A Service-Oriented Operating System and an Application Development Infrastructure for Distributed Embedded Systems

Martin Lipphardt[1], Nils Glombitza[1], Jana Neumann[2], Christian Werner[1], and Stefan Fischer[1]

1 **Institute of Telematics, University of Lübeck, Germany**
2 **Institute of Information Systems, University of Lübeck, Germany**

Abstract

The paradigm of service-orientation promises a significant ease of use in creating and managing distributed software systems. A very important aspect here is that also application domain experts and stakeholders, who are not necessarily skilled in computer programming, get a chance to create, analyze, and adapt distributed applications. However, up to now, service-oriented architectures have been mainly discussed in the context of complex business applications. In this paper we will investigate how to transfer the benefits of a service-oriented architecture into the field of embedded systems, so that this technology gets accessible to a much wider range of users. As an example, we will demonstrate this scheme for sensor network applications. In order to address the problem of limited device resources we will introduce a minimal operating system for such devices. It organizes all pieces of code running on a sensor node in a service-oriented fashion and also features the relocation of code to a different node at runtime. We will demonstrate that it is possible to design a sensor network application from a set of already existing services in a highly modular way by employing already existing technologies and standards.

Digital Object Identifier 10.4230/OASIcs.KiVS.2011.26

1 Introduction

Pervasive computing can be found in a large variety of application scenarios. Hence, many different kinds of pervasive systems are realized by different classes of devices. Wireless sensor networks (WSN) represent one particular pervasive scenario, which is characterarized by the usage of resource constraint devices. A lot of effort was put into technical and algorithmical problems in WSN, yet implementing a WSN application is still a tedious task. Many parameters have to be considered like network density, traffic patterns and mobility models. Additionally, possible environmental influences must be preconceived. Therefore, besides profound programming abilities a lot of expert knowledge and understanding is needed to design a functioning and robust sensor network application. This holds especially if we consider the fact that actual users of sensor network technology are not usually skilled in the field of computer science. Application scenarios for WSNs can be found in many different fields stretching from military, science, logistics to health care, where each application has its specific requirements and characteristics.

As state of the art for WSN application development we observe a development cycle that can be subsumed to the graph given in Figure 1. A potential user of WSN technology describes the problems with requirements, e.g., monitoring phenomena or tracking objects. The user drafts the requirements and presents them to a WSN expert. Based on the requirements the WSN expert creates a model of the application scenario. The size and density of the network is determined. Mobility models are designed and the data rates are estimated. With consideration of environmental influences the scenario is evaluated in a simulator. Based on the results, the WSN expert chooses different protocols and algorithms and finally implements the WSN application. After the successful deployment of the sensor network, the user receives data from the network and draws conclusions which may bring up new aspects that might affect the requirements to the WSN. Again the user presents his new requirements to the WSN expert.

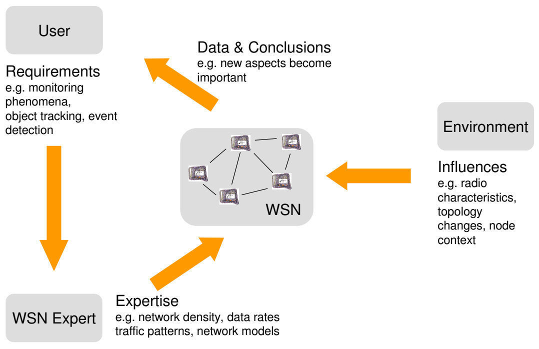
User
Data & Conclusions
e.g. new aspects become important
Requirements
e.g. monitoring phenomena, object tracking, event detection
Environment
WSN
Influences
e.g. radio characteristics, topology changes, node context
Expertise
e.g. network density, data rates traffic patterns, network models
WSN Expert

Figure 1 Current WSN application development cycle

Observing this situation we see parallels to the early days of software development in the late 1950s. In these days, a lot of expertise was needed to deploy even programs solving simple tasks. As a consequence the usage of information technology was reserved for the programmer himself. Since then a lot of effort was put in the development of new programming paradigms to enable actual target user groups to get access to new information technologies. Up to now, the main focus of sensor network research is on mastering resource constraints and achieving robustness in WSNs. Having overcome a lot of technical barriers, we have the possibility to make the wireless sensor network techology accessible for users. Our goal is to make the development of a distributed sensor network application easier allowing non-WSN experts to deploy a sensor network. With regard to the distributed nature of WSN applications and the complexity of interactions between network topology and environmental influences, we consider two aspects to be crucial for enabling users without WSN expertise to access WSN technology: abstractions of application development and runtime modification of deployed WSN applications.

An abstraction for application development makes the complexity of the sensor network application more transparent to the user. It must enable the user to develop different applications easily and in a minimum of time. Since a user with no WSN expertise cannot predict the behavior of the WSN or the WSN application, there might be a lot of trial and error during application development and deployment. Enabling the sensor nodes to change their software during runtime can ease the process towards a successfull deployment and is therefore inevitable.

In this work we introduce *Surfer OS*, a service-oriented minimal operating system for embedded devices that allows an integration and replacement of services during runtime. Additionally we present an infrastructure that offers a *Service Repository* and allows graph based service composition and runtime adaptation of WSN applications for *Surfer OS*.

This work is structured as follows: The next section will give an overview over current WSN operating systems, middlewares and frameworks with the focus on the development of distributed WSN applications. In Section 3 we will present our basic idea for WSN application design and introduce the interactions among the components of our infrastructure, namely *Surfer OS*, the *Service Repository* and the *Service Composition*. Section 4 gives a detailed description of *Surfer OS* and its implementation. Section 5 describes the *Service Repository* and the data stored with each service. In Section 6 we give details about the application development for *Surfer OS* with a graphical tool support. Furthermore, we show how the process of *Service Composition* can be integrated into enterprise applications as well as business processes. In Section 7 we discuss the benefits and downsides of our presented operating system and infrastructure. We conclude the paper with Section 8 summarizing our contribution and giving an outlook on future work.

2 Related Work

The most widespread operating system used for sensor networks is TinyOS [6, 7]. Using a component based programming model, TinyOS is more than an operating system. It is a framework with a set of components that allows for building an application specific operating system for embedded devices. Each component resembles a functionality and exposes one or more interfaces. The fine grained component-based architecture allows the composition of applications before the deployment of the sensor network. This makes the application development for sensor networks more transparent.

An extension to TinyOS which allows modifications of the WSN application during runtime is FlexCup [12]. FlexCup is able to dynamically exchange and link TinyOS components on the sensor nodes for adapting applications to new needs. The five steps of the FlexCup update process are storage of code and meta-data, symbol table merge, relocation table replacement, reference patching, and installation and reboot of the application.

Dunkels et al. developed the Contiki [4] operating system for sensor nodes. With Contiki it is possible to load and unload individual applications or services at runtime [3].

Mantis [1] is a WSN operating system that offers a comprehensive set of system application programming interfaces (API) for I/O and system interaction. Providing this API Mantis hides the complexity of scheduling and concurrent access to ressources claiming a shallow learning curve for programmers with C knowledge. Mantis also allows binary code updates on sensor nodes during the runtime of applications. Based on calls to a system call library provided by the Mantis system kernel applications are updated and written to the EEPROM. By resetting the node the software update process is completed and the new application is executed.

Han et al. introduce SOS [5] a dynamic operating system for sensor nodes. SOS allows the dynamic interaction of independent software modules. SOS uses position independent code to achieve relocation and jump tables for application programs to access the operating system kernel. Application programs can register function pointers at the operating system for performing inter-process communication.

With Maté [8], a virtual machine is available for TinyOS. Maté abstracts from the operating system allowing only a specific set of instructions. A module written with these instructions can be easily integrated into a running application allowing a (re-)configuration and adaptation of the application.

SensorWare [2] is a framework that uses Tcl scripts as abstraction for implementing distributed applications. The scripts can be send into the WSN during runtime. They can migrate within the network and replicate themselves.

The programming framework EnviroSuite [10] by Luo et al. proposes a new programming paradigm called environmentally immersive programming. EinviroSuite offers languages primitives that transparently map onto a library of algorithms for sensor networks allowing the programmer to think diretcly in terms of environmental abstractions.

OASiS [11], a programming framework for service-oriented sensor networks, applies the paradigm of service-orientation on top of TinyOS. In OASiS each activity is implemented as seperate service. The goal is to simplify programming by providing an abstraction, where the applications behavior is described as modular dataflow blocks between services in a specific service graph.

3 Operating System and Infrastructure

The related work shows that supporting the application development and providing an intuitive network abstraction to the application developer is an important issue. We observe that on different levels from the basal operating system to high level frameworks much work is spent to make the application development easier. Each of these approaches has its strengths but none fulfills our needs to enable users without expertise to create and deploy a WSN application sufficiently.

On operating system level, a lot of understanding of the distributed nature of sensor networks and their constraints is needed. The component based approach of TinyOS allows a composition of the application, but still the right protocols and algorithms for the deployment must be known upfront. Approaches which are based on loadable binary code modules enable the user to change the application during runtime. But often these approaches such as [12] and [4] exploit specific platform characteristics or necessitate a restart of the reprogrammed nodes what can be critical for keeping a consistent network state and is an additional aggravation of the development process. Overall, an operating system alone does not provide a sufficient abstraction for the application developer.

Middlewares and frameworks offer a more intuitive abstraction for the programmer. Using a virtual machine like Maté allows for a platform independent and smooth integration of modules into a running application since no restart of the application is needed. But the programmer is bounded to a fixed subset of instructions offered by the virtual machine. Additionally the programmer has to learn the specific syntax for the virtual machine. Furthermore, such approaches are always tailored to a specific type of application constraining the user to a specifc scenario. Frameworks as well as middlewares already offer a specific set of algorithms and protocols. The developer has no chance to modify or optimize this layer to his needs.

3.1 Overview

In our approach, we want to offer the application developer as much flexibility as possible by providing an easy to use application development support at the same time. The fundament is the service-oriented paradigm, which is commonly used in the field of distributed applications. In contrast to other service-oriented approaches for WSNs we consider all functionalities on a node down to the hardware interfaces as services. Where hardware interfaces are always present on the nodes (non-migratable), all other services need to migrate into the WSN and onto the nodes. We deploy the nodes solely with the non-migrateable services. This implies that neither application logic nor any protocols are present on the node right after deployment. On demand, services representing a routing protocol or a localization algorithm

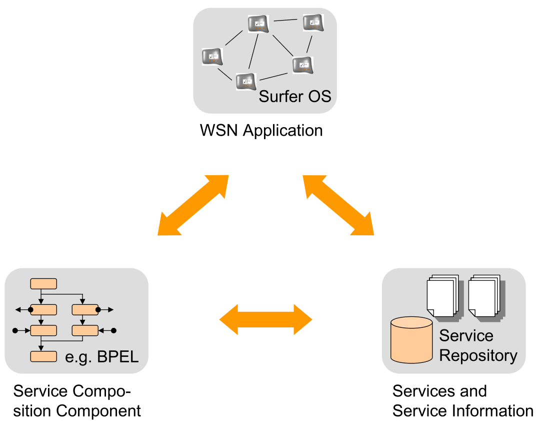
Surfer OS
WSN Application
e.g. BPEL
Service Composition Component
Service Repository
Services and Service Information

Figure 2 Components of the infrastructure for WSN application development

or part of the application logic migrate successively into the WSN composing themselves to a WSN application.

In order to realize this migration of the services and support an intuitive development of an application, the presented infrastructure consists of three components shown in Figure 2: a minimal operating system called *Surfer OS*, a *Service Repository* and an interface for *Service Compositon.*

The *Service Composition* interface allows for an abstraction for composing services and initiates the migration. The services themselves are stored in the *Service Repository* from where they migrate into the WSN. The operating system on the nodes integrates the services into the running application. In the following we will describe the requirements for each component.

3.2 Operating System

For a successful sensor network deployment, the usage of adequate protocols and algorithms for a specific scenario is crucial. Being not able to predetermine radio characteristic and all network parameters such as data traffic patterns, topology changes or node mobility, the user of WSN technology must be able to exchange all protocols or functionality from the application logic down to the communication layer. In terms of the service-oriented paradigm all functionalities offered by protocols and algorithms are encapsuled into different services. The operating system must provide the possibility to add, remove or exchange services on the nodes during the runtime of the deployed sensor network. As a consequence, right after the deployment the nodes basically offer only two kinds of functionalities: (1) access to the available hardware resources on the node via service calls and (2) management of services.

The access to the hardware resources is realized as very basic service calls. The operating system must publish the access point to the radio interface, the sensors and other hardware resources as services, since such functionality demands for special hardware I/O like, e.g., the reading and writing on special registers. Higher level services such as a radio stack are present as services providing a predetermined interface to other services but the implementation on the node does not offer any higher functionality yet. The service that encapsules the radio stack for sending a packet via the radio is not yet providing any routing layer. This service in its initial version on the node directly accesses the hardware interface sending the “plain” packet.

Managing the services means allowing the integration, removal and replacement of services. The operating system must react on special service packets which are directly processed. When adding the services, the provided functionality must be published on the nodes by the operating system. The functionality must be made accessible in an intuitive way, e.g., via a service and a function name. Every service contains a service type and a version number. The service type is a numerical representation of the functionality provided by the service, e.g., “routing”. The version number distinguishes between different implementations of the service. Based on the type of a service and the version the operating system determines whether to add the provided service, to replace a present service or to ignore the received service. If a service of a specific type is already present, it is replaced if a different version of this service type is received. Additionally, dependencies among services must be preserved by the operating system in case of replacement of services. To realize these demands, we implemented the operating system *Surfer OS* which is described in Section 4.

3.3 Service Repository

In order to compose WSN applications, the user of WSN technology needs a collection of different services providing different functionalitites. Programmers with WSN expertise develop a lot of different algorithms and protocols suited for different scenarios and applications. If these algorithms and protocols are implemented as services for *Surfer OS*, they can be applied by a WSN user in his application. Therefore, we need a place where these services can be uploaded, stored and accessed by a component which manages the service migration. For this task we implemented a so called *Service Repository*. It acts as a container for migrateable services where services can be up- and downloaded or managed by the service programmers via an easy to use user interface.

In order to enable the service programmer as well as the service user to use the *Service Repository* it must be accessable via an intuitive and easy to use user interface. This interface has to provide the above mentioned functionalities namely: displaying all available services including important service information, providing the possibility to insert new services and upload the corresponding service code, and providing the possibility to change the service information as well as the service code itself.

3.4 Service Composition

The third component of our approach addresses the process of *Service Composition*. This component enables abstraction for the user of WSN technology for creating sensor network applications by composing a set of services.

One important characteristic of a service-oriented approach for distributed applications is its flexibility. Thus, the requirements to the design of the *Service Composition* component are very versatile. If we consider how services can be composed, we identify two scenarios: Services can be composed by a *Human Composer* or by a software performing the composition to which we refer to as *Automatic Composer*.

The *Human Composer* is the developer of an application who is able to deploy a WSN application even without expertise in this field. By selecting a set of services out of a service repository, he can compose an application tailored to the demands of his individual application scenario ad-hoc during runtime.

The composition of services can also be performed automatically by a software, the *Automatic Composer*. Today sensor networks become more and more part of enterprise IT systems or even flexible parts of business processes. Therefore, it must also be possible that such systems can control the composition of services for the sensor network automatically.

The realization of the *Service Composition* component has to meet the requirements of both scenarios: the *Human Composer* and the software based composition. Therefore, it must provide an intuitive interface to the *Human Composer* and a standardized interface for the composition of services by an external software.

4 Surfer OS

4.1 Concept and Architecture

As described in Section 3.2, the main task of our operating system *Surfer OS* is to provide access to available hardware resources and to manage services, which compose the complete WSN application including even lower layers such as the communication stack. The management of services includes addition, removal and replacement of services. At the moment of deployment of the sensor network, the nodes do not provide any further functionality.

The central unit within *Surfer OS* that organizes the services is the *ServiceManager*. The *ServiceManager* keeps a dynamic symbol table that maps a tuple consisting of a service type and a function name onto a function pointer. This allows an intuitive usage of services by accessing them via the service type and the function name. Initially, the *ServiceManager* keeps the non-migratable services (dark-colored in Figure 3) representing the hardware interfaces in the symbol table. This includes the access to the sensors, the radio, the RS232, the memory, as well as scheduling as shown in Figure 3. Additionally, basal service implementations representing the radio stack and the RS232 stack are provided using the specific hardware interfaces.

Surfer OS identifies special service packets which are then processed by the *ServiceManager*. With these packets a service can migrate onto a node or the command for erasing a service on a node can be transmitted. If such a migratable service (light-colored in Figure 3) is received, the *ServiceManager* checks the service type and the version number. If a service of this type is not present, the new service is included into the application. The new functionalities are added under the service type in the symbol table of the *ServiceManager* such as the beacon service in Figure 3. The *ServiceManager* offers an interface where services themselves can access other services and the hardware via the services already registered in the symbol table. If a service of the same type but with a different version number is already present, the present service is removed and the new service is added. In this way even non-migratebale services can be replaced as shown in Figure 3. In order to preserve dependencies such as registrations for callback, the registration is provided by a subscription service. This service keeps a table with service type and callback addresses. If a service is replaced, the previously registered callbacks for this service type can be retrieved by the new service. This allows a seamless replacement of services that are used by other services. Thus, we can even exchange, e.g., the services representing the communication stack enabling us to exchange routing protocols during runtime.

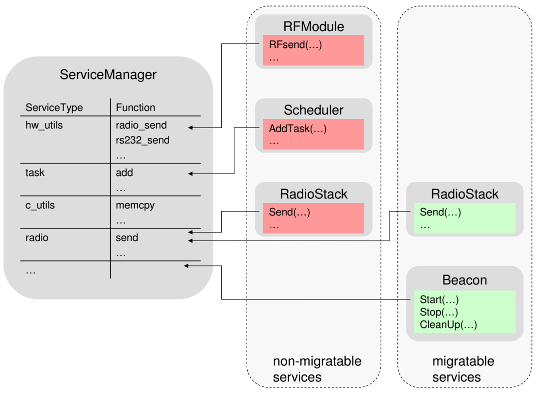

Figure 3 The *ServiceManager* module with the dynamic symbol table

4.2 Implementation

We implemented *Surfer OS* on the *pacemate* sensor node platform [9]. The *pacemate* platform uses the Phillips LPC2136 microcontroller with 256 kbyte Flash memory and 32 kbyte RAM. The microcontroller realizes a 32 Bit ARM architecture. The LPC2136 provides in-application programming allowing for erasing and/or programming the Flash while the application is running. The complete Flash or a single Flash sector can be erased in 400 ms. Writing 256 bytes to the Flash takes 1 ms. Program code is executed from the Flash but can also be executed directly in the RAM.

The *Surfer OS* is implemented in C. The *ServiceManager* keeps the addresses of functions as function pointers. The services are compiled independently. For the migration of the services onto a node we have to solve the problem of code relocation. We use a scheme that provides a solution for the general problem of code relocation. This approach exploits the partition of the ELF format into .TEXT, .DATA and .BSS segments that can be used on every hardware platform that uses the ELF file format. After a service is received completely and its code is relocated successfully, the *ServiceManager* accesses the new service via a predefined interface.

The services follow a specific service template. Every service implements at least four functions: ServiceInit(...), ServiceStart(), ServiceStop() and ServiceCleanUp(). The Service-Init(...)-function performs the registration process of the service. The ServiceStart() and ServiceStop() start or stop the actions of the service. The ServiceCleanUp() is executed when the service is removed. The programmer of a service has to make sure that all registrations and callbacks are removed in this function.

The ServiceInit(...)-function is invoked by the *ServiceManager* after the service is received. The linker file ensures that this function is located at the very beginning of the programm code in the .TEXT segment enabling the *ServiceManager* to find the entry point to this function. A predefined signature allows the *ServiceManager* to pass arguments into the ServiceInit(...)-function. In this way the function pointers to the interfaces of the *ServiceManager* are given to the service. The service uses these interfaces to retrieve available functionality from other services and register its own provided functions. If the registration of functions is completed, the ServiceInit(...) returns the status to the *ServiceManager*. If the registration was successful, the process is completed. The *ServiceManager* executes the ServiceStart()-function that is now available in the symbol table. If the registration of functions failed, the *ServiceManager* would remove the service from the symbol table and would free the allocated memory erasing the service.

Since received services are written to the Flash memory, they are persistent even if the node is turned off. If the node is rebooted, the present services can be integrated and restarted. In this way, the software configuration of a node can be preserved, reused and even analyzed in case of failures.

5 Service Repository

In Section 3.3 we introduced the *Service Repository* as a collection of services which are provided by service programmers with WSN expertise and can be applied by the WSN user. To meet the described requirements, we identify three main modules that compose the *Service Repository*: a graphical *User Interface*, a *Repository Service* and a *Machine Code Modification Service*.

The *User Interface* lists available services and visualizes complete information of services such as service type, version, size and author. For the service programmer, it provides input masks to upload implemented services. The user interface additionally offers the possibility to change the service information in the database as well as in the stored service code. Using the migration scheme described in Section 4.2 the services themselves are stored as machine code for the target sensor node platform which is in our case the pacemate platform. We realized the user interface as Web-based application.

The connection to the database is established by the *Repository Service*. This service – realized as Web Service – encapsulates the upload of services into the database and makes the stored data available. More precisely, it offers interfaces to store, delete, update and display the content of the database.

The functionality to allow modifications of the machine code in order to update service type and version is provided by the *Machine Code Modification Service*. If the user changes this information via the input mask provided by the user interface, this modification service writes the new information back into the machine code. This avoids unnecessary recompilations of the service code and guarantees consistency between the database entry and the according service code.

6 Service Composition

The aim of our contribution is to make WSN technology available for users without any WSN expertise. In Section 3.4 we described the requirements for the *Service Composition* component. We identified two scenarios: either the *Human Composer* or the *Automatic Composer* is controlling the composition process. We demanded that the composition component of our presented infrastructure has to meet the requirements of both scenarios. In order to fulfill these requirements, we have to divide the composition component into two modules which are designed following the service-oriented architecture programming paradigm. We need one module which controls the actual migration process (*Migration Control*). As second module, we need the *Migration Infrastructure* consisting of the *Repository Service* and the *Migration Service* (see Figure 4).

The *Migration Infrastructure* is the basis of the *Service Composition* component. While the *Repository Service* (cf. Section 5) enables the access to a set of *Surfer OS* services, the *Migration Service* realizes the interface to the sensor network. The controlling module of the migration process retrieves services from the *Repository Service*. Calling the *Migration Service* pushes the services into the WSN and completes the migration process outside the sensor network. Analog to the functionality of pushing *Surfer OS* services into the sensor network, the *Migration Service* provides the removal of services from sensor nodes. As the pushing process, the removal process is controlled by controlling modules as well.

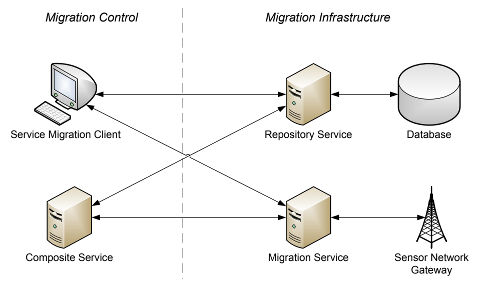

Figure 4 Architecture of the implementation of the *Service Composition* component

The *Migration Control* manages the migration process and offers the abstraction for the user for the WSN application development. As part of our infrastructure we provide a graphical user interface application allowing the *Human Composer* to select services of a repository and to transfer them into the sensor network. In order to embed the service migration process into enterprise IT systems using the *Automatic Composer*, such systems have to provide the capability of controlling the migration. Based on the *Repository Service* and the *Migration Service* and the fact that we use Web Services as communication backbone, we are able to build other high level services on top using classical programming languages like Java or C++ as well as business process modeling languages such as BPEL (Business Process Execution Language, [13]). Especially embedding the migration process into business processes, which are designed with a graphical tool support, enables non IT personnel to easily compose *Surfer OS* services.

7 Discussion

7.1 Benefits

Having implemented all three components, the *Surfer OS*, the *Service Repository* and the interface for *Service Composition*, we now have an infrastructure for quick and easy WSN application development. Opposed the development cycle for WSN applications previously shown in Figure 1, our approach realizes an application development process shown in Figure 5.

The service-oriented approach and thus the loosely coupled services provide an abstraction that is commonly used for distributed applications.

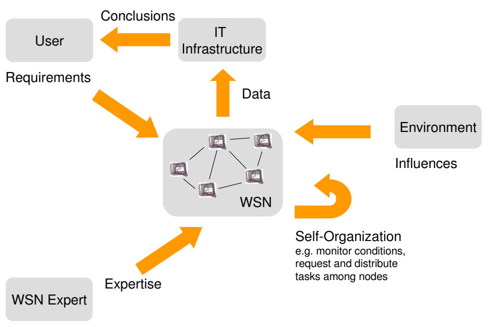

Figure 5 Realized WSN application development process

A user of WSN technology can design and adapt a sensor network application with *Surfer OS* independently. Successively the user migrates services representing application logic as well as different protocols into the WSN and onto the nodes composing his own application for his specific requirements. The ability to add, remove and replace services during runtime gives the user a high degree of flexibility and independence. Especially in contrast to approaches using multi-hop over the air flashing and replacing the whole software image on a sensor node, the service based approach of *Surfer OS* produces less network traffic if changes to an application become necessary. The services themselves, especially those that demand for special WSN knowledge, are provided by WSN experts who have the expertise in the challenges of distributed ad-hoc systems. Services can also be implemented by the user following the service template. In order to compose a WSN application, the user can utilize the provided graphical user interface for manually transfer needed services onto the nodes. Our presented infrastructure also allows for using even professional application design and integration tools such as BPEL, which represent the WSN application as business processes. Using such tools eases the integration of the sensor network into an existing IT infrastructure.

The services provided in the *Service Repository* can also implement selforganizing functionality causing other services or data to migrate or replicate themselves automatically. Even a stateful migration of services is possible by migrating the according .DATA segment from the RAM and the dynamically allocated memory as well. Services providing self organizing functionalities can be realized with *Surfer OS*. They enable the network to react independently on topology changes or depleting energy or memory ressources extending the lifetime of the sensor network application. Furthermore, such approaches again hide network complexity from the user.

Additionally, there is no need for predicting and modeling radio characteristics in order to specify a routing protocol or other topology dependent algorithms. The service-oriented approach in *Surfer OS* that encapsules every functionality as a service allows the replacement even of the radio stack including the replacement of routing protocols. Thus, the right protocols can be chosen during the deployment till satisfactory results are achieved. In the same manner the RS232 stack can be replaced if changed IT infrastructure demands for a new data format.

During the development of different services we realized that the service-oriented approach for applications respectively protocols offers even more benefits than just the composition of applications. When implementing protocols, e.g, for routing or localization, we found out that often some basic functionalities are many protocols in common. E.g., a lot of protocols have to be aware of their current neighborhood size or even of the node ids. Therefore all these protocols send so called hello beacons. In this way the nodes in the direct neighborhood are aware of the presence of this neighboring node. When using more than one of these protocols within one application, each protocol performs its own beaconing. This results in a waste of radio bandwidth. Additionally, when the node has already transmitted data packets, there is no need for additional beacons, since the neighbors are already aware of the presence of the node. We realized that even protocols can be subdivided into several services where each service can focus on performing as efficiently as possible. In this way we can reduce the needed resources. Memory can be saved since different services may use the functionality of one service instead of providing their own functionality. In the same way, the communication costs can be reduced since a needed action like a beacon is only performed once.

7.2 Downsides

Adding functionality to the network after deployment has its price. Tightly coupled systems such as TinyOS can be optimized for a special application. Unneeded code modules can be ommitted during the linking process of the application before deployment resulting in a minimal code size. In order to support future application modifications by addition or removal of services, *Surfer OS* must provide all basic functionalities of the sensor node. *Surfer OS* must offer software interfaces to all hardware components on the node which might be used by services that migrate onto the node during deployment even if a hardware component stays unused during the deployment. This implies that the code size of the *Surfer OS* itself cannot be optimized for special applications.

The integration of services and thus the relocation of code itself demands for some additional information and program logic. Each service must provide relocation information in order to make the code executable on the target node. Additionally, *Surfer OS* must keep a dynamic symbol table and offer interfaces to the *ServiceManager*. For the relocation information of the integrated services and for the code for the *ServiceManager* and symbol table additional memory is needed on the nodes. Due to this, a loosely coupled *Surfer OS* application offers less potiential for optimization of code sizes than a tightly coupled application.

Tightly coupled monolithic sensor network applications are usually flashed onto the nodes before deployment with special programming tools that might include special programming hardware. Right before deployment the batteries of the nodes can be fully charged. A *Surfer OS* application is adapted after the nodes are already deployed. The *Surfer OS* infrastructure can be connected to any arbitrary node within the network, allowing the addition or removal of services in the whole network. This flexibility implies that services have to be distributed throughout the network. Thus the modification of an application is naturally accompanied by additional network traffic.

8 Conclusion and Future Work

As state of the art, it is a very tedious task to design and deploy a sensor network application. Special skills in distributed application design and WSN expertise are inevitable. In this work we presented a service-oriented minimal sensor node operating system called *Surfer OS* and an application development infrastructure which allows an application design from a set of existing services making wireless sensor networks accessible for a wider range of users.

In contrast to current monilithically designed applications the highly modular architecture of the *Surfer OS* provides a high degree of flexibility allowing the user to constantly adapt an application to his individual needs. The possibility to migrate stateful services allows for new strategies to efficiently use the resources provide by the network. The presented solution can be transferred into other areas of pervasive computing, where flexible applications on resource constrained devices are needed. With our work we aim to ease the development process of pervasive technology in various fields of application.

References

1 H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: System support for multimodal networks of in-situ sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, pages 50–59, 2003.

2 Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the 1st international conference on Mobile systems, applications and services (MobiSys '03)*, pages 187–200, New York, NY, USA, 2003. ACM.

3 Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06)*, pages 15–28, New York, NY, USA, 2006. ACM.

4 Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

5 Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services (MobiSys '05)*, pages 163–176, New York, NY, USA, 2005. ACM.

6 Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

7 P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

8 Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS-X)*, pages 85–95, New York, NY, USA, 2002. ACM.

9 Martin Lipphardt, Horst Hellbrueck, Dennis Pfisterer, Stefan Ransom, and Stefan Fischer. Practical experiences on mobile inter-body-area-networking. In *Proceedings of the Second International Conference on Body Area Networks (BodyNets'07)*, 2007.

10 Liqian Luo, Tarek F. Abdelzaher, Tian He, and John A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *Trans. on Embedded Computing Sys.*, 5(3):543–576, 2006.

11 Xenofon Koutsoukos Sandeep Neema Manish Kushwaha, Isaac Amundson and Janos Sztipanovits. Oasis: A programming framework for service-oriented sensor networks. In *IEEE/Create-Net COMSWARE 2007*, January 2007.

12 Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006*, pages 212–227. Springer, 2006.

13 OASIS WS-BPEL Technical Committee. Webservices – Business Process Execution Language Version 2.0, 2005.