

# A Service-Oriented Middleware for Integration and Management of Heterogeneous Smart Items Environments

Jürgen Anke<sup>#+</sup>, Jens Müller<sup>\*</sup>, Patrik Spieß<sup>\*</sup>, Leonardo Weiss Ferreira Chaves<sup>\*</sup>  
<sup>\*</sup> SAP Research, CEC Karlsruhe – <sup>#</sup> SAP Research, CEC Dresden  
<sup>+</sup> Dresden University of Technology

{juergen.anke, jens.mueller, patrik.spieess, leonardo.weiss.f.chaves}@sap.com

## Abstract

*In this paper, we describe concepts for a middleware that enables communication between smart items networks and business applications. Smart items networks consist of RFID systems, wireless sensor networks, and embedded systems. The integration of such devices into business applications is challenging, since each device type has its own protocols and programming interfaces. Existing middleware simplifies development of software which runs on smart items but not integration with back-end systems. We propose middleware concepts that abstract from the smart items network on a service-oriented level to ease integration into business applications. Our middleware allows the development of business applications which are able to deploy, run and query services from the network without any knowledge of the underlying smart items network.*

## 1. Introduction

The term smart items denotes real world objects that are made intelligent by using certain technologies such as RFID tags or attached wireless sensor nodes which are more powerful because they have their own energy supply allowing them to execute more complex software, in this context called services.

A middleware supporting the communication with smart items allows new applications and improvements to existing ones. Examples are automatic product monitoring, and decentralization of business logic for faster response in critical situations. The key challenge for this integration is the heterogeneity of smart items platforms. Therefore, the goal is to abstract from these differences and provide business applications with a uniform way to communicate with smart items.

The concepts we have developed are based on real-world cases and requirements. This is achieved by working on a middleware called (SI)<sup>2</sup> - Smart Items Services Infrastructure. We developed (SI)<sup>2</sup> to unify the middleware architectures of two EU-funded projects.

The CoBIs<sup>I</sup> (Collaborative Business Items) project aims to develop a new approach to business processes involving physical entities such as goods and tools in enterprise environments. An important goal of this project is to apply recent advances in the area of sensor networks, in order to distribute business logic functionality to "smart" physical entities thus reflecting what is actually happening in the real world.

The PROMISE<sup>II</sup> (Product Lifecycle Management and Information Tracking using Smart Embedded Systems) project aims to leverage the potential of smart embedded networked systems for product lifecycle management. By monitoring the status and operational conditions of products, real-world data is collected to support decisions for predictive maintenance, recycling, and product design improvements.

In this paper, we present concepts for platform-independent description, deployment, and invocation of services. It is structured as follows: Section 2 analyzes the requirements for our middleware. Section 3 presents related work. In section 4, parts of our architecture are described. Finally, section 5 concludes the paper with an outlook.

## 2. Analysis

### 2.1. Requirements

The smart items we want to support are of different types, all of which have different characteristics. These

---

<sup>I</sup> <http://www.cobis-online.de/>

<sup>II</sup> <http://www.promise.no/>

characteristics pose requirements on our middleware concept, as we aim to provide applications with a uniform access to these smart item types. In the following, we describe the key characteristics and derive requirements from them.

**Multiple platforms.** In a smart items environment there can be a number of different device types which support specific communication protocols and component technologies. Examples are OSGi, UPnP, Particle sensors, and Mote sensors. Thus, it is required to get a uniform access to functionality of all of these platforms, to allow for factoring out common functionality across platforms and easy extensibility to new platforms.

**Component mobility.** The invocation of services shall be independent of the physical location of the component which provides that service. Mobile smart items might connect to different middleware nodes, depending on their location. This changes the network topology and might also change the IP address and other network parameters of that node.

**Intermittent connections.** If smart items are mobile, they can be intermittently connected to the network. Typically, an application would not be aware, whether a device is connected or not, when a request is placed. Therefore, we need to make sure that requests for data on currently disconnected devices do not cause errors.

**Service mapping.** When new services are deployed to the smart items network, various constraints have to be considered. Therefore, a valid mapping of services to nodes has to be found prior to deployment.

**Service description.** Mapping services to network nodes and accessing services from an external application requires sound description of functionality, interface, and requirements of a service.

## 2.2. Problem statement

Due to the heterogeneity of device types, concepts for abstracting from device specifics are required for a middleware that provides applications with a uniform access to smart items. In this paper, we address the issues of platform-independent service description, service deployment, and service invocation.

## 3. Related Work

An architecture that is similar to ours is being developed within a project called Amigo ([1], [2]). Their middleware aims at enabling ambient intelligence within the networked home environment by seamless integration of heterogeneous service technologies.

They also deal with service deployment using semantic descriptions of services.

The Kairos middleware [3] follows similar goals – abstraction of the node structure to allow macro-programming – but one still needs to program sensors.

In an existing approach, uniform invocation of services on embedded systems is achieved with web service technology [4]. Embedded systems are normally unable to handle the overhead incurred by standard web services. Therefore, light-weight web services are used which reduces the size of SOAP messages. As requests are translated on the embedded system, all web service clients can directly invoke such services without a special middleware. However, this approach is limited to devices which are powerful enough to run a SOAP engine. Thus, it is not applicable to RFID and most sensor node types.

## 4. Proposed solution

In the following, we present the concepts we have developed for service management and invocation in a smart items middleware called (SI)<sup>2</sup> - Smart Items Service Infrastructure. After an architecture overview we explain how services are described, followed by a description of the service mapping component which maps services to nodes to plan their deployment. Finally, we give details on the invocation mechanism for deployed services.

### 4.1 Architecture overview

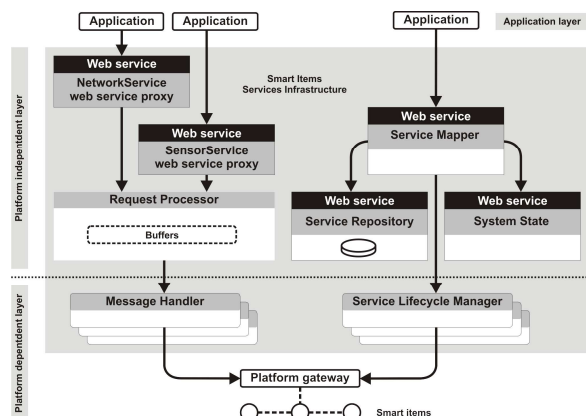


Figure 1: Architecture overview

The architecture of the middleware is divided into two layers, a platform independent and a platform dependent one. The platform dependent layer consists of Message Handlers and Service Lifecycle Managers. Message Handlers convert events generated by smart items of a specific platform into platform independent

events that can be picked by middleware components and applications. On the other hand, Message Handlers convert generic invocations of services running on smart items into the respective platform specific mechanisms. Service Lifecycle Managers are responsible for deploying, starting, and stopping services. The main components of the platform independent layer are the Request Processor, the Service Mapper, and the Service Repository which are presented in the subsequent sections. The architecture is shown in Figure 1.

## 4.2 Service description

Services need to be described both in a human and machine readable way to ensure convenient selection and highest-possible automation during network operation.

**4.2.1. Verbal Service Description.** For an application developer to easily find the services offered by the smart items environment, services are kept in a browsable and searchable service repository. Each service is annotated with a verbal description of its function and keywords.

**4.2.2. Description of the service interface.** Application modelers and/or developers want to access sensor networks as a set of web services. For easy integration, they need a description of the interface that is similar to WSDL, and endpoints where they can invoke services. In CoBIL<sup>III</sup> (Collaborative Business Item Language), we describe the interface of a service consisting of a number of actions that can be invoked along with the parameters of each action in WSDL format. The middleware part of our system uses the interface descriptions to dynamically create application-level web services. Calls to these web services are automatically converted and forwarded to the target nodes and vice versa, the response is converted to a web service message.

Furthermore, the service interface contains a list of notifications (of events) that a service can emit. Applications can subscribe to any event that is defined in a service description. The middleware handles subscriptions and format conversion between the application and the proprietary sensor network messages.

**4.2.3. Service description supporting automatic deployment.** Applications do not want to care about

installing new services on appropriate nodes. Each service should be annotated, so that, at the time a service is needed for the first time, instances of the service can be automatically installed on an appropriate set of nodes. The appropriate set is determined by a number of deployment requirements, the current system state, and the requirements of the applications that access the smart items.

The deployment requirements comprise the following information: Dependency requirements express the need for an instance of another service to be installed on the same node, on a neighboring node, or on any node of the network. Static requirements model the configuration of the node that the service should run on, including core device, power source, and sensors and actuators. Dynamic requirements concern context that is subject to change on a target node like minimum/maximum thresholds of sensor values, remaining system resources like CPU, memory, and network bandwidth. Coverage requirements define the appropriate density of deployment of a service, i.e. on a certain percentage or a certain number of nodes. Quality requirements express that a service can only be installed on nodes where certain other services run with a minimum quality. Quality is defined as a number between 0 and 1 which is assessed and made available by each service itself.

Each of these deployment requirements is assigned a weight  $w$  where  $0 < w \leq 100$  (default = 50) and 1 means the requirement is a mere recommendation, 100 means that the requirement must be met. The weights are used if not all conditions of a deployment can be met. In this case, the requirements with the lowest weight are disposed first. If e.g. the deployment requirements of a service state that it should run on 80% of all nodes ( $w = 100$ ) and that nodes must have a temperature ( $w = 60$ ) and humidity sensor ( $w = 50$ ), but only 60% have both sensors and 20% have only temperature sensor, the service would be deployed on all nodes that have a temperature sensor.

## 4.3 Service mapping

The Service Mapper is the component responsible for deploying services to a network of smart items. When a service is to be deployed, the Mapper gets the service description and the system model from the middleware, and searches for a feasible deployment.

The deployment problem falls under the category of resource constrained scheduling. Execution time does not matter and to reduce complexity we disregard preemption, i.e. services will not be relocated. Communication costs are constant, since our network

---

<sup>III</sup> <http://www.cobil-online.de/cobil/v1.1/>

of smart items has no routing and uses one-hop-communication.

We used the  $\alpha / \beta / \gamma$  notation of Graham et al. [5] to classify the deployment problems since it allows one to easily lookup the complexity ([6], [7]) and find related work. Deploying a non-compound service is classified as  $P / p_j = 1, res\ sor / C_{max}$  and can be solved in  $O(n)$  [6]. The deployment of a compound service is classified as  $P / p_j = 1, res\ sor, intree / C_{max}$ . Its sub-problem  $P / p_j = 1, intree / C_{max}$  can be solved in linear time [6], but there seems to be no related work for the whole problem [7]. That is why we implemented and tested a heuristic to deploy compound and non-compound services.

**4.3.1 Deployment algorithm.** We designed, implemented and evaluated a probabilistic algorithm, which runs centralized as a part of the middleware. It searches the nodes one by one to find nodes which may run the service. After that, the probabilistic algorithm chooses one node based on a given strategy, for instance choose the node with the most battery time left, most processing power left, most memory left.

**4.3.2 Test scenarios.** We tested the Service Mapper on a fixed system model read from an XML file. A basic set containing 21 nodes of three different CPU types was created. Several of these sets were put together to form networks of different sizes with up to 420 nodes.

We used a very complex service consisting of several other atomic and compound services which run either locally or on the nodes neighborhood to test the robustness and scalability of our algorithm. This service and its sub-services always need to be deployed on a certain percentage of nodes on the network (atomicity). This way, the deployment of the services depends on the number of nodes the network has, allowing us to test the same service with a variable amount of nodes.

**4.3.3 Test results.** Figure 2 shows the average values our algorithm needed to create a deployment using the random selection strategy. The algorithm is still able to find feasible solutions in a sensor network with 420 nodes. We also used other services and sensor networks to evaluate the algorithm, but the tendencies remained the same, as shown in Figure 2.

#### 4.4 Service invocation

Once the services are deployed onto the network, their invocation has to be handled. To hide the details

of the different platforms from the client, e.g. a backend application, a layer of indirection has to be introduced. The client calls a web service proxy which is dynamically created for registered services. All proxies send requests to a component called Request Processor which is responsible for buffering messages and finding the correct Message Handler instance.

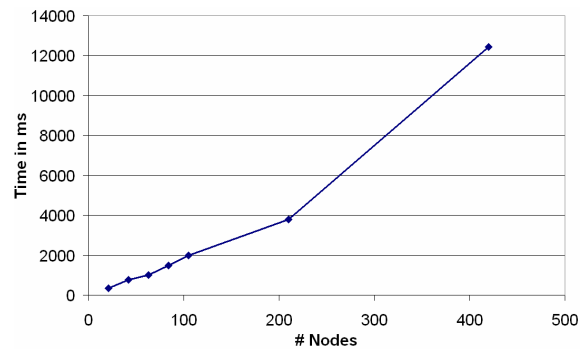


Figure 2: Performance of presented algorithm

**4.4.1 Dynamic proxy generation.** Services running on wireless sensor nodes are represented by web services within the (SI)<sup>2</sup> middleware that act as proxies for invocation. These services can thus be called by any application either running outside or within the Java EE application server. However, there is not a web service proxy for every service instance on every node. This approach would not scale well as there might be thousands of services running on nodes within the wireless sensor network. Instead proxies are generated dynamically only if a new service is inserted into the Service Repository.

After inserting a new node service into the repository, the CoBIL description is parsed. Based on this analysis code for direct and buffered invocation of node services is generated.

If a method of a node service is called using a buffered request, the invocation does not fail in case the node is not connected to the network at that point of time. Instead the call is postponed until the node is available again. The calling application is then notified and can access the return value. The generated methods of a temperature service could look like this:

- *String getTemperature(String nodeID)*
- *String getTemperatureBufferedRequest(String nodeID, String callbackURL)*
- *String getTemperatureResult(String requestID)*

In this example the first method returns the temperature value or fails if the node is not available at invocation time. The second method triggers a buffered

request and returns immediately with a string representing the request ID. After the request has completed successfully, the application is notified using the provided callback URL. The third method can then be used to retrieve the result.

**4.4.2 Lookup and invocation of services.** The web service endpoints for node services are automatically registered with the name service of our middleware and can thus be looked up by applications. The name service also provides information on the web service endpoints of other middleware components.

All invocations of web service proxies are delegated to the Request Processor. Its task is to handle platform-independent requests for a specified service, transmit it to the correct Message Handler instance, and return the respective result. Requests can be placed for a service on a single node or a group of nodes. In the following, we describe the steps that are performed to fulfill these tasks. A request consists of the following elements:

- Service identifier
- Optional: service parameters
- Target identifier (single node or group)
- TTL (time to live)
- Optional: web service endpoint for notification of completion

If a request is placed for a single node, the Request Processor first determines the platform of the node using the Device Registry. This is necessary to select the correct Message Handler type. After that, the connection status of the specified node is checked using the System State component. If the node is offline, the request may be buffered in the Request Processor. In that case, a request ID is returned to the calling application which identifies the request for future access. As soon as the node connects to the network, the correct Message Handler instance has to be found, as there can be multiple platform specific nodes with several message handlers running in the network. The information, to which Message Handler the node is currently connected, can also be obtained from the System State component. Finally, the request is transmitted to the identified Message Handler instance which then translates the platform independent request into a node-specific request and executes it. The result is returned to the Request Processor and either returned to the calling application or stored in the result buffer where it can be retrieved by the calling application using the request ID.

Requests to a group are resolved into single requests and executed separately. The Request Processor keeps

track of the execution status for each individual request that belongs to a group. Once they are all completed, the partial results are assembled into the overall result for the group request.

## 5. Conclusions and Outlook

Integration of smart items into backend applications poses several challenges to a middleware solution. The key issue is to provide uniform access to heterogeneous hardware platforms in terms of service deployment and service invocation. In this paper, we have presented a middleware concept based on an effort to unify the middleware architecture of two EU-funded projects. It provides platform-independent service deployment and service invocation. In the future we will focus on the definition of a data structure for internal representation of requests in a platform-independent manner.

## 6. References

- [1] A. Uribarren, J. Parra, J.P. Uribe, K. Makibar, I. Olalde, and N. Herrasti, "Service Oriented Pervasive Applications Based On Interoperable Middleware", *Proceedings of the 1st International Workshop on Requirements and Solutions for Pervasive Software Infrastructures*, 2006.
- [2] F. Le Mouél, N. Ibrahim, Y. Royon, and S. Frénot, "Semantic Deployment of Services in Pervasive Environments", *Proceedings of the 1st International Workshop on Requirements and Solutions for Pervasive Software Infrastructures*, 2006.
- [3] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming Wireless Sensor Networks using Kairos", *Proceedings of the 2005 International Conference on Distributed Computing in Sensor Networks*, 2005.
- [4] R. van Engelen, "Code Generation Techniques for Developing Light-Weight XML Web Services for Embedded Devices", *Proceedings of the 2004 ACM Symposium on Applied Computing*, ACM Press, New York, 2004, pp. 854-861.
- [5] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey", *Annals of Discrete Mathematics* 5, 1979, pp. 287-326.
- [6] Leung, Joseph Y-T, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, Chapman & Hall/CRC, Boca Raton, 2004.
- [7] P. Brucker, S. Knust, *Complexity results for scheduling problems*. [Online] Available at <http://www.mathematik.uni-osnabrueck.de/research/OR/class/>, [Updated 06-04-2006].