Dynamic Integration of Heterogeneous Mobile Devices

Christian Bartelt, Thomas Fischer, Dirk Niebuhr,
Andreas Rausch, Franz Seidl, Marcus Trapp
Technische Universität Kaiserslautern
Fachbereich für Informatik
Gottlieb-Daimler-Straße
D-67653 Kaiserslautern
Germany

{bartelt|fischer|niebuhr|rausch|f_seidl|mtrapp}@informatik.uni-kl.de

ABSTRACT

In these days the trend "everything, every time, everywhere" becomes more and more apparent. As consequence of this trend everyone has a lot of small or invisible devices in his direct environment, e.g. mobile phones, PDAs, or music players. Also some network technologies to connect the different devices like (W)LAN or Bluetooth moved mainstream.

Today in most domains the considered devices and technologies are integrated in isolated applications with fixed hardware settings. But humans live in changing environments and have varying requirements, so they need customizable systems which adapt dynamically. As many different types of devices exist, it is a big challenge to integrate them within one system. In this paper we introduce a concept that enables dynamic integration of heterogeneous devices at run time. Although our concept is at an early stage we built a promising implementation in the domain of assisted training to validate the basic principles.

Categories and Subject Descriptors

D.2.10 [**Design**], D.2.11 [**Software Architectures**], D.2.12 [**Interoperability**]

General Terms

Design, Reliability, Experimentation

Keywords

Software architecture, runtime integration, runtime configuration, mobile devices, mobile systems

1. INTRODUCTION

The common progress towards increasingly available, possibly mobile devices exists obviously everywhere. Those devices facilitate work in business and in everyday life. Usually we have to cope with many different types of those devices, which may even realize complex functionalities. For these reasons it is difficult to be able to use all of their specific capabilities because we have to learn how to use these devices efficiently.

DEAS 2005, May 21, 2005, St. Louis, Missouri, USA.

Moreover, users want to use these devices in many different contexts. For instance users want to interact with other surrounding systems via their mobile devices. Hence these devices need a common mechanism for integration into other systems. Thereby, we differentiate between static and dynamic integration. *Static integration* means that the device is integrated at development time, e.g. by hard-wiring it, while integration at run time is called *dynamic integration*.

To sum up, users have to cope each day with more and more complex, possibly *mobile devices*. During usage *dynamic integration and cooperation* of the devices with surrounding systems is required. As these devices are *heterogeneous* particularly pertaining to their usability and integration features nowadays sophisticated users with specific skills are required. In order to let common users benefit from these devices, an intuitive way of usage and integration is needed to shorten the initial phase of skill adaptation.

Ambient Intelligence (AmI) systems try to solve this problem by using a different approach [1]. Their goal is to provide assistance to the users while hiding the complexity. As an AmI system should be nearly invisible to the user, dynamic integration should work without user interaction whenever possible. Furthermore, AmI systems should not only be easy to use, but also support the users.

From an architectural point of view an AmI system is a distributed application, consisting of interacting IT-components. Further IT-components can be integrated into the AmI system at any time and integrated IT-components can be disconnected from the system, respectively they can fail as a result of a defect.

Hence an AmI system is an autonomic system as it has knowledge about itself and its environment, it configures and reconfigures itself, and it protects itself from malfunction while hiding complexity from users [2]. In this paper we elaborate a general architectural approach to support an important feature of autonomic systems: dynamic integration of heterogeneous devices into an overall AmI system.

The remainder of this paper is structured as follows: Section 2 introduces the application scenario which has been implemented in our prototypical AmI system. In Section 3 the logical architecture of the AmI system is described. Section 4 deals with the realization of the logical architecture within a given physical setting. Section 5 presents the general dynamic integration concept required for real autonomic systems. Finally a short conclusion is given.

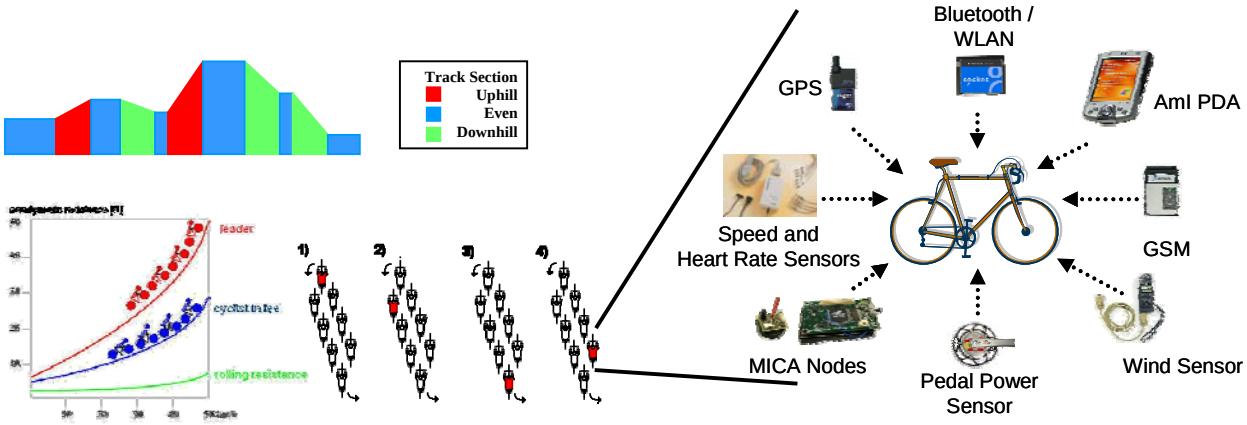

Figure 1: AmI Assisted Bicycle Training

2. AMI APPLICATION SCENARIO

One of the AmI application scenarios we focus on is assisted training [3]. With assisted training we refer to systems that support training of athletes. In this scenario we currently work on a system to support bicycle teams.

The AmI system shall optimize the training effect of a racing bicycle training group. Based on several input factors like the cyclist's individual physical condition, their individual training plan, or the given track profile (cf. Figure 1) the system has to optimize the group training by controlling the group formation, their speed, and the cyclist's positions within the group. Therefore an objective function is required, to solve the training optimization.

This requires the bicycles to form a network enabling the AmI system to gather data from the whole team during the training tour. Therefore, each bicycle is equipped with a more or less comprehensive set of low-power sensors and actors, communication features, and computation nodes as shown in Figure 1. Additionally, dynamic integration is required, as a team may split up during training, sub teams may join again, and the coach may accompany the team in a car. For acceptance reasons the system has to be easily usable and light-weight, nevertheless. Thus, the system has to deal with strongly restricted components (cf. Section 4).

Our current prototype implementation focuses on the dynamic integration of various components dealing with pulse control within a single bicycle (cf. Section 3). The device displaying the pulse sensor data may change: Different cyclists may use different output devices, for instance a watch, a mobile phone, or a headset. Moreover, the coach may join the cyclists in a car and watch a cyclist's pulse rate on his laptop. The presented prototype implementation has to decide dynamically, which of the available devices fits best (cf. Section 5).

Obviously this feature is a general issue. It is not only required for output devices but also for all other types of devices including sensor and actuator devices as well as communication and application nodes. Hence the presented architectural concepts for dynamic integration of heterogeneous devices may be a general approach for the autonomic systems under consideration.

3. LOGICAL ARCHITECTURE

As mentioned in Section 1, AmI systems are distributed applications which integrate heterogeneous and independent IT-components. Hence, the overall logical architecture follows the broker pattern [4]. As shown in Figure 2 the application consists of a set of decoupled and interoperating components using a common communication bus as broker.

To support dynamic integration the logical architecture follows the principle of a service-oriented architecture [5].Each of the services provided by the cooperating components can be assigned to one of the following service categories:

- *Technical services* provide application independent basic middleware functionality, e.g. lookup service for registering and discovering other services or a security service.
- *Functional input services* gather data from the environment and insert it into the system, e.g. a pulse sensor and a pulse range input interface.
- *Functional output services* pass data to the user, like a device for a visual or acoustic output of the pulse rate.
- *Functional application services* use input and output services and realize the application logic on top of input and output services, e.g. a pulse control application warning the user if the pulse rate is out of the predefined range.

In Figure 2 the different service categories are marked by different colours and corresponding icons at the service interfaces. Each service can be further characterized by its *interface type*. Different colors and specific interface names illustrate the different interface types in Figure 2. Our prototype implementation provides five interface types: *PulseSensorInIF*, *MaxMinPulseInIF*, *PulseControlAppIF*, *ShowPulseOutIF* and *LookupTecIF*.

The system can deal with all kinds of *implementations* for the different interface types.. Figure 2 contains two service implementations – a *mobile phone* and a *head set* based implementation – for the interface type ShowPulseOutIF. An additional service implementation is currently leaving the system, another one is joining the system. These two service

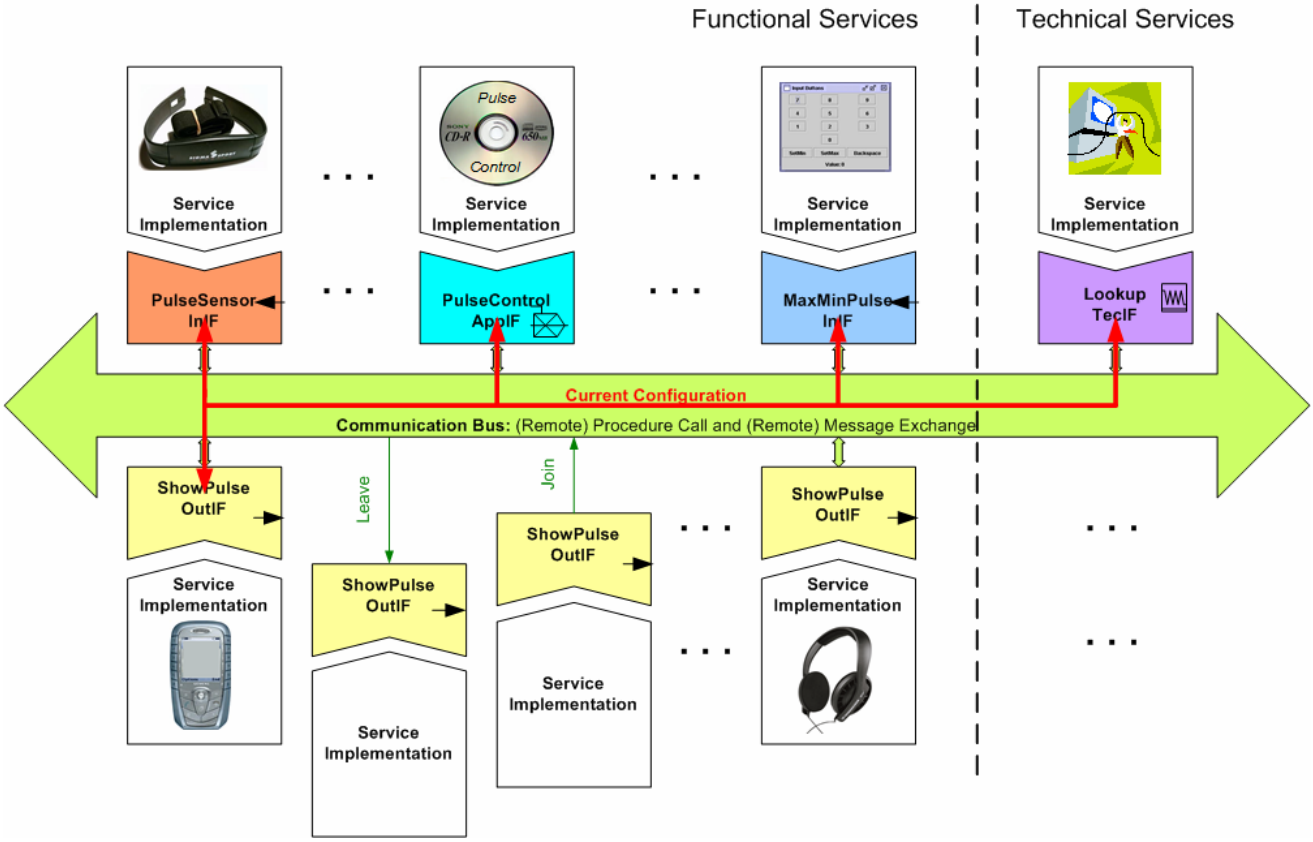

Figure 2: Logical Architecture

implementations are dynamically integrated respectively disintegrated.

Services are not statically integrated at design time. Each service is dynamically integrated into the overall system at run time. Similar to other middleware infrastructures like CORBA [6], EJB [7] and .NET [8] we use a lookup service to manage and maintain all available services. When a service implementation joins the system it registers its interface type and additional descriptive information.

This additional information is required to discover the best fitting service implementation for the current situation. For instance, as illustrated in Figure 2 a visual and an audio pulse output device are available. Descriptive elements may contain general attributes such as “video” and “audio” to describe the different service implementations. Either the application service or a technical trader service [6] select the best configuration with respect to the current situation and especially to the user’s needs. Whenever the available services or the current situation changes the actual established configuration has to be revised.

Each application service has two different modes as shown in Figure 3: discovering and operating. The application service starts in discovering mode. It tries to discover the required services via the lookup service. Once all of these services are registered at the lookup service the application will receive interfaces to these service and switch to operating mode.

As already mentioned the lookup service may provide more than the required services. For instance, there are more service implementations available and the lookup service can not decide which one is the best for the application service. Then, the application will select one service by using some heuristics, e.g. best display resolution or user preference for video instead of audio output (cf. Section 5).

In our prototypical implementation an application in operating mode polls permanently both the pulse sensor and the pulse range input device for new data. The pulse rate and boundaries are evaluated and the result – pulse value plus optional warning – are sent to the output device.

4. PHYSICAL ARCHITECTURE

Once the logical architecture is defined a physical architecture realizing the logical architecture has to be elaborated. As illustrated in Figure 4 a *physical architecture* consists of:

- *Software platform* suitable for providing the technical services and the communication bus of the logical architecture (cf. Figure 2).
- *Hardware components* supporting the selected software platform. Each hardware component serves either as execution host for a set of functional or technical service implementations on top of the software platform or as substructure of the communication protocols.
- *Functional service implementations* corresponding to the functional services of the logical architecture in Figure 2. Each functional service implementation uses

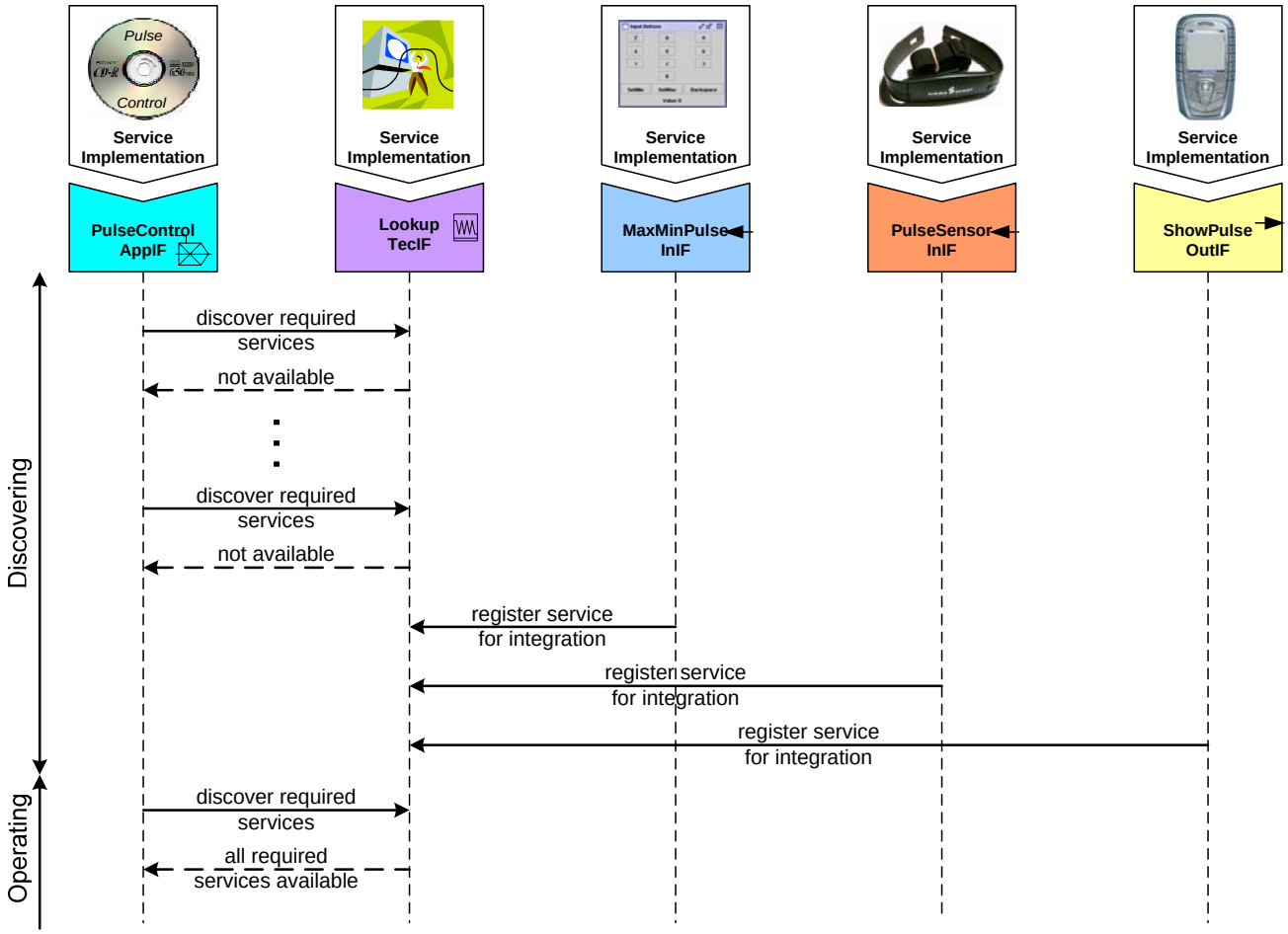

Figure 3: Application Service Modes for Dynamic Integration

the software platform in consideration of the hosting hardware components.

- *Physical configurations* defining concrete and executable systems. A physical configuration is a mapping from a set of required functional services to a set of available implementations.

The methodical approach chosen to derive a physical architecture that corresponds to the logical architecture is outlined in the following paragraphs. Our goal is the implementation of a prototype of our AmI system.

First a proper software platform has to be selected. For the prototype implementation a Java based solution turned out to be the easiest way to provide a small and simple realization of the proposed concepts. Here, the Jini Framework was chosen which is based on Java using the RMI communication mechanism (cf. [9], [10].

The next step is to select the *hardware components* used for the prototype implementation. Most programmable hardware devices like mobile phones, PDAs, and Tablet PCs provide Java support. Hence hardware components supporting our chosen software platform (Jini) are available off-the-shelf.

Then, the *functional service implementations* for the prototype implementation under consideration have to be elaborated taking into account the restrictions of the potential hosting hardware components. For instance, a functional service implementation of an audio pulse output requires audio output capabilities of the hosting hardware component.

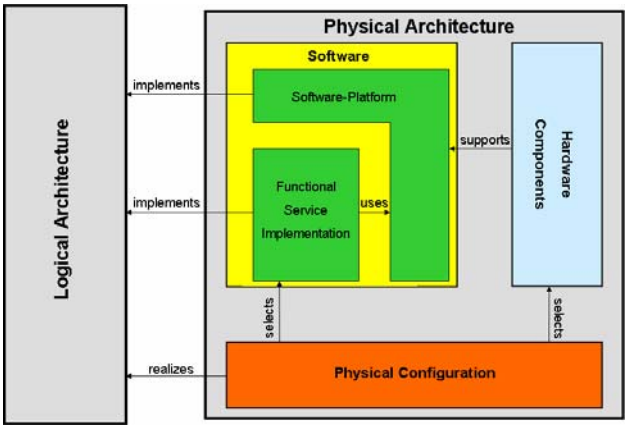

Figure 4: From Logical to Physical Architecture

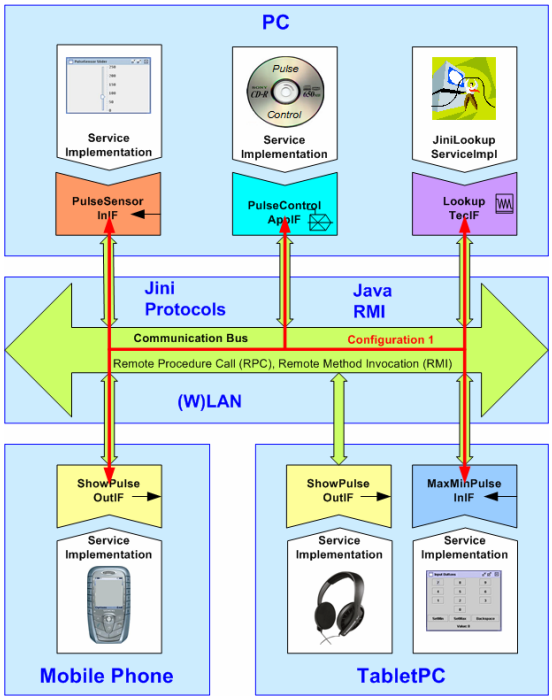

Figure 5: Physical Architecture of the Prototype

The final step is the definition of all possible *physical configurations* realizing the logical architecture. Thereby, the technical restrictions as well as reasonable usage scenarios have to be taken into consideration.

Obviously, these four tasks to derive a physical architecture from a given logical architecture can not be performed sequentially in a fixed order. The results of every task influence the other tasks.

The initial physical architecture for our prototype implementation is illustrated in Figure 5. All required elements of the logical architecture are deployed on one of the shown hardware components – PC, Tablet PC, mobile phone, and (W)LAN.

The architecture shown in Figure 5 contains two possible physical configurations. Configuration 1 uses a visual pulse output service implementation and is indicated by the red line. Another configuration can be defined with an audio pulse output service implementation instead of the visual output (not shown).

As already mentioned in the previous section the pulse control application selects the best fitting service implementations from the available ones considering the actual context.

However, the physical architecture presented in Figure 5 requires that each hardware component executes a Java virtual machine to host the service implementations. Obviously this is not a realistic assumption, as some *restricted hardware components* will neither be able to execute a Java virtual machine nor provide (W)LAN access required for the RMI communication, e.g. a headset or the pulse sensor.

In the presented physical architecture an individual integration of each restricted hardware component is required. This is not an acceptable solution for our main goal of providing a general architectural approach for the dynamic integration of heterogeneous mobile devices.

Note that this restriction cannot be circumvented by selecting a software platform other than Jini. Any software platform imposes requirements on the hosting hardware components. Consequently, there will always be hardware components which do not meet these requirements, especially components that do not incorporate a microprocessor at all, e.g. most available sensor nodes.

To support dynamic integration of those restricted hardware components we have elaborated a concept shown in Figure 6 which we will refer to as *device bay*. The proposed concept is quite similar to the Jini Surrogate Architecture [11], [12], but was newly implemented here.

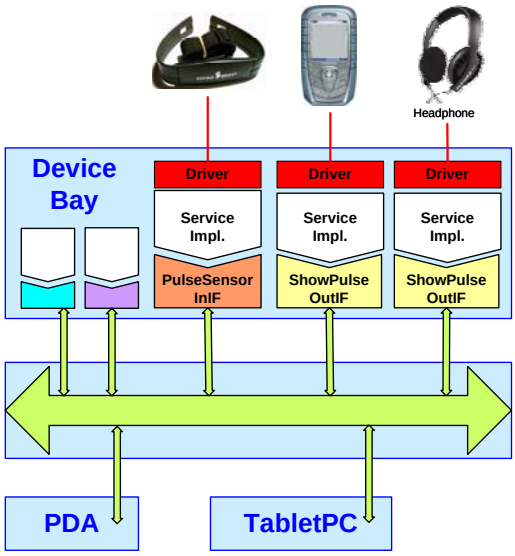

Figure 6: Physical Architecture containing a Device Bay

The device bay is a representative of the restricted hardware components within the Jini federation. Therefore a management component of the device bay looks permanently for new restricted hardware components to integrate. Once a restricted hardware component to integrate has identified itself, the device bay retrieves the corresponding hardware driver from the hardware component or from a driver database and instantiates it. Subsequently the device bay associates the driver with the hardware device and provides it with a service implementation. This service implementation represents the restricted hardware component within the Jini federation.

With the described device bay concept we are able to integrate further devices like a pulse sensor, a headset, a restricted mobile phone or a watch display in a standardized way.

5. DYNAMIC INTEGRATION CONCEPT

In the preceding sections we introduced our concept for dynamic integration of heterogeneous mobile devices using a service-oriented architecture. We described that an application has to select all required services from a pool of available services to fulfill its specified functionality. In addition according to the current usage situation of the application it is important to select not only the required but also the best services. In this section we describe how such a selection process works.

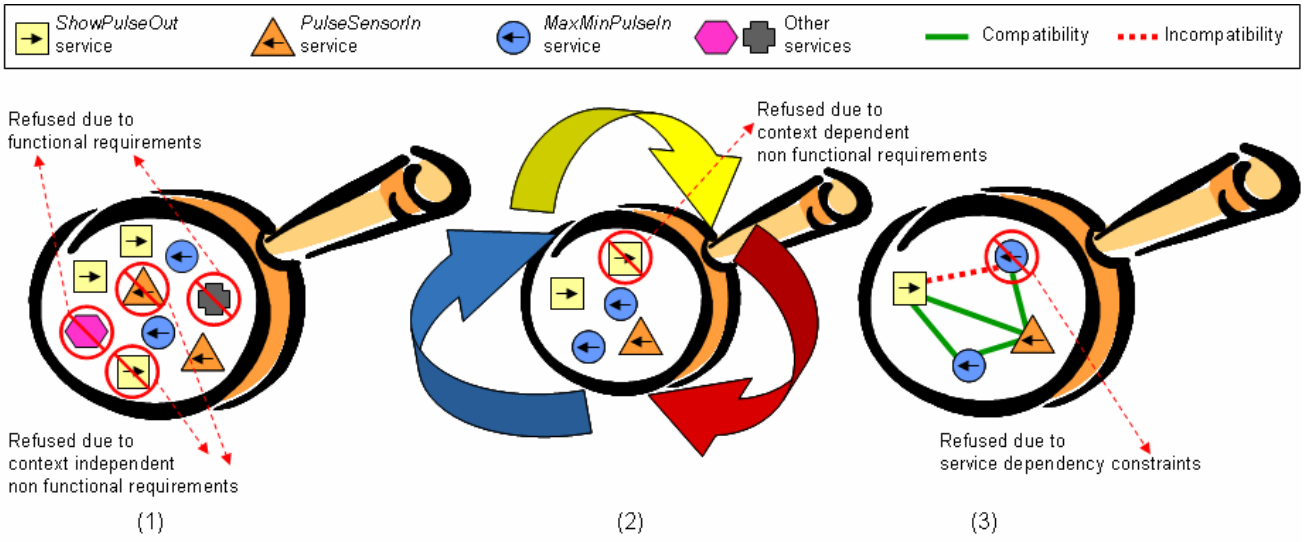

Figure 7: Service Selection Tasks

While searching for services we have to take into account the following three tasks:

(1) Discovering correct services regarding their functionality and context independent requirements.

(2) Selecting the best services depending on the current context.

(3) Selecting valid service configurations.

The first task deals with the identification and selection of all kinds of services required by an application to fulfill its specified functionality. Thereby, it is important to take into account not only the functional requirements regarding a service (its interface and behavior) but also the non-functional requirements, the quality of the service. So even if a service has exactly the interface the application is looking for, it is still possible that it could not be selected due to quality aspects like, for instance, usability, security, safety, or performance aspects.

Figure 7 (1) illustrates an example selection process in the context of our prototype. As mentioned in Section 3, the pulse control application requires three services: *PulseSensorIn*, *ShowPulseOut*, and *MaxMinPulseIn*. Figure 2 illustrated that there is possibly more than one service of each type available. During the application's service discovery, all unidentifiable and thus incompatible services can be refused directly by the lookup service. All of the remaining services fulfill the functional requirements as their interfaces match the needed interfaces for the application. Due to additional, non-functional requirements (minimum screen size and wireless sensors), the remaining *ShowPulseOut* services that represent a very small display and the remaining *PulseSensorIn* service that represents a wired pulse sensor may be refused.

All of these refusals have in common that we can trace them back to decisions made at development time of the application. They are completely independent from the application's runtime context. The services including their interfaces needed by an application can be described at development time as well as the constant quality requirements.

Unfortunately, Jini provides only basic support for performing the first task of our selection process. When a service joins a Jini federation it registers its interface with the lookup service. Each application service uses this interface to discover services that implement it. Thus, incompatible services can be refused easily. However, the mechanism of adding extra descriptive information to the interface while registering with the lookup is limited in comparison to other interface description techniques like the CORBA interface description language (IDL). It is only possible to add so called *Entry* objects to specify the service with more details. Furthermore, an application needs to know not only the interface of the service to be discovered, but also some of its entries to describe it. Jini offers only exact match searches and does not support any kind of fuzzy matching. Thus, further improvement is necessary.

After all services were refused that do not satisfy the constraints described above, there may still be several services of each type remaining. But which specific services should be selected by the application? In order to answer this question we have to take into account the remaining two tasks in our service selection process.

The second task deals with the identification and selection of the best available services during run time of the application according to the current context. Thereby, we also take additional features of the device into account, if available as all services that do not fulfill the functional and non-functional requirements were refused during the first step of the selection process.

The context that influences the service selection consists of several factors. The most important context factor is the user of the application. We have to take into account his preferences (e.g. high resolution video output devices might be preferred), his capabilities (e.g. 150 words per minute can be typed using a standard keyboard), his disablements (e.g. deafness), and his physical characteristics (e.g. gender). Another context factor that influences the service selection process is the environmental setting. This includes the brightness and the noise level of the ambience the application is running in. Another important thing that constitutes the environmental setting is the current state of the application and the whole system. The best service for an

application in normal state may be a bad selection for the application in maintenance or emergency state. Unfortunately, we do not always have the opportunity to select the best services for the current context, as these services are not always available. Thus, we have to select the best service available.

Figure 7 (2) illustrates the second selection step in the context of our prototype. Generally users of our system have to pay attention on the street and other cyclists. To guarantee adequate supply of all information needed to optimize their training, nevertheless, the remaining audio output device (Bluetooth headset) will be selected instead of an available visual output device (PDA).

In contrast to the first, context independent task of our selection process, the selection can not be decided at development time, and thus has to be decided at runtime when all context factors are known. This implies a service selection process that never ends as long as any context factor can change. Thus, over time the set of selected services may change many times depending on the changing context of the application.

Jini provides some basic support for the first service selection task, it does not provide any support for the second task. Further improvement is also necessary.

The third task of our service selection process deals with dependencies between services. Unfortunately, not all possible combinations of services are selectable due to incompatibility reasons. Thus, for the first time we cannot refuse or accept services by analyzing them separately.

There are several kinds of dependencies between services that can lead to a refusal of a specific service or even a whole set of services. First of all, it is possible that some services cannot be used at the same time, even if they belong to different types of services. For example, if there are two wireless devices that are connected via the same infra-red interface to a system, only one device can be selected, because the infra-red interface does not support multiple connections at the same time. There are a lot more dependencies between services that need to be considered.

Figure 7 (3) illustrates the third selection step in the context of our demonstrator. We only have to decide which *MaxMinPulseIn* service should be selected (wireless or wired keypad). As the receiving unit of the Bluetooth headset and the receiving unit of the wireless keypad use the same wired connection to the system, they cannot be used at the same time. According to the selection of the wireless headset during the second task of our service selection process the wireless keypad has to be refused.

Again, Jini does not provide any support for this task. It is very important to mention that the three tasks of our service selection process are not perfectly independent from each other. Especially task two and three have many interrelations. Definitely, task one should be taken into account first to avoid further investigations of services that do not fulfil the requirements.

If all service selection tasks are performed and there are still several services of the same type, even after several iterations of the second and third task, a mechanism is needed that decides which services should be selected. In this case external aspects should be taken into account as well as user interaction. Some of these aspects are non-functional requirements like consistency of usage and design over different systems, general usability guidelines or performance aspect that can not be related to the current application directly.

6. CONCLUSION

According to the exemplary scenario we motivated, that dynamic integration is a basically aspect of Ambient Intelligence systems. Referring to this we derived a logical architecture supporting dynamic integration and showed a methodology how to get a physical architecture from it. Based on the explained scenario we were able to derive a general concept for dynamic integration and described an implementation for our assisted training scenario. Many aspects were discovered where further work can be settled. First of all, Jini provides only basic support for the concepts regarding dynamic integration as introduced in Section 5. Therefore a dynamic integration platform for better support of these concepts is required. A better customization of the AmI-system to the environment is one aspect for optimization. So, hardware constraints like computing power are currently not taken into consideration when selecting a physical configuration. This should also be integrated into a platform. Another aspect is the consideration of safety and security in the system. When building safety critical systems, verification of correctness is a very important point. In order to enable dynamic integration for these systems, a concept for verification at runtime has to be integrated. These are only two aspects which have to be incorporated into the prototype. The result would be a complete system that can be used to evaluate the new concepts.

ACKNOWLEDGEMENTS

This work was supported in parts by the BelAmI project of Fraunhofer IESE - which in turn is funded by the German Federal Ministry of Education and Research (BMBF), the State of Rhineland-Palatinate's Ministry of Science (MWFFK), and the Fraunhofer Gesellschaft e.V.

REFERENCES

[1] Remagnino, P., Foresti, G. L.: *Ambient Intelligence: A New Multidisciplinary Paradigm*. IEEE Transactions on Systems, Man, and Cybernetics – Part A: SYStems and Humans, Vol 35, No. 1, January 2005

[2] Murch, R.: *Autonomic Computing*. Prentice Hall PTR, 2004.

[3] Schürmann, B.: *Wireless Assisted Training*. http://www.eit.uni-kl.de/AmI/en/inhalte.html.

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[5] Krafzig, D., Banke, K., Slama, D.: *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall PTR, 2004.

[6] OMG: *CORBA 3.0 – OMG Specification*. http://www.omg.org/technology/documents/corba_spec_catalog.htm

[7] Sun Microsystems Inc.: *EJB. Enterprise Java Beans Specification, Version 2.1*. http://java.sun.com/products/ejb/index.jsp.

[8] Brox, D.: *Essential .NET*. Addison-Wesley, 2002.

[9] Sun Microsystems Inc.: *Jini Network Technology Homepage.* http://www.sun.com/software/jini/.

[10] Sun Microsystems Inc.: *Jini Community Site*. http://www.jini.org.

[11] Sun Microsystems Inc.: *Jini Surrogate Project Homepage*. http://surrogate.jini.org/.

[12] Sun Microsystems Inc.: *Jini Technology Surrogate Architecture Specification, v1.0 Standard.* http://surrogate.jini.org/sa.pdf