Integrating Sensor Nodes into a Middleware for Ambient Intelligence

Holger Klus, Dirk Niebuhr, Oliver Weiß
University of Kaiserslautern
Gottlieb-Daimler-Str.
67653 Kaiserslautern, Germany
{klus | niebuhr | o_weiss}@informatik.uni-kl.de

ABSTRACT
The development of infrastructures enabling dynamic and automated composition of technical systems is a big challenge. This paper addresses a new idea of allowing service-oriented systems to reconfigure themselves. Therefore, we propose DoAmI - a Domain-specific Ambient Intelligence middleware platform for dynamic integration of services as well as their reconfiguration during runtime. Thereby, one of the features of the platform is that it is capable of interconnecting services based on their availability. In this paper we concentrate on presenting how resource constrained sensor nodes can be integrated into a system using this platform.

1. INTRODUCTION
The vision of Ubiquitous Computing or Ambient Intelligence aims towards supporting people in their daily live at home or at work. In recent years more and more embedded components have become available trying to make this vision come true. Unfortunately these components rarely work together allowing the formation of dynamic systems with emerging capabilities. Building infrastructures for dynamic and automated composition of these components to exchange data and form new systems is a big challenge. For allowing services to work in an ambient manner, these services have to be interconnected. Moreover, Ubiquitous Computing or Ambient Intelligence systems have to be dynamically adaptive since some services are mobile and may appear suddenly during runtime. The dynamic integration of services into a system at runtime therefore is crucial for these kind of systems [1]. For simplicity we will refer to those systems as Dynaptive Systems in the following. This paper addresses a new idea of allowing Dynaptive Systems to reconfigure themselves and introduces an approach for integrating resource constraint sensor nodes.

We propose DoAmI - Domain-specific Ambient Intelligence middleware platform for service reconfiguration and dynamic integration. The heart of the DoAmI platform is a Configuration Service. Moreover we show how a specific sensor node can be integrated in such a self-configuring system.

1.1 Context
The work presented in this paper was (partially) carried out in the BelAmI (Bilateral German-Hungarian Research Collaboration on Ambient Intelligence Systems) project [2], funded by the German Federal Ministry of Education and Research (BMBF), the Fraunhofer-Gesellschaft and the Ministry for Science, Education, Research and Culture of Rhineland-Palatinate (MWWFK). The DoAmI platform is developed to serve as basis for connecting services of the BelAmI project. The application domain of the BelAmI project is assisted living, where a demonstrator for elderly care is built [3]. In this demonstrator multiple embedded sensors collect data within an intelligent home environment. The goal is to aggregate this sensor information to probabilistically gather results about the constitution of elderly people and to support them in various manners.

1.2 Paper outline
The paper is structured as follows: First of all we distinguish our middleware platform from related work. Then we continue with a short description of the application example in chapter 3 which will be used to explain our concepts in the following. Afterwards, we describe in chapter 4 how the DoAmI platform is structured, how DoAmI services are integrated into a system, and how they are configured automatically. In chapter 5 we sketch a concept for how resource constrained sensor nodes can be integrated into the platform and end up with conclusions we derived from this work.

2. RELATED WORK
The core of the DoAmI platform is the Configuration Service, which enables the automatic reconfiguration of a service landscape. In the following we present a selection of technologies, both state of the art and state of the practice that deal with automatic reconfiguration.

2.1 State of the Art
There are several middleware solutions from the research community supporting proactive and reactive system adaptability (cf. [4]). These can be further categorized in component and system adaptation frameworks. DoAmI falls into the category of reactive system adaptation. In other words it supports changing the configuration of a system without

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA ’06 Portland, Oregon USA
Copyright 2006 ACM X-XXXX-XX-X/XX/XX ...$5.00.
(a) the necessity of knowing all possible types of services in advance and (b) without knowing the explicit description of the reconfiguration behavior. In [5] reactive system adaptation is also supported. The C2 [6] architectural style comes into play when one sees architectural connectors as well as components as first class entities. The authors have developed an infrastructure that allows adding and removing components and connectors at runtime by using a special scripting language. Moreover, they support modifying associations between connectors and components. Finally, a special validator component checks whether the modifications are valid. The motivation is different than DoAmI in this case. The authors aim to support manual reconfiguration for runtime system maintenance instead of automated reconfiguration which is offered by DoAmI. In [7] proactive system and component adaptation is described. A special-purpose executable modeling language enables the explicit description of the adaptation behavior of embedded systems. By contrast, DoAmI provides for less expressive and implicit description of adaptation. This is achieved by the different dependency configurations available with each component. The surrogate architecture [8] exists for integration of resource-constrained nodes into Jini networks. The Device Bay approach presented in chapter 5 is based on this surrogate architecture which we applied to our underlying CORBA middleware.

2.2 State of the Practice

Dynamic Loading (e.g. Java Class Loading), Service-oriented Architectures (e.g. Web Services) and Dependency Injection [9] are widespread technologies that support the dynamic integration of services and offer some reconfiguration facilities. Regarding Dynamic Loading and Service Orientation, the main differences between them and DoAmI are (a) the instantiation of types that usually takes place for every imported type and (b) the active service lookup. Dependency Injection features the most similarities to the DoAmI Configuration Service, since the main idea of Dependency Injection is that service requestors are passive. They do not look up services; they only define needed services, which are delivered by the infrastructure. DoAmI works in that way; it analyzes service interdependencies, looks up available services and returns the required references automatically and asynchronously in terms of a callback mechanism. One important difference of DoAmI is the reconfiguration support. DoAmI updates the component wiring in the event of modification of the service landscape (i.e. service appearance, failure, update). Another main difference lies in the multiple configurations of services: A DoAmI service registers a series of different configurations, each one contains provided and required interfaces. In other words there is not only one set of dependencies that must be resolved but an ordered collection of alternative dependency configurations.

In the following we will introduce a small application scenario used in this paper to explain the DoAmI platform and the integration of sensor nodes into it.

3. APPLICATION EXAMPLE

There are many possible scenarios where Dynaptive Systems can be used. This section introduces an example which covers briefly the aspects necessary for the understanding of sensor node integration into DoAmI. We imagine an “intelligent” fridge. This fridge has mainly two features next to cooling its content:

1. *Monitor the temperature:* The fridge is able to monitor, whether its inner temperature exceeds a given threshold. In this case an alert is activated in order to notify the user of a malfunction before the food inside the fridge is getting spoilt.

2. *Monitor the door:* The fridge checks whether the door is open for a longer period than a given time. In this case again an alert is activated.

The thresholds for both features should be configurable by the user. However, each feature can only be provided based on the availability of sensors. The first feature requires a temperature sensor whereas the second feature can only be provided if a door sensor is present. If both sensors are present, the Temperature-Monitor could also be provided in an improved version, which warns only if the door is closed in order to avoid false alerts. The DoAmI platform is expected to decide at runtime, which features can be provided by the system dependent on the concrete available services. In the following we describe the architecture of this platform.

4. PLATFORM ARCHITECTURE

DoAmI is a service-oriented middleware architecture. For the addressed Dynaptive Systems we identified four logical layers. On the top level we have the *Application Layer*, where functional or application services are placed. The interconnection of these services for the exchange of data is done via the *Middleware Layer*. This layer contains administrative services like a *Node Service* or the *Configuration Service* described in Section 4.1. Where the *Configuration Service* deals with service coupling at runtime, the *Node Service* is responsible for lifecycle management of platform and application services on the physical hardware nodes. Additionally the middleware layer provides a so-called *Device Bay* (see chapter 5), which integrates resource-constrained nodes. The bottom two layers (*Communication Layer* and *Physical Layer*) are out of the scope of this paper, since we focus on available standards for their implementation. In the following part we begin with a detailed introduction of DoAmI’s configuration service. Afterwards we describe the service model which is used by the configuration service.

4.1 Configuration Service

One important concept of DoAmI is that all services may offer different configurations. Considering the example introduced before, you can think of the monitoring service, which has three configurations: One requiring a temperature sensor and a door sensor, a second one needing only a

![Figure 1: The Type View of an Example Service](image_url)
temperature sensor, and finally a third one requiring only a door sensor. Each of these configurations is specified by the service interfaces which it provides (e.g. \textit{Fridge Temperature-Warning Interface} or \textit{FridgeDoorWarning Interface}, respectively). A representation of the monitoring service can be seen in Figure 1. The main task of the Configuration Service is to determine, which services can form a Dynaptive System and how they can be interconnected. Therefore, all services have to register at the Configuration Service. The Configuration Service checks for all registered services, whether they contain a runnable configuration and which interfaces this configuration provides. With runnable configuration we mean that all service interface references can be set to service interfaces of other runnable service configurations. According to these runnable service configurations the Configuration Service interconnects services that have requested to run with other services, sets the current configuration, and finally lets the service run. Moreover, it does the same for all services that are interconnected to those services that initially requested to run. In order to enable the Configuration Service to configure the Dynaptive System, all application services have to be compliant with the service model introduced as follows. This is necessary since the Configuration Service has to extract all configuration relevant information from the services at runtime.

### 4.2 Application Service Model

Services within Dynaptive Systems can be seen from two different levels: the instance view and the type view. You can see the service model in Figure 2 containing the instance view on the right hand side whereas the type view is depicted on the left hand side. The Configuration Service uses the type information in order to decide which system configuration has to be established. It then selects concrete instances for this configuration and connects them.

The instance view describes a concrete service implementation. This implementation may support several configurations, which can be selected by the Configuration Service depending on the available service implementations in the specific environment. The supported configurations moreover are ordered enabling the Configuration Service to decide which configuration is the better one if multiple configurations are runnable. The first configuration is considered to be the best one, therefore the Configuration Service will consider \textit{Configuration1} as the best one in our example depicted in Figure 1. Finally when running, a service implementation contains a reference to the currently running configuration enabling the configuration service to decide, whether a newly runnable configuration is better than the active configuration.

The type view describes which types of configurations are supported by a service implementation type and which types of Service Interface References and Service Interfaces are declared, respectively, exported by a service implementation configuration as it had been depicted in Figure 1. This information has to be derived from the instances by reflection at runtime or defined separately in advance. The Configuration Service manages all the type information and uses it for calculating the dynamic configuration of the system.

### 4.3 Configuration Process

As already stated in [1] configuring Dynaptive Systems is a three step process. The following subchapters explain, how each of these steps is enacted by the DoAmI platform.

#### 4.3.1 Discovering correct services regarding their functionality and context independent requirements

The new idea of the DoAmI platform is to turn this around, since services now specify what they need in their different configurations and do not look up other services themselves. They get the required services delivered directly by the Configuration Service, if they are available. They are therefore only passive components within the configuration process.

#### 4.3.2 Selecting the best services depending on the current context

The Configuration Service selects the best runnable configuration for each service that requested to run. This is calculated depending on all services which are known to the Configuration Service. Further context adaptivity is currently under development, since in the near future services should not only be able to specify what kind of service they need, but also which quality of service they require.

#### 4.3.3 Selecting valid service configurations

This is not addressed by the DoAmI platform at the moment. Therefore, a formal specification of service constellations which are allowed or forbidden, is required. Another possibility, which will be addressed in the near future will be the more precise specification of the provided and required interfaces based on the expected behaviour and pre- and
Since we would like to integrate and configure resource constrained sensor nodes as well, we follow the Device Bay approach introduced in the following.

5. INTRODUCTION APPROACH

The Particle Computer device [10] is used to host different services that trigger the available sensing capabilities. Comparable industrial products are for instance the MicaZ from Crossbow [11]. Key features of the Particle device are the wireless ad-hoc communication in self organized networks, the ability to run basic C-programs addressing the sensors and the small size of the hardware board. Furthermore a bridging unit exists which acts as part of the sensor network and forwards the communication to an Ethernet host. One main problem to address is the limited capacity of sensor nodes, for instance the very small memory or the difficult energy supply. Consequently the Particle cannot provide the environment to implement all layers for a full platform integration into DoAmI. The middleware layer can only be adopted in a very basic way, CORBA-references or Java libraries do not exist. This means also the application layer has to be constrained and a specific design for Particles is needed. The solution is an additional component, which supports resource constrained hardware devices and is able to integrate them into the DoAmI platform. This component is called Device Bay and is introduced in the next sections.

5.1 Generic Device Bay

The Device Bay maintains the connection between the Particles and the other DoAmI nodes. That means the services physically located on the sensor nodes are involved through the Device Bay, thus hosting proxies with provided and contained interfaces. The design is based on [12], referencing to a Device Bay as a computer connected to the application network which is able to integrate several homogeneous kinds of hardware nodes into a network. The main task of the Device Bay is to locate new Particles and to set up their service implementations. This means that all necessary information for this process has to be stored on the Particle. This information has to take the dynamic nature of Dynaptive Systems into account, where an arbitrary appearance of different services and even different implementations of the same service may occur. Therefore the type and instance view proposed in 4.2 has to be represented in a form that allows the Device Bay to bridge calls to Particle services and vice versa.

5.2 Device Bay Architecture

The design of the Device Bay is divided into a technical and a logical part. The technical part covers the communication concerns like un-/marshalling method calls into packets used in Particle networks. As the reference implementation of DoAmI uses Java and CORBA technology, synchronous Java method calls have to be translated into an asynchronous manner to be send as packets over the wireless network. This assures that there is no observable difference between a service located on a Particle and any other DoAmI service. In addition there is a component that is responsible for setting up services, that means requesting the information about types and instances of service implementations. The gathered information is used to build an adequate proxy for the service interfaces in order to enable DoAmI’s Configuration Service (see 4.1) to interconnect the appropriate instances.

The logical part reflects the instance service model of DoAmI as depicted on the right side of figure 2. Since the Configuration Service has to determine the available configurations and service users have to call methods in provided interfaces, there have to be basic stub implementations. The chosen solution is using a fixed generic implementation of InstanceServiceImplementationIf and InstanceServiceImplementationConfigurationIf, whereas InstanceServiceInterfaceIf and InstanceServiceInterfaceReferenceIf are specific and must be handled dynamically. As a consequence a Particle service “tells” which interfaces in which configuration are provided/needed and proxies represented by lean stubs can be loaded dynamically.

Consider that the Monitoring Service from figure 1 is placed on a Particle. The information stored on the node about its service implementation must be transmitted to the Configuration Service via the Device Bay including the following: Serviceld (“Monitoring Service”), three ConfigurationIds together with their provided and contained interfaces in the form of Interfacelds.

Figure 3 shows the interconnection of three Particles implementing the Temperature Service, the Door Service and the Monitoring Service. The Particles communicate with each other and with additional external DoAmI Services using the corresponding proxies on the Device Bay. Thereby the Monitoring Service uses the provided interfaces from the Temperature Service and the Door Service while offering its provided interfaces for other services.

5.3 Dynamic Loading of Stubs and Code Generation

Since a developer of Particle services does not want to deal with packet transmission and data (de-)multiplexing, a transparent view of services as proposed in the service model is desired. In the context of this model it is possible to generate the proxy stubs automatically. This means the developer specifies the provided interfaces, for instance as IDL-description and a Java class is generated. This handles all the communication together with the fixed generic part of the implementation and can be loaded dynamically during runtime. In addition it is possible to generate C-skeletons.
for the Particle, too. Thus low-level communication is handled by generic entities on both sides, whereas the developer implements generated function bodies in the Particle code, “ready to run on DoAmI”.

For the Monitoring Service this means e.g. generating the TemperatureSensorInterface-stubs from an IDL-file, which could contain a method like `int getTemperature()`. Together with a ServiceId, ConfigurationId and the Interfaced “TemperaturSensorInterface”, this information suffices to build a Java-class that can be loaded dynamically and act as stub for the Temperature Service on the Device Bay. Furthermore a C-function body will be created for the developer, where only the getTemperature method has to be implemented. Parameters and return values are fixed and enable generic marshalling these values. This way the developer can focus on design of a service without the need to care about technical issues.

Figure 4 presents a sequence diagram with a possible behaviour for the Monitoring Service. The service may periodically check the status of the fridge and hence must get the current temperature value. It uses the `getTemperature()`-method contained in TemperatureSensorInterface by first sending this request to its monServProxy-stub as a packet containing a string, denoted by the method `sendPacket()`. The request is parsed into a method `getTemperature()` invoked at the connected tempServProxy. This stub bridges the call in a similar way to the Particle implementation `tempServ`, which answers with the string “10” in a packet. This string is again parsed into an object from the type Integer representing the value 10 and returned as string to the Monitoring Service, which can process further actions.

6. FUTURE WORK AND CONCLUSION

Due to the Configuration Service, it is very easy to start up a Dynaptive System, since we do not have to specify the concrete system wiring. Nevertheless during the implementation of the Assisted Living demonstrator, some issues remained open. One is the cyclic dependency between services that is needed in some cases. However, allowing cyclic redundancies may cause the whole system to hang up in an endless loop at runtime. Moreover, an Event Service specification is under development within the DoAmI platform. This service should enable all services to communicate not only by remote procedure calls but also using typed events.

In the past we already implemented a similar platform on top of Jini [13], however an integration of C++ services into such a platform is a rather hard task and we wanted to provide a platform enabling a very easy development of application services. As a proof of concept, the DoAmI platform therefore has been implemented using the Orbacus CORBA ORB [14] as underlying middleware for Java services as well as for C++ services. Moreover, we showed the integration of resource constrained nodes by using the Device Bay concept. However, the generation of Particle service bodies from idl-files still has to be implemented for easier use. When implementing services for CORBA, one of the biggest disadvantages is that the business-code for services is normally intertwined with lots of CORBA specific code. The DoAmI platform has implemented an abstraction layer for CORBA that allows the developer of a service to focus just on the business-code. Our middleware platform therefore enables developers to create services Dynaptive Systems very easily.

7. REFERENCES