented which respects all aspects of the implemented DPWS specifications: WS-Addressing, WS-Discovery, WS-Transfer, and WS-Eventing. This test application incorporates a service instance running as a server deployed on the Gumstix. A DPWS client was also implemented and started on another node within the same local area network. All tests of the WS-Addressing, WS-Discovery, and WS-Transfer went fine. Regarding the WS-Eventing specification the test service makes use of all defined message types in [WS-Eventing] which includes the subscription to an event, its renewal, a get status request, an unsubscribe message, and a subscription end message in case the event source terminates a subscription unexpectedly. All these tests succeeded on the previously described platform.

Even though there are currently no benchmarks available, some preliminary conclusions can be put to record. Although DPWS does provide adequate features to address common management and usage issues of device networks (Addressing, Discovery, Description, Control, and Eventing), it currently requires participating devices to support the Simple Object Access Protocol and therefore the processing of rather complex XML documents. For the Gumstix platform which is comparable to todays moderately sized devices like PDA’s or Wi-Fi routers this seems to be acceptable. However, looking at even smaller devices like pure sensor nodes the support for DPWS is troublesome due to more rigid resource constraints. In order to meet those resource constraints it is proposed to employ code generation that will allow the development of service specific DPWS stacks rather than a generic Java-based DPWS library.

6 Conclusions

Deploying DPWS on top of an OSGi platform is a feasible approach in order to bring the Web Services world to the embedded systems domain in an efficient manner. Both time and effort needed for development and deployment of services on embedded platforms are significantly reduced. Moreover the dynamic management of services through the OSGi platform provides a profound basis for resource management enhancements on Java-based embedded devices which still needs some more elaboration. Future work will focus on this topic and further enhancements for the developer’s perspective are planned which shall simplify the integration and chaining of the generic set of base level services provided by the middleware. All results regarding Java development on the Gumtix platform will be contributed to the developer community.

References


DPWS: S. Chan et al., “Device Profile for Web Services”, February 2006


JamVM: JamVM on sourceforge, WWW page http://jamvm.sourceforge.net/, accessed 19.03.2008


