A Middleware Approach to Configure Security in WSN

Peter Langendoerfer[1], Steffen Peter[1], Krzysztof Piotrowski[1], Renato Nunes[2], and Augusto Casaca[2]

[1] IHP, Im Technologiepark 25, 15236 Frankfurt (Oder), Germany
{langendoerfer|peter|piotrowski}@ihp-microelectronics.com
[2] INOV, Rua Alves Redol, 9 - 1000-029 Lisboa, Portugal
{renato.nunes|augusto.casaca}@inesc-id.pt

Abstract. Security configuration of standard systems is a tedious and error prone task. Doing this for WSN is even more complex due to the scarce resources of the sensor nodes. In order to simplify this task we propose a middleware architecture as well as a configuration tool. The main idea is that the configuration tool selects security providing modules such as appropriate cipher means. The choice is based on a detailed description of security needs of the application under development as well as on the description of the available security modules and sensor nodes. The middleware architecture supports configuration before and after deployment of the sensor nodes. It consist of an essential core that provides configuration features and an additional layer in which the security modules are clustered.

1 Introduction

It is obvious that security is an essential need in ubiquitous wireless computing and there are plenty of means that promise to secure Wireless Sensor Networks (WSNs). In WSNs obvious parameters like security strength are overshadowed by network structure, the number of available base stations, requirements in response times, frequency of security operations and especially by the energy consumption of cryptographic operations. All these parameters must be considered if security is supposed to be incorporated in the WSN. This is a very challenging task even for security experts, and programming of a WSN is already a complex task by itself.

Our approach to reduce complexity of the realization of an appropriate security level for a given WSN application is to provide a configurable and adaptive security middleware. The configuration of an initial set of security modules is done by our configuration kit (configKIT) before deployment of the application. With respect to security issues, the application programmer has to specify only which security functionality, e.g. data secrecy and authentication is needed by the application. In addition she has to provide some information on the sensor node configuration, e.g. processor, memory size etc. Based on this data the configKIT selects the appropriate security modules. The security modules come with a self description concerning functionality and required resources for example. These modules are provided by the UbiSec&Sens project [16] into which our activities are also embedded. Adaptability ensures that security modules can be exchanged after deployment, e.g. if application needs have changed or if vulnerabilities of a certain module have been detected which require an update. This functionality is achieved by our modular architecture which separates core functionality needed for adaptability support from pure security functionalities.

The rest of this paper is structured as follows. Section 2 provides a short state of the art. The configuration tool is introduced in section 3. The following section discusses our middleware architecture. The paper concludes with a short summary and an outlook on further research steps.

2 State of the Art

Wireless sensor networks are mainly used to gather data about a certain environment, see for example [13]. Due to this focus also research in the middleware area has somewhat concentrated on supporting data storage and retrieval issues in WSNs. Some prominent approaches are tinyDB [10], Cougar [18] and Hood [17] to name just a few.

Some work towards flexible middleware for WSNs has already been done. These approaches try to provide application independent support to applications but are mainly focusing on communication issues in one form or another. In [19] authors introduce the concept of reconfigurability for middleware in pervasive computing. Here the major part, if not all components, of the middleware is located at a PDA device and the task of the middleware is merely discovery provision of available data. Authors of [15] propose an application independent scheme for defining groups of sensors to provide easy adaptability of a WSN to new applications. Here part of the adaptation logic is placed at the sensor nodes. A similar approach exploiting roles of sensor nodes is proposed in [9].

Our middleware approach differs from those cited above by that we are focusing on a very specific functionality, i.e. security instead of trying to provide a communication or programming abstraction. In our approach flexibility is addressing support of a wide range of applications and individual support of the security needs of each application. I.e. we are trying to provide a tailor-made security solution for each application, and provide means to update the security modules during the runtime if necessary. In order to achieve this goal we are working towards a middleware compiler which selects security modules based on application and sensor node requirements and constraints respectively. In this area some work has been done, which did not focus on WSN and security issues but aims at a similar goal, i.e. providing tool support for development of a certain middleware. Most of those approaches are based on model driven architecture (MDA) [2]. The tool sets Cadena [7], VEST [14] and CoSMIC [1] are MDA based and try to support development of platforms for embedded systems. By that they provide functionality similar to our approach, the difference is that we are focusing on security and do not use MDA but defined our platform architecture independently of any formal model. In addition only VEST supports the modelling of security aspects.

In [5] the authors discuss the integration of security aspects into a formal method based development of networked embedded systems. The focus of the security analysis language (SAL) is merely on information flow between networked entities. By that it might be a way to model security requirements of applications residing on top of our security modules and to verify whether or not our middleware compiler selected the correct modules.

3 Middleware Security Compiler

It is the goal to have a middleware that provides security functionality in a very flexible and adaptable way.

The security provided by our toolbox can be tailor-made on a per application basis and even be adapted during the life cycle of a certain application. This level of flexibility is achieved by a modular middleware architecture and by introducing the concept of a middleware compiler.

3.1 Overview of the Compiler Architecture

Since the gravity center of this work is security for WSN as such and not for a specialised application domain or even a single application, several solutions for each security service are required to be capable to provide security for a wide range of applications. This means that in order to provide flexibility of the choice there is a need for multiple modules that provide a specific functionality—service but differentiated with respect to security parameters, code size, etc.

The selection of the suitable security modules is done by our middleware compiler. In order to generate a suitable set of security modules for a certain application reasonable constraints need to be defined in advance. These constraints are on one hand due to the limitations of wireless sensor nodes and on the other hand imposed by the security that the application under development requires. The hardware driven constraints are for example processing power and available energy to name just a few. Application dependent constraints are lifetime of the overall network, security features like secrecy of measured data or similar. In order to define the relevant constraints an XML based description language for sensor nodes and application requirements is under development. Also the role of a specific sensor, if static or default, influences its software set-up. The sensor node description provides information concerning the hardware set-up of a sensor node plus relevant information of its software configuration such as operating system used and already allocated memory.

In addition to the description of the above mentioned constraints the security modules provide a self description. This description provides information concerning the functionality of the module, and the needed resources—memory footprint and processing power. If a module needs other additional modules its description also contains information on potential dependencies. For example, an cipher mechanisms may require that a secure random number generator is also deployed.

Additionally, since the resources of a sensor node are usually constrained we suggest that every functionality that might be used by multiple modules is also present as a module in order to avoid code redundancy.

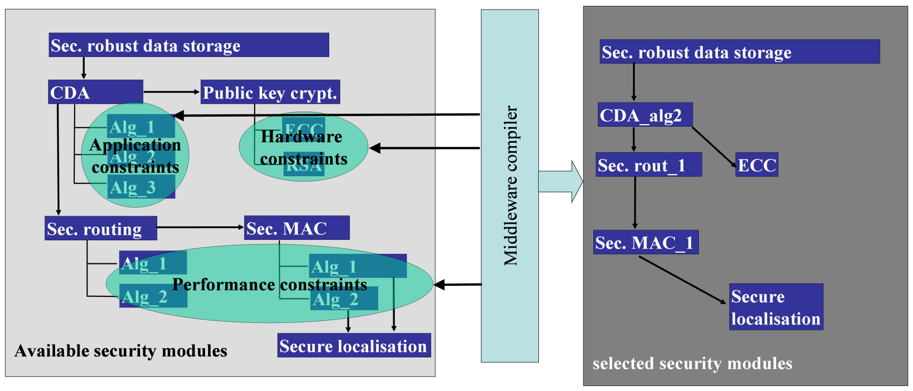

Fig. 1. The Middleware Compiler: the appropriate modules are selected based on application requirements and sensor node constraints.

Figure 1 illustrates the idea of the middleware compiler. The result of a successful compiler run is an instance of the secure middleware—the right hand side of the figure.

3.2 Compiler Operation

Our compiler approach requires the definition of abstract and concrete APIs between security modules. Using these interfaces the selection of the suitable security modules can be done by our middleware compiler. This compiler requires in addition the following information:

- required functions
- available modules
- constraints concerning performance and security

In following, these points are explained referring to an example. The example function is the authentication of another node.

Required functions It describes the needed functionality. In this example it is the authentication. The description does not contain any further specification details.

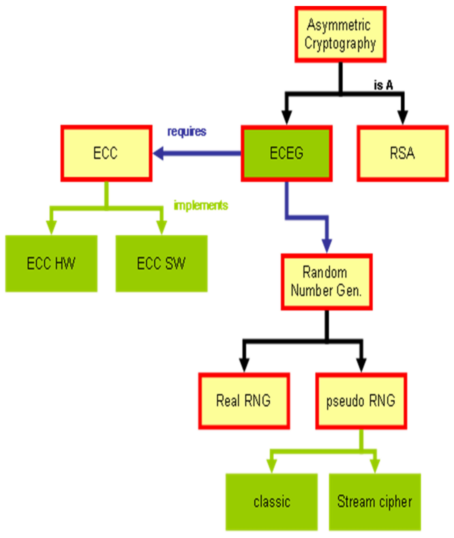

Fig. 2. Exemplary fragment of a dependency graph for asymmetric cipher mechanisms. Green boxes contain real code while bright boxes are logical classes—interfaces.

Available modules The description of available modules can be considered as a kind of database that contains every security module, its dependencies, interface description, security parameters, code size, etc. It is the notion that every module that is available has such a description. Based on these single module descriptions the compiler generates a network of modules, with dependencies and available implementations. Figure 2 shows an example of such a dependency graph. It is a part of the complete structure for our example, and as shown, there is a choice of the underlying mechanisms used by the authentication. It can be either ECEG (Elliptic Curve ElGamal) or RSA. If ECEG is chosen, an ECC (Elliptic Curve Cryptography) implementation and a Random Number Generator are required. For both of them diverse implementations are available.

Constraints concerning performance and security Based on the graph of available modules and the required functions the compiler determines possible configurations (i.e. sets of modules) that do not have open dependencies and provide every needed function. Indeed, some of these configurations will not meet the requirements of the system, either because hardware parameters are not met or due to wrong security properties. The description of such constraints helps to filter out configurations that do not fulfil the requirements. For instance, in our authentication example, there might be a constraint saying that the code size for the authentication functions must be less than 2 kB. In such a case, every configuration where the code size is more than 2 kB is withdrawn. Another possible constraint is security strength. If the example needs very strong authentication, all configurations based on weak cryptography implementations are withdrawn.

Final Evaluation At this point a set of possible configurations that meet every constraint, should be picked out. It is considerable that now the compiler evaluates extended parameters like energy consumption, total code size, performance, security implications and the like and chooses the best configuration or presents the results to the developer who chooses the best set-up.

4 Flexible Security Middleware Architecture

4.1 Overview

Starting on the highest level, there are two main classes of participants, i.e. sensor nodes and Configuration Centers. Figure 3 depicts our intended middleware architecture for both these classes. A Configuration Center runs applications that support the management of the WSN and allow access to data from specific sensors and aggregated data, and also to receive alarms. There may be multiple instances of Configuration Center, thus from now on the term Configuration Center refers to the class not to a single instance, if not otherwise stated.

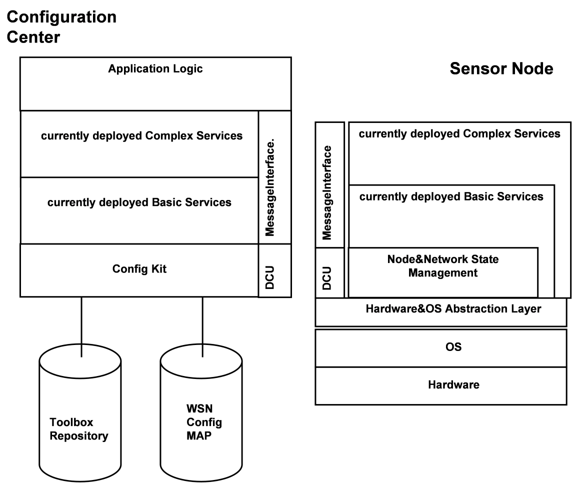
Configuration Center
Application Logic
currently deployed Complex Services
currently deployed Basic Services
Config Kit
MessageInterface.
DCU
Toolbox Repository
WSN Config MAP
Sensor Node
MessageInterface
DCU
currently deployed Complex Services
currently deployed Basic Services
Node&Network State Management
Hardware&OS Abstraction Layer
OS
Hardware

Fig. 3. Our middleware architecture: Configuration Center on the left and sensor node on the right hand side

Our middleware architecture distinguishes between three classes of components, where each component consist of one or more modules. These classes are:

- sensor node abstraction layer
- middleware core
- security services

The sensor node abstraction layer is the only operating system and hardware dependent component. It has to be adapted individually for each OS/hardware combination that shall be supported.

The middleware core consists of modules that are necessary to guarantee proper functionality of the other modules and those that are needed on all devices of a system based on our approach.

Security services are modules that contain the actual security functions, i.e. cryptographic means, security protocols and the like. The security services may use each other what implies a logic differentiation between basic services and complex services.

The general middleware architecture is mainly independent of the participant type, i.e. the deviations between sensor nodes and Configuration Centers are minor. The major differences concern the presence of the sensor node abstraction layer that is needed at sensor nodes, and the inclusion of the Middleware Compiler that is necessary only at the Configuration Center.

The concrete instantiation of the middleware, i.e. the modules deployed at the sensor nodes and at the Configuration Centers depends on:

– the currently running application
– the current role of the sensor node
– sensor node capabilities

4.2 Core Components

State Management Module (SMM) The SMM monitors the sensor node and maintains its state. By that it can trigger a code update for example if the sensor node reaches the management state, which might be caused by expiration of timers or by external triggers such as detection of malicious actions.

Each sensor node can be in one of the following four states (see Figure 4):

– M0 off-the-shelf state: the node is equipped with basic functions needed for the initialization process. The initial software configuration is put on the node. In order to protect keys, this can happen in a secure environment. After that the node is ready for deployment.
– M1 initialization: after the sensor nodes are deployed they are performing the network set-up, e.g. exploring their neighbourhood, setting up routing information, etc.
– M2 normal operation: the sensor node executes its application specific task.
– M3 management: If an unexpected behaviour is detected, e.g. caused by environment or by an attack, the node will enter the management state. The reason for the interruption is analysed and appropriate actions are initiated. The management state is also entered when an code update has to be executed. Whether a dynamic code update or re-start of the sensor node is necessary, depends on the trigger which initiated the transition from M2 to M3. If DCU has to be executed the Configuration Center shall verify if only the requesting node has to be updated or whether other nodes in the network also need to be re programmed (see Figure 5)

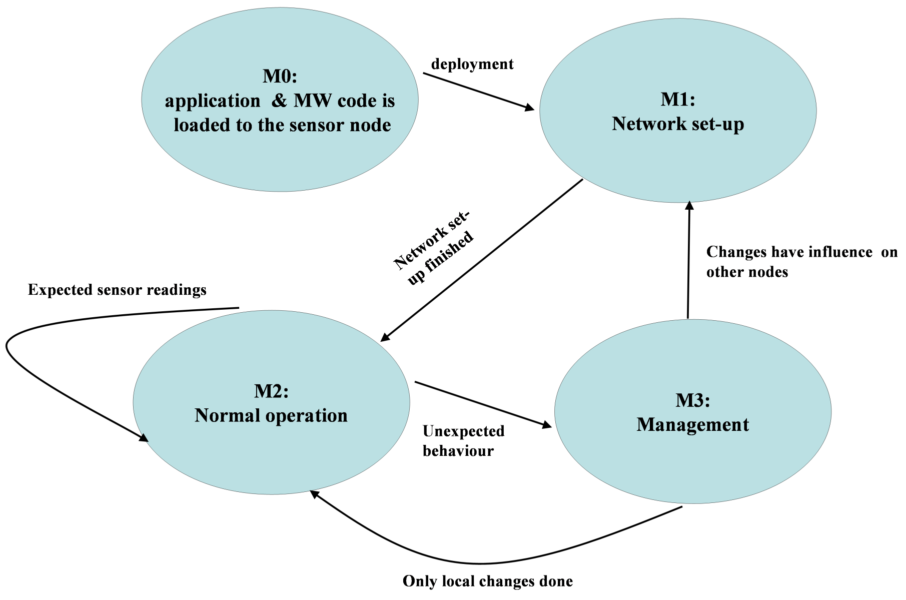

Fig. 4. Sensor node states before and after deployment

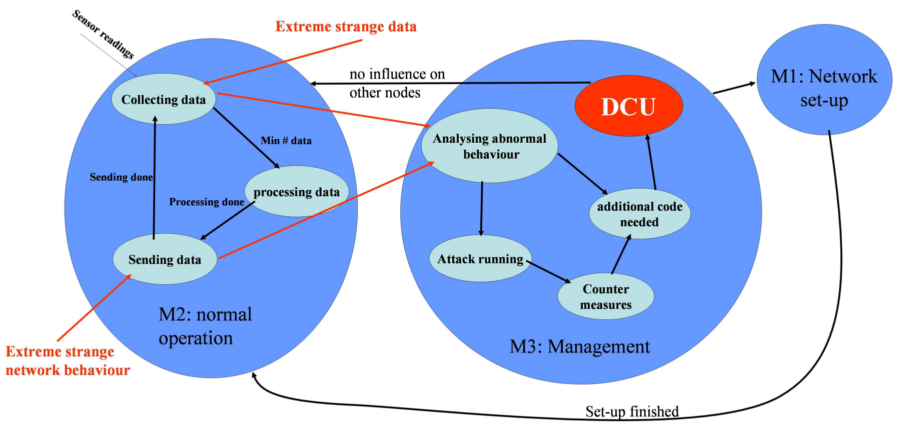

Fig. 5. State M2 and M3 including their internal states and events triggering the transition from M2 to M3 and vice versa

Message Interpreter The Message Interpreter provides local intelligence which is needed to decide for example if the current sensor node is capable to answer a query correctly or whether it has to forward the message. In addition it is a kind of middleware scheduler which passes incoming data to the corresponding middleware modules.

The message interpreter consists of two parts. It has a static part that is responsible for dealing with all messages that are directed to the middleware core components DCU and SMM. Its configurable part depends on the services deployed on the node. In order to properly support the configurable part the message interpreter uses a kind of registry which is shared with and maintained by the DCU module. Each time a module is exchanged, deleted or additionally installed the DCU module updates the registry. Thus, the message interpreter always knows which modules are available. Depending on the currently available modules and the node's current role and the message address the message interpreter decides what to do. In principle it is a three stages decision chain.

1. If the node is not the intended recipient the message forwarded.
2. If the node is the intended recipient the message interpreter checks if the corresponding module is deployed. If *yes* It is checked if the sensor node run in the appropriate role. If *yes* the message is delivered, otherwise it it forwarded to a more appropriate node.
3. If the node is the intended recipient but the corresponding module is not deployed the SMM is informed about this misalignment. The SMM can then decide what to do. Options are sending a misalignment message back to the configuration center, requiring a code update or just ignore the misalignment. The reaction of the SMM will depend on the sender of the message, i.e. if the sender is a well known trustworthy party some action will be taken otherwise the misalignment might be ignored.

Abstraction layer The abstraction layer provides generic interfaces to basic and complex services so they can be developed independent of the underlying operating system. Due to the nature of the security modules under development we foresee two interfaces: a storage and a communication interface. The former will provide memory management functionality such as allocation of memory, store and fetch operations of data items used by higher layers. The communication interface handles incoming and outgoing messages. The latter are passed as payload to the appropriate OS dependent interface. Incoming messages are passed to the message interpreter after removing all protocol headers and trailers if necessary, i.e. if the OS did not already remove them.

DCU This module is necessary to allow reconfiguration of sensor nodes during their lifetime. Potential triggers can be newly detected vulnerabilities of security modules or a simple reconfiguration due to deployment of new applications.

It can be assumed that there is a need for change or adaption of the security mechanisms also after the deployment of the nodes. It can be caused by a changed network structure or size, or even by a broken security algorithm. This is why dynamic code updates are an substantial need of our security middleware. The diagram shown in figure 5 refers to DCU in state M3. That means that the system requires the capability to change the functionality running on the sensor node during runtime whenever it is needed by the management. In kernel based operating systems like Contiki[4] such dynamic updates are not very challenging. Processes can be added or stopped and executable code can be stored or removed from the node. In very resource efficient operating systems like TinyOS[8] that merge operating system and application to one image it is not that straightforward to change the functionality. Recently several mechanisms have been developed that provide code update functionality for TinyOS. Currently we are focusing on FlexCup [11],[12] that allows to change methods at runtime.

If the configuration of sensor nodes is changed during their life time, this is recorded at the WSN configuration map repository (see fig. 3). The repository always reflects the middleware instantiation of all sensor nodes starting from the first set-up. The current set-up of all nodes within a certain part or with a common task is used as an additional constraint whenever a code update is required after deployment. By that interoperability inside the WSN can be guaranteed, e.g. the use of different aggregator node election algorithms can be avoided.

4.3 Service Layer

As already mentioned there are two logic classes of services. This has been reflected in Figure 3 as well. The services are build from modules. In general, one module can use other modules, i.e. can use the functionality provided by other modules. This dependency causes that some of the services are on the top of the stack or tree, what actually causes that their functionality is more complex compared to services from lower parts of this structure.

This differentiation here is completely independent from the kind of functionality provided. It is only based on the information if a service is used by another or not.

Basic services are modules that do not support several functionality but just one. But they may rely on other basic services such as cipher means do since they require random number generators to be deployed. An example of basic service may be the secure random number generator presented in [3].

Complex services have more complex functionality, e.g. aggregation and persistent storage of sensor readings can be provided by the tinyPEDs service [6]. In order to fulfil their tasks such services may require support from basic services, e.g. to do encryption or decryption. But a complex service may also be implemented in a monolithic way so that it does not need any basic services to fulfil its tasks. Therefore the complex services may access the abstraction layer directly as basic services do.

5 Conclusions

In this paper we have introduced a flexible security providing middleware approach. Our way to achieve flexibility is twofold. On one hand we propose the use of a configuration tool that compiles a security architecture at development time, and on the other hand we provide an architecture that allows for dynamic exchange of security modules at run time.

The use of a middleware compiler like approach ensures that the application programmer no longer needs to be also a security expert, and by that reduces the probability of mis-configurations. The capability of up-dating the security configuration during run time is especially beneficial for long-living applications. It allows to exchange successfully attacked security means against still unbroken ones. The re-compilation of the security configuration of a certain sensor node is also executed by our configKIT.

We are currently finalising the XML based description languages for security modules, sensor node description and security requirements of the application. Our next research steps concern the selection of security modules and means to prove that the combination of the selected modules really provides the required security means at the required security level.

Acknowledgments

The work described in this paper is based on the results of IST FP6 STREP UbiSec&Sens. UbiSec&Sens receives research funding from the European Community's Sixth Framework Programme. Apart from this, the European Commission has no responsibility for the content of this paper. The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

1. Krishnakumar Balasubramanian Arvind. Applying model-driven development to distributed real-time and embedded avionics systems. *International Journal of Embedded Systems*, Special issue on Design and Verification of Real-Time Embedded Software, April 2005.
2. A. Brown, J. Conallen, and D. Tropeano. *Models, Modeling, and Model-Driven Architecture (MDA)*, chapter Introduction:. Springer Verlag, 2005.
3. C. Castelluccia and A. Francillon. Tinyrng, a cryptographic random number generator for wireless sensor network nodes. In *5th Intl. Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks, IEEE WiOpt*, 2007.
4. A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *In First IEEE Workshop on Embedded Networked Sensors*, 2004.
5. Matthew Eby, Jan Werner, Gabor Karsai, and Akos Ledeczi. Integrating security modeling into embedded system design. In *International Conference and Workshop on the Engineering of Computer Based Systems*. IEEE, 2007.

6. J. Girao, D. Westhoff, E. Mykletun, and T. Araki. Tinypeds: Tiny persistent encrypted data storage in asynchronous wireless sensor networks. *Ad Hoc Networks Journal (Elsevier).*, to appear.
7. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.
8. J. Hill, P. Levis, S. Madden, A. Woo, J. Polastre, C. Whitehouse, R. Szewczyk, C. Sharp, D. Gay, M. Welsh, D. Culler, and E. Brewer. TinyOS: http://www.tinyos.net, December 2005.
9. Manish Kochhal, Loren Schwiebert, and Sandeep Gupta. Role-based hierarchical self organization for wireless ad hoc sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 98–107, New York, NY, USA, 2003. ACM Press.
10. Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.
11. 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 *Wireless Sensor Networks: Third European Workshop, EWSN 2006.* Springer-Verlag, 2006.
12. A. Poschmann, D. Westhoff, and A. Weimerskirch. Dynamic code update for the efficient usage of security components in wsns. In *Proceedings of the 4th Workshop on Mobile Ad-Hoc Networks (WMAN)*, 2007.
13. Kay Römer and Friedemann Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004.
14. J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. Vest: An aspect-based composition tool for real-time systems. In *Proceedings of the IEEE Real-time Applications Symposium*, 2003.
15. Jan Steffan, Ludger Fiege, Mariano Cilia, and Alejandro Buchmann. Towards multi-purpose wireless sensor networks. In *ICW '05: Proceedings of the 2005 Systems Communications (ICW'05, ICHSN'05, ICMCS'05, SENET'05)*, pages 336–341, Washington, DC, USA, 2005. IEEE Computer Society.
16. Website. Ubiquitous sensing and security in the european homeland *http://www.ist-ubisecsens.org/*.
17. Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, New York, NY, USA, 2004. ACM Press.
18. Yong Yao and J. E. Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record*, 31(2):9–18, September 2002.
19. Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, and Sandeep K. S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–40, 2002.