RESOURCE-AWARE SERVICE ARCHITECTURE FOR COLLABORATION OF SENSOR NODES

Jan Blumenthal
Institute of Applied Microelectronics and Computer Science
University of Rostock
Richard-Wagner-Str. 31
18119 Rostock, Germany
jan.blumenthal@uni-rostock.de

Dirk Timmermann
Institute of Applied Microelectronics and Computer Science
University of Rostock
Richard-Wagner-Str. 31
18119 Rostock, Germany
dirk.timmermann@uni-rostock.de

Abstract Software on sensor nodes usually cannot be updated after deployment of sensor nodes. Thus, dynamical requests in networks and adaptations of the evaluation methods are complex.

This paper presents a novel service architecture (RASA) for small, mobile, and resource-critical sensor networks. This architecture features software changes by injecting services at runtime. These services are executed on the sensor nodes. The design of RASA simplifies data aggregation and locally collaboration of sensor nodes. Thereby, it is possible to extract implicit information of the network and to adapt the software to dynamical changing processes. Hence, RASA is able to react and to change network behavior depending on current conditions. Additionally, it supports the reusability of existing services or parts of services.

RASA meets the strong requirements of sensor networks regarding small resources, e.g. low memory consumption, supporting mobility, and robustness which is demonstrated by a collaboration example.

Keywords: Middleware, Service Architecture, Collaboration, Wireless Sensor Networks

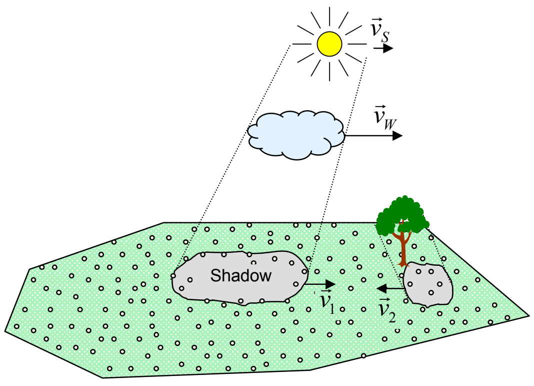


Figure 1. Typical application scenario showing a collaboration of sensor nodes to detect moving phenomenons.

Introduction

Main task of sensor networks is measuring the environment conditions with build-in sensors. One simple example is gathering the temperature around a node. If a given threshold is passed, an event should be triggered to inform other nodes or the base station. That means, sensor nodes simply measure and transmit data as assumed in most of the application scenarios. But in fact, this is inefficient due to missing data aggregation. In worst-case, hundreds or thousands of messages are transmitted and especially sensor nodes close to the base station would resign very quickly due to exhausted batteries caused by a huge number of transmissions. Nevertheless in case all messages were received by a base station, the evaluation of all messages would require huge resources.

As an example in a sensor network with n sensor nodes, exact positioning of nodes requires several $n * n$ matrices. Moreover to detect a movement, the base station must be recalculate the position continiously based on incoming sensor node´s data. Thus, transmitting all measurements to the base station is inappropriate. A local data aggregation between a group of nodes is necessary.

More important than data aggregation is collaboration of sensor nodes to detect different and moving phenomenons. Figure 1 visualizes two groups of shaded sensor nodes. To detect these groups, all nodes collaborate and exchange measurements perceived by light sensors.

Collaboration algorithms can be implemented directly into software of sensor nodes. This software is usually programmed into the flash memory of a sensor node. But sometimes, environment conditions change or are different than expected. Thus, programmed algorithms might be working wrong. In other cases, it is necessary to run additional evaluation algorithms to determine implicit data which was unknown before deployment, e.g. unexpected velocity behavior, surprising termination of nodes, building of mixed groups, or any other phenomenon. To determine what exactly happens in the network, software must be adjustable or changeable. Hence, all nodes need software updates to meet new requests. But collecting all nodes to reprogram them with newly adapted software is economically senseless. Thus, software on sensor nodes must be adaptable at runtime in the terrain. One possibility to support dynamical requests and data aggregation at runtime is based on mobile services.

In this paper, we present a new service architecture which is optimized to support dynamic collaboration of sensor nodes. Further, it considers the special basic conditions for wireless sensor networks in particular and supports highest flexibility within the strong resource limitations on sensor networks.

The remainder of this paper is organized as follows. Section 1.1 introduces services and considers the requirements of architectures in mobile wireless sensor networks. Next, Section 1.2 surveys already related data collection protocols and service architectures supporting data aggregation and downloadable executable code. Further on in Section 1.3, we present our service architecture followed by Section 1.4 which describes the collaboration of services running on sensor nodes at an example. Finally, the paper ends with a conclusion.

1. Requirements for Software on Sensor Nodes

A service is a piece of software to reply remote requests, to interact between devices, and to hide the heterogeneity of a distributed system. Services are a growing technology in mobile ad hoc networks. But in sensor networks, services hit upon several barriers caused by limited resources, e.g. small batteries and energy consumption respectively, few program and very few data memory, highly specialized and tiny operating systems. Thus, saving messages to temporary buffers for retransmission or storing huge service data is nearly impossible. Further, development of a service architecture common to all sensor networks is restricted by different and complex memory architectures of microcontrollers used in sensor nodes, various memory types with different addressing schemes, either separation or combination of program and data memory, and dissimilar argument delivery to subfunctions (stack based, memory based, or register based).

Based on our analysis, a service architecture for sensor networks must meet their special needs compared to resource-uncritical ad hoc networks, must be highly optimized regarding the strong requirements, and therefore fulfil following criterions:

- Supporting new services at runtime
- Interaction and collaboration of services between sensor nodes
- Autonomously running services
- Extraction of implicit information
- Simple programming of services with large flexibility
- Fast and resource-aware seeding of services
- Low usage of data memory
- Minimizing number of data transmissions
- Specialization to the specific memory architecture
- Reusage of existing program modules

2. Related Work

A lot of service architectures or data aggregation schemes are already proposed for sensor networks. All of them match most of the introduced criterions but not all.

The most important concepts are proposed by the University of Berkeley, e.g. MATÉ [1] which is a byte-code interpreter for TinyOS [2]. It is a small communication-centric virtual machine designed as a component for the system architecture of TinyOS. The motivation to develop MATÉ was to solve novel problems in sensor network management and programming, in response to changing tasks, e.g. exchange of the data aggregation function. However, the associated inevitable reprogramming of hundreds or thousands of nodes is restricted to energy and especially storage resources of sensor nodes. Furthermore, the network is limited in bandwidth and network activity as a large energy draw. MATÉ attempts to solve these problems by propagating small code capsules through the sensor network. The virtual machine of MATÉ provides the possibility to compose a wide range of sensor network applications by using a small set of primitives. In MATÉ, these primitives are one-byte instructions and they are stored into capsules of 24 instructions together with identifying and versioning information. In contrast to our design issues, MATÉ lacks the flexibility to use existing system components and it is limited by available data memory to store capsules and runtime data. Further, it is difficult to use MATÉ in conjunction with complex data aggregation schemes or group building algorithms. Thus, MATÉ is very useful for small and easy applications.

Another proposed software is TinyDB - a query processing system for extracting information from a network of sensor nodes running TinyOS [3]. TinyDB provides a simple, SQL-like interface to specify the kind of data being extracted from the network along with additional parameters, e.g. data refresh rate. The primary goal of TinyDB is to prevent the user from writing embedded C programs for sensor nodes or composing capsules of instructions regarding to MATÉ. The TinyDB framework allows faster development and deployment of data-driven applications than developing, compiling, and deploying a TinyOS application. Given a query specifying the data interests, TinyDB collects data from sensor nodes in the environment, filters and aggregates the data, and routes it to the user autonomously. The network topology in TinyDB is a routing tree. Query messages flood down the tree and data messages flow backup the tree participating in more complex data query processing algorithms. Thus, the flexibility of TinyDB is limited by the TinyDB implementation. But more important, TinyDB is simply a query processing system and not a service infrastructure to interact between neighboring sensor nodes. Similar approches are Cougar [4] and SINA [5].

3. RASA Service Architecture

RASA is a new service architecture which meets the introduced requirements (changing runtime behavior, collaboration of nodes, extracting implicit data etc.). This service architecture uses services which are injected to the network by a node (inquirer) at runtime. A service is forwarded to other nodes and installed as well as executed on all nodes of the network successively. After their installation, services are accessible by other nodes and other services, too. Due to small available data memory in sensor nodes and the relatively high energy costs of data memory compared to flash memory, it is important to reduce consumption of data memory. Therefore, our objective is to store the code and most of data in flash memory.

A service is usually characterized by handling the heterogenity of a network and is executed on top of the middleware layer lying upon different operating systems. We tightened this definition to meet the special conditions regarding limited resources in wireless sensor networks. In our view, a special middleware layer on top of the operating system, as the well known CORBA [6], to support completely different devices is

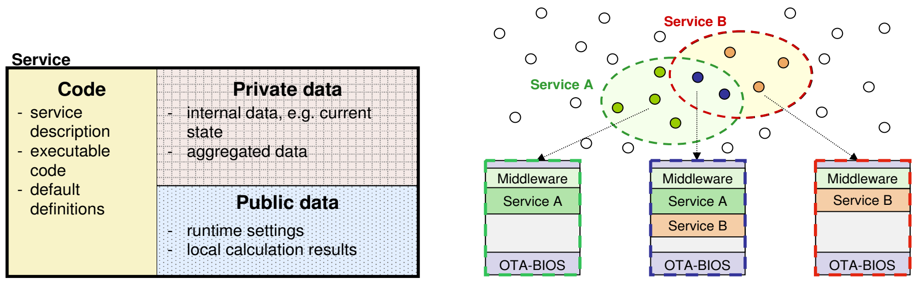


Figure 2. a) Structure of a service in RASA. It is divided into executable code and constants, public data, and private data. Code, constants, and public data are designed to be transmitted to neighboring nodes within a service message. b) Memory organization in case of running two relocatable services at once.

oversized and not applicable. In principe, a sensor network has only one main task, it should measure and aggregate data followed by a transmission. Therefore, a sensor network is build of homogeneous sensor nodes to reduce production costs, to ease programming of nodes as well as to reduce adaptation effort. There is no reason to deploy and support different types of sensor nodes. In our view, most node applications will use same or similar hardware and run applications adapted for the underlaying hardware. Thus, our middleware layer has reduced functionality. It contains basically a service dispatcher and hides the driver layer. The service dispatcher manages, connects, and runs services. The resulting middleware layer does not hide a heterogenic network, but therefore, it provides an interface to services which is significantly more relevant than hiding at all costs.

3.1 Services

Mobile agents are tiny programs which move through the network and are executed on the nodes. If necessary, these mobile agents are forwarded to neighboring nodes and executed there, too. Our service architecture is based on the concept of mobile agents [7].

In our service architecture, a service is simply divided into three parts: executable code and constants, private data, and public data (Figure 2). First, code and constants contain executable code (mobile agent) either as native code or as virtual code designed for a virtual machine. Native code has the advantage (compared to virtual code) that the corresponding service is extremely fast and has a very high degree of freedom caused by direct hardware programming.

Constants are definitions and defaults to initialize the local instance of a service. Second, public data contains information representing the

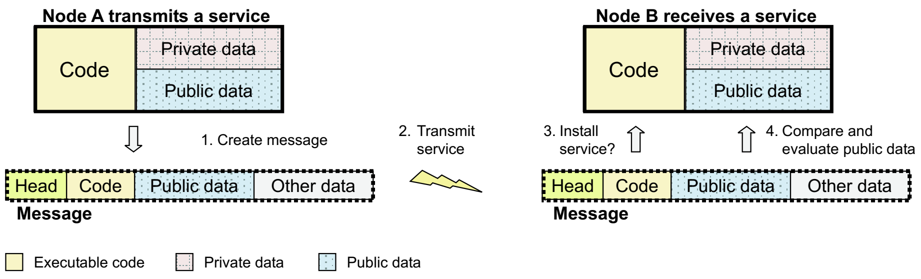


Figure 3. Forwarding a service by embedding executable code, constants and public data into a message (Step 1) and forwarding from node A to node B (Step 2). Node B installs the executable code (Step 3), if necessary, and executes the code to compare node´s A public data with local public data (Step 4).

state or current results of the service instance. This data is public and therefore accessible by other nodes. In contrast, private data is not accessible by other nodes. It is used to store measurement values or internal variables.

Executable code, constants, and public data are transmitted by one service message from one node to another (Figure 3). Therefore, public data must be serializable. It must not contain data pointers or other references to local conditions. Additionally, public data is relocatable. After receiving a message, constant code is programmed into the non-volatile flash memory. The runtime system provides data memory required for private and public data. Received public data is used to optimize start conditions of the service and to aggregate data.

The selection which data is assigned to public or private data highly depends on the service. Usually, measured data is stored in private data memory and aggregated data is stored in public data memory to share these results with other nodes. Nevertheless, an important design criterion is the reduction of message´s length.

3.2 Splitting into Modules

According to our requirements, our service infrastructure is designed to reuse existing software components. In our context, these components are called modules. A module is a small piece of executable code with public as well as private data. Modules can be assembled together to fulfil a specific task. At least one module is required per service.

Figure 4 demonstrates a service consisting of four modules (analysis, grouping, aggregation, and transmission). Modules are usually executed in sequence or in any other order, if specified. Each of these modules has its own private and public data area within the service´s data space.

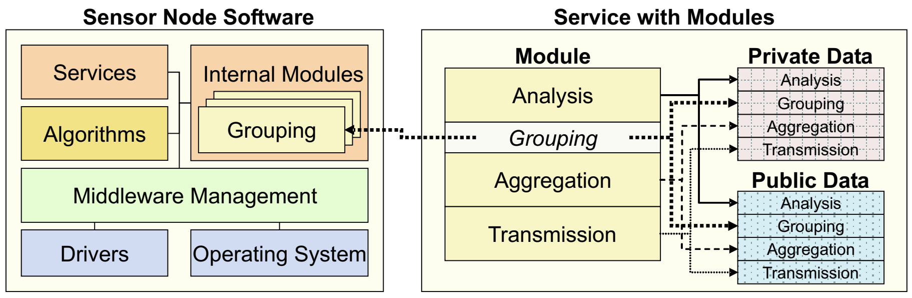


Figure 4. Interaction between original installed node software and flashed service with access to an internal module (Grouping). The grouping definition of the service is only a frame which references directly to the internal grouping module. Note, the internal grouping module uses service´s data memory to store own data.

Thus, if the service is transmitted to an another node, the complete code of all modules and the public data block are transmitted as exemplified in Figure 3.

The service presented in Figure 4 denotes another nice feature of our service architecture - using internal modules. On this way, software components such as *grouping of nodes* may be outsourced, defined as internal modules and installed before deployment. These modules can be used by any installed service at runtime simultaneously due to using different parts of the data memory. Internal modules are not be transmitted anymore, because they are enclosed within the node software. Thus, the size of code within a service message is reduced significantly.

3.3 Embedding a Service

Services are managed by a service dispatcher. To store received services, node software holds preallocated pieces of flash as well as data memory. Due to the separation of constants (executable code and constants) and data, this service architecture is able to run on systems with von-Neuman and Harvard configurations. Figure 5 shows the memory organization scheme for a Harvard architecture (8051 mircocontroller). The bold framed areas visualize available memory preallocated by the node´ software to store services. If a service was received, the service dispatcher checks whether the service fits into the remaining memory space and installs the service, if possible. During installation, the service dispatcher sets the correct absolut memory pointer to access the correct data area.

Figure 5 further displays how a service is organized within the memory pool. It starts with a service description (Head: Service) which

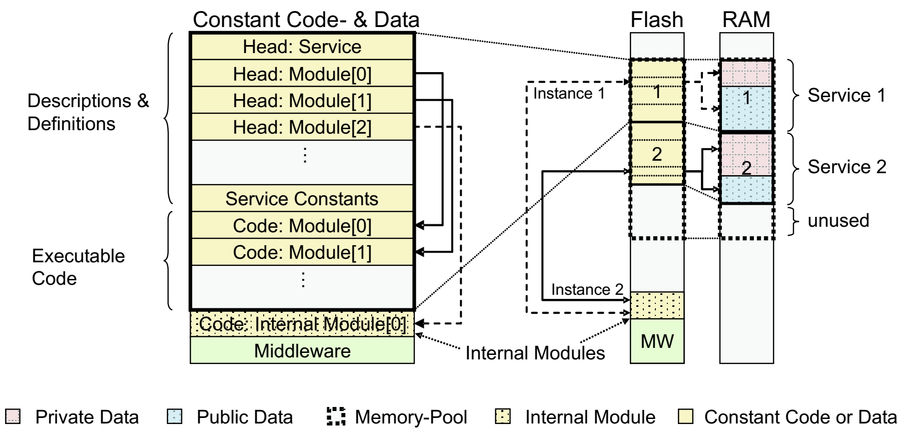
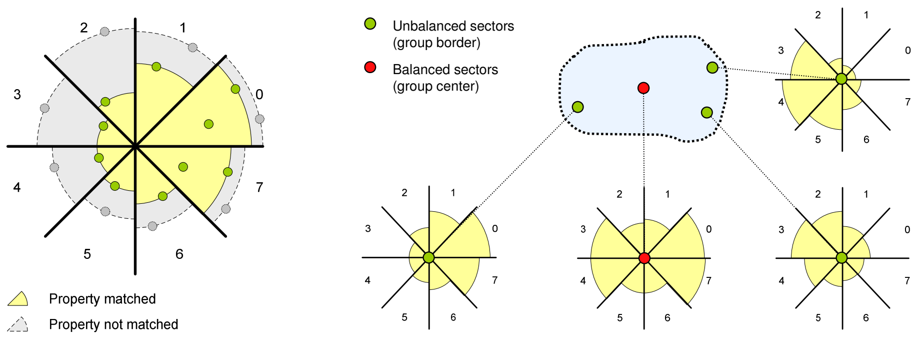


Figure 5. Memory organization scheme for a Harvard architecture (8051 mircocontroller) of two possible services (instance 1 and instance 2). Each service references to an internal module. To store runtime data, this internal module was assigned to a part of the service´s data memory.

defines required data space, number of modules, and a unique service identifer (ID) to distinguish different services. After service definitions, each module is described. Here, relative data offsets point to the public and private data memory within the assigned data area. The module definition further contains relative offsets to module-specific functions within the module´ memory of executable code. These offsets are used to initialize, start, and end the service. After all definitions, application depending constants and executable code follow.

All defined offsets are references to the beginning of the service. Thus, the service dispatcher can easily calculate physical addresses of all functions. Internal modules are announced by a special flag which supplementary indicates that relative code offsets are added to a system-wide jump table within the BIOS memory space. Hence, this jump table separates the BIOS from services. Thus, same service might be run on various BIOS versions provided all required jumps are implemented.

In most cases, internal modules require additional data memory. The size is determined at compile time and is taken into account to the data offsets for private and public memory areas. Thus, calling internal modules leads to an invocation of a BIOS function by using a part of the data memory assigned to the current running service.



2
1
3
0
4
7
5
6
Property matched
Property not matched
Unbalanced sectors (group border)
Balanced sectors (group center)

Figure 6. a) Border description by 8 sectors (45°), b) Local views of group borders at different sensor nodes

4. Collaboration of Services

A suitable example to explain a collaboration between sensor nodes is the organization of groups in sensor networks. In groups, all measurements are aggregated locally without a central node and the result is forwarded to the requesting node. Groups consist of sensor nodes which satisfy same property, e.g. current temperature of all group members oscillates between 20°C and 25°C. All emerged groups are dynamically rearranged depending on the environmental conditions.

A group in our view is defined by sectors with an angle of 45° (Figure 6). The radius of a sector is given by the position of the most remotely neighboring sensor node (*property node*) which meets the property requirements. If there is a sensor node within this sector which does not achieve the requirements (*non-property node*), the radius of this sector is reduced accordingly. Thereby, the sensor node gets a local view of the group´s dimensions. The border description by sectors is extremely memory saving, because storing sectors requires only 64 bytes constantly.

Further, 45° sectors allow comparison methods without the need of trigonometric and hyperbolic functions and is therefore able to run on systems without floating point unit. Mapping of real borders to sectors is faulty and increases with higher distances to the local node. But as an approximation of a group expansion, the description error is acceptable. Sector borders represent a good compromise between computational and communication effort, resource consumption, error and benefit.

To accomplish the requirements of software in sensor networks described in Section 1.1, we are currently testing a service as visualized in Figure 7. This service consists of several modules. One of these is the grouping module which defines how a group memberchip (we call this the *property*) is determined. This grouping module does not contain any

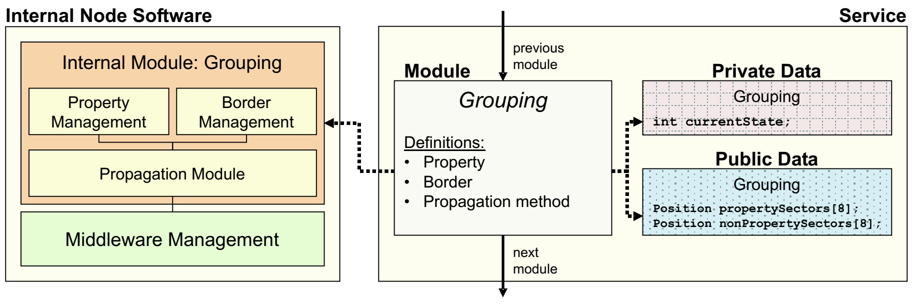


Figure 7. Structure of a service using internal modules to organize groups. The description of the service´s module *Grouping* only defines property handling, border detection, and propagation method but no executable code. Executable code of the internal grouping module is statically linked to the node´s software and not part of the service. Datatype *Position* consists of two 16 bit wide integer variables (x,y).

executable code but it links to the internal grouping module whereby it selects the border management function as well as the propagation method. The border management handles detection and management of borders. The propagation method defines how group borders are distributed to other nodes. By this separation, changing the border management, e.g. using 16 sectors, only requires a recompilation but not a change of the interfaces within the service´s modules.

The collaboration of sensor nodes to create a group is done within the propagation module. Generally, if a sensor node receives a service, it creates an instance of the service and initializes this instance with the received public data of the remote node. Then, the service starts and runs through the service´s modules. In the grouping module, the service checks if it meets the property description or not and updates the appropriate sector arrays (Figure 7: *propertySectors* and *nonPropertySectors*). In case an update of the sector arrays was necessary, the propagation module initiates a transmission of the service including code, constants and public data (both sector arrays). All nodes receiving this message may now update their own sector arrays and broadcast their own sectors, too. To increase coverage speed and reduce communication overhead, each node stores at least 8 property-based sectors and 8 sectors of non property-based sectors.

5. Conclusion

This paper presents the novel service architecture (RASA) for flexible and mobile services in wireless sensor networks. These services are mobile agents containing executable code which are injected into the network without previous knowledge and are executed on every receiving node. A service consists of several modules to support reusability of already written components and to reduce service´s size by outsourcing often used components to the node software (BIOS). Further, the paper presents an example application to group sensor nodes depending on freely definable conditions.

This architecture meets the strong requirements present in sensor networks such as updating the network with new software versions or services heeding small resources, data aggregation, and extracting implicit information from the network. Due to its design, the service architecture is applicable in different hardware architectures (von Neuman, Harvard) and guarantees highest degree of freedom in programming services.

References

[1] P. Levis, D. Culler, MATÉ: a tiny virtual machine for sensor networks, in Proc. of ACM Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS), December 2002.

[2] J. Hill et al., System architecture directions for networked sensors, in Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[3] S. Madden, J. Hellerstein, and W. Hong, TinyDB: in-network query processing in TinyOS, Intel Research, IRB-TR-02-014, October 2002.

[4] C. Jaikaeo, C. Srisathapornphat, and C.-C. Shen, Querying and Tasking in Sensor Networks, SPIE's 14th Annual Int'l. Symp. Aerospace/Defense Sensing, Simulation, and Control, Orlando, FL, Apr. 2000

[5] P. Bonnet, J. Gehrke, and P. Seshadri, Querying the physical world, IEEE Personal Communication, October 2000.

[6] Object Management Group, Common Object Request Broker Architecture (CORBA/IIOP), V3.03, March 2004.

[7] D. Chess, C. Harrison, A. Kershenbaum, Mobile Agents: Are they a good idea?, IBM, New York, 1995.

[8] J. Blumenthal, M. Handy, D. Timmermann, SeNeTs - Test and Validation Environment for Applications in Large-Scale Wireless Sensor Networks, 2nd IEEE International Conference on Industrial Informatics, INDIN 04, page 69-73, ISBN: 0-7803-8513-6, Berlin, Germany, 2004.