Supporting Lightweight Adaptations in Context-aware Wireless Sensor Networks

Amirhosein Taherkordi, Romain Rouvoy, Quan Le-Trung, Frank Eliassen

Supporting Lightweight Adaptations in Context-aware Wireless Sensor Networks

Amirhosein Taherkordi[1], Romain Rouvoy[1,2], Quan Le-Trung[1], and Frank Eliassen[1]

[1]University of Oslo,
Department of Informatics,
P.O. Box 1080 Blindern,
N-0314 Oslo, Norway
{amirhost, rouvoy, quanle, frank}@ifi.uio.no

[2]ADAM Project-Team, INRIA-USTL-CNRS,
Parc Scientifique de la Haute Borne,
40 avenue Halley, Bt. A, Park Plaza,
F-59650 Villeneuve d'Ascq
romain.rouvoy@lifl.fr

ABSTRACT

Context-aware environments are being populated with *Wireless Sensor Networks* (WSNs), observing sensory context elements, and adapting their behavior accordingly. Although *adaptation* has been known as a common approach for addressing context-awareness, the resource-scarceness of WSNs raises the requirements for *lightweight* adaptations. The related work in the field of updating WSN applications mostly focuses on *i)* developing techniques to distribute a monolithic program to a set of nodes or *ii)* reprogramming the whole sensor nodes, which have been seen as impractical and inefficient solutions for a large number of sensors deployed in inaccessible regions. In this paper, we propose a new software development paradigm, which revisits the way WSN applications are designed in order to optimize the adaptation process. Our approach promotes lightweight adaptation by proposing a component model reconfiguring modules at the behavior-level instead of component-level. We evaluate this model by analyzing a sample reconfigurable application atop CONTIKI—a popular operating system for sensor nodes. The preliminary analysis shows that our adaptation approach is efficient in terms of energy consumption, memory usage, and reconfiguration complexity.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design

General Terms

Design, Performance

Keywords

Wireless Sensor Networks, Context-awareness, Adaptation Middleware, Software Component Model

CAMS 2009, June 16, Dublin, Ireland

1. INTRODUCTION

WSNs have been considered as an emerging technology for monitoring and controlling a variety of applications, such as environmental surveillance, infrastructure monitoring, home and office security, and medical monitoring [1,2]. Applications for WSNs are gradually moving towards *pervasive computing* environments, where sensor nodes have tight interactions with actuators, deal with the dynamic requirements and unpredictable future events, and behave according to the environmental context surrounding them [3,4]. Applications for such environments must observe continuously their execution context in order to detect the conditions under which some behavioral adaptations are required.

In addition to the application context-awareness, WSN architecture itself brings its own context concerns at sensor *node-level* and *network-level*. The former is raised when a wide range of sensor nodes is deployed in a heterogeneous network containing sensors with different sensing parameters, system software, and resource capabilities. The latter refers to the networks in which a high number of nodes are deployed in inaccessible places and we need to update the software in some particular regions, therefore individual software updating becomes an impractical and inefficient solution.

Dynamic reconfiguration of software components has been a common approach for addressing context-awareness of applications. Reconfiguration may include adding, replacing, or removing a component, and also changing the value of component variables. The overhead of existing component-based reconfiguration techniques [6,7] brings new challenges to resource-limited networks, such as WSNs, beside the fact that most system software for WSNs fails to support dynamic component reconfiguration.

Recently, we proposed a novel distributed middleware approach, called WISEKIT, for addressing the dynamicity of WSN applications [9]. WISEKIT provides an abstract layer accelerating development of adaptive WSN applications. As this middleware supports adaptation of component-based software, we consider, in this paper, how the component development model can improve the performance of adaptation, besides the efficiency achieved by the WISEKIT middleware. In particular, we consider how REWISE, our software component model for adaptive WSN applications, can minimize the overhead of software reconfiguration. The notion of behavior reconfiguration in REWISE makes it possible to upload only the changed part of a

component to the sensor node, instead of uploading the whole software component [8]. We believe that this model can be also exploited in other application areas, such as mobile applications and embedded systems.

The rest of paper is organized as follows. In Section 2, we demonstrate a motivating application illustrating the requirements for adaptation in WSNs. To propose the new development paradigm, we first gives a short introduction to the WISEKIT middleware in Section 3, then we present our approach based on the concept of REWISE component model in Section 4. Next, in Section 5, we demonstrate the efficiency of using REWISE in a sample reconfigurable application running on CONTIKI operating system [10]. Related work is presented in Section 6. Finally, Section 7 concludes this paper and identifies some future work.

2. MOTIVATING SCENARIO

In this section, we present an application scenario in the area of home monitoring to further motivate our work. Most of the earlier efforts in this field employed a high-cost wired platform for making the home a smart environment [11,12]. Future home monitoring applications are characterized as being filled with different sensor types to observe various types of ambient context elements, such as temperature, smoke, and occupancy. Such information can be used to reason about the situation and interestingly react to the context changes through actuators [4].

Figure 1 illustrates a hypothetical *context-aware home*. Each room is equipped with the relevant sensor nodes according to its attributes and uses. For instance, in the living room three "occupancy" sensors are used to detect the movement, one sensor senses the temperature, and one smoking sensor detects the fire in the room. Although each sensor is configured according to the preliminary requirements specified by the end-user, there may happen some predictable or unpredictable scenarios needing behavioral changes in sensor nodes. Basically, these scenarios can be considered from two different aspects: *i*) application-level, and *ii*) sensor-level. The former refers to the contextual changes related to the application itself, *e.g.*, according the end-user requirements for the living room, if one of the occupancy nodes detects a movement in the room, the temperature nodes should stop sensing and sending their measurements. The latter further concerns with the capabilities and limitations of a particular sensor node, *e.g.*, if the residual energy of temperature sensor is lower than a pre-defined threshold, the aggregated data should be delivered instead of sending all individual sensor readings.

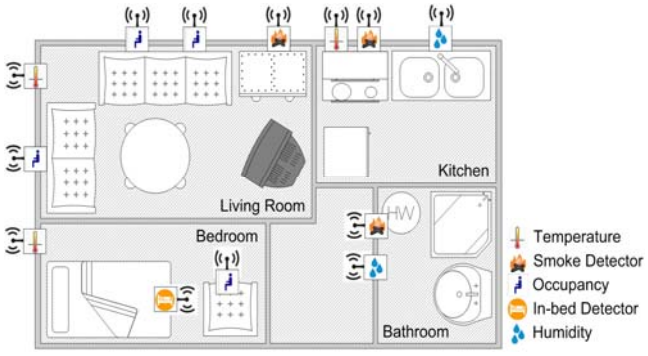

Figure 1. Description of the home monitoring system.

Besides the above concerns, the recent requests for *remote home monitoring*, which enables the owner to check periodically the home state via a web interface, are being extended by the request of *remote home controlling*. This need also brings some other new challenges in terms of dynamicity and makes the issue of adaptivity more significant.

Considering statically all above concerns becomes quite impossible when many of these scenarios should be supported simultaneously by the application on a resource-limited node. Moreover, at the same time, the relation between the context elements needs to be maintained in order to reason timely on a change. Obviously, supporting all these requirements during the application run-time needs: *i)* an abstract middleware approach to address the dynamicity and adaptivity needs, and *ii)* a new way of constructing application modules with regards to the adaptation needs and limitations of WSNs.

3. WISEKIT IN A NUTSHELL

WISEKIT is a novel distributed component-based middleware approach for addressing the dynamicity of WSN applications [9]. WISEKIT provides an abstract layer accelerating development of adaptive WSN applications. Using this middleware, the developer focuses only on application-level requirements for adaptivity, while the underlying middleware services expose off-the-shelf APIs to formalize the process of adaptive WSN application development and to hide the complexity of the technical aspects of adaptation.

As the sensor nodes are mostly organized in a hierarchical way [13], WISEKIT is distributed among nodes according to the level of hierarchy of a particular node. *Hierarchical adaptation* is based on the idea of placing adaptation services according to: *i*) the scope of information covered by a particular node, and *ii*) the resource richness of that specific node.

Figure 2 presents WISEKIT in the sensor node, where the fine-grained application reconfiguration takes place. WISEKIT in the sensor node contains services for: *i)* updating the value of components' parameters based on the local adaptation policy (Local Reasoning), *e.g.*, changing the value of a particular parameter in Logger, *ii)* receiving adaptation request from cluster head (Adaptation Proxy), *iii)* temporarily storing the new component's image (Component Repository), and *iv*) loading new component, reloading or removing a running component (Component Reconfigurator), *e.g.*, replacing the Sampler component with a newer one. This service is considered as the key part of our middleware in sensor node because the performance of WISEKIT depends highly on how efficient the component reconfiguration task is carried out.

In addition to the mechanism for component reconfiguration, the *degree of component reconfigurability* has also significant impact on the performance of adaptation. By degree of reconfigurability, we mean that to which level of application's assembly a change can be considered. In the worst case, this level includes the whole application, while the best cases are different. We may be able to minimize the update to some lines of code or even to a statement, however, at the same time, the mechanism for performing such an update may become quite difficult and impractical. Therefore, there is a trade-off between the degree of reconfigurability and the complexity of reconfiguration mechanism. In the rest of this

paper, we discuss how the WISEKIT application's components should be structured in order to optimize the adaptation cost.

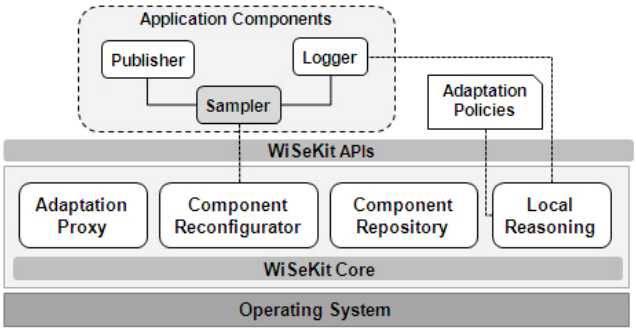

Figure 2. WISEKIT services within sensor node.

4. THE REWISE COMPONENT MODEL

As mentioned above, application development paradigm brings its own efficiency metrics in terms of adaptation needs. In this section, we briefly our new software component model, called REWISE, for WSN applications, which enables lightweight adaptation of software components [8]. The main idea behind REWISE is to consider the application as an integration of a set of business components so that each potentially reconfigurable service of a component is implemented as a separate component, instead of implementing it as a "method". ReWiSe is partially inspired by the dynamic module system of OSGi [5] and the way service references are dynamically resolved. In contrast to OSGi, which enables coarse-grained reconfiguration of application bundles, we aim at offering a fine-grained model of component reconfiguration, which is tailored for resource-limited embedded systems, such as WSNs.

Figure 3 illustrates the constituents of the REWISE component model. Like other popular component models, the interaction of REWISE with other components is established through interfaces, receptacles, events, listeners, and properties. In fact, the outer white box is similar to what has been described in the previous models [6,7]. The main dissimilarity is in the implementation of component's interfaces. Particularly, in this model an interface is not implemented as a method within the component body, but it is implemented in a separate component containing "just" the implementation of that interface, and no more functionalities. Let us call these kind of components TinyComponent. Therefore, for each interface of the main component, we have a corresponding TinyComponent implementing that interface. The main component is wrapped with *Interface Interceptor Wrapper* to route the interface calls of other components to the corresponding TinyComponent of an interface.

As REWISE conforms to the standards of a typical software component model, developers can use both REWISE and other existing component models depending on the component type (adaptive or not adaptive).

4.1 Concept of TinyComponent

A TinyComponent represents an interface implementation. For each interface of a component, there is a corresponding TinyComponent containing just one method that realizes the interface. The interaction of TinyComponent with other elements inside the component brings new technical issues, while its connection with the world outside the main component occurs in the same fashion as previous models.

Firstly, unlike direct interface implementation in the main component, which has easy access to the main component variables, TinyComponent is located outside the scope of the main component, therefore a TinyComponent is not able to reach to the variables of the main component. This problem is resolved by passing a reference of the main component to the TinyComponent, thereby the scope is reachable for TinyComponent through a variable containing a pointer to the main component.

Second question that arises is how TinyComponents can interact among themselves, like what occurs between methods in common component models. Like the first problem, this case is originated from the fact that the scope of the main component is not accessible from the TinyComponent. Similarly to the resolution of the first question, the reference of the main component passed to the TinyComponent can give access to every thing inside the main component, namely variables and interfaces. In the next section, we discuss in more details how *Interface Interceptor Wrapper* facilitates such a calling among TinyComponents.

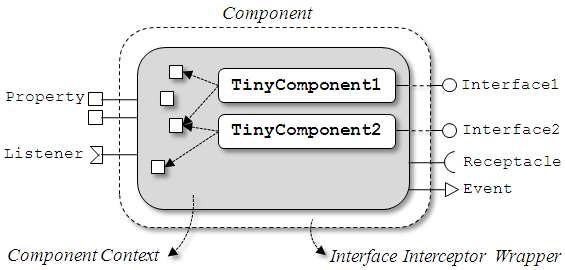

Figure 3. REWISE Component Model.

4.2 Concept of Interface Interceptor Wrapper

To systematically access TinyComponents, REWISE should be enhanced with a mechanism capable to route calls from a specific interface to the corresponding TinyComponent. Since TinyComponents are going to become the new candidate for replacement in the forthcoming reconfiguration mechanism, a component cannot maintain the name of a TinyComponent in a hardcoded manner. Therefore, the *Interface Interceptor Wrapper* is responsible for handling dynamically the references to TinyComponents. The first time a service is executed, the wrapper reads from its mapping configuration file the name of the TinyComponent implementing the service and then caches the reference to the corresponding TinyComponent in its local data. Afterward, other requests for that service will be automatically forwarded to the assigned TinyComponent. The configuration file contains a set of structured data identifying the name of the corresponding TinyComponent for each interface. Moreover, the wrapper is responsible for passing the reference of the main component to the TinyComponent.

4.3 Concept of Component Context

A major challenge in reconfiguration mechanisms is how to preserve the state of a component during the reconfiguration period. Since the unit of replacement in REWISE is the stateless TinyComponent, the replacement candidate has not any state to miss. In fact, the state of the component is preserved in the main component. *Component Context* is an abstract concept indicating the current values of all private and public member variables in the main component. The context is accessible for TinyComponent

through the main component reference passed to the TinyComponent method.

4.4 Support for Reconfiguration

Basically, the three concepts of REWISE ensures that components have a high degree of reconfigurability.

Firstly, the notion of TinyComponent makes the behavioral-level configuration possible; thereby if a portion of component (interface implementation) needs to be updated we do not need to change the full component image. As communication between nodes is *the main source of energy* consumption in WSNs, minimizing the size of update code can considerably reduce the cost of adaptation. The other advantage of using TinyComponent is that the safe state can be determined only by checking the interactions of the candidate TinyComponent with others rather than checking the interactions of the whole main component.

Secondly, the interface interceptor wrapper undertakes the action of switching from the old TinyComponent to the new one. As mentioned before, a mapping configuration file is attached to the wrapper to specify the map between each interface and its corresponding TinyComponent. Note that if the name of the new TinyComponent is the same as previous, the mapping file remains unaltered.

The final improvement is offered by component state management mechanism. In fact, in previous component models for WSNs, the stateful main component is subject to replacement, while in REWISE the stateless TinyComponent becomes the new replaceable unit. Consequently, component variables maintain their values (component context) during the component lifetime.

5. PRELIMINARY EVALUATION

In this section, we demonstrate the efficiency of using REWISE in a reconfigurable WSN application. As mentioned before, this application is run atop the CONTIKI operating system [10].

CONTIKI is a lightweight operating system for sensor nodes with support for dynamic loading and replacement of individual programs and services. CONTIKI provides facilities for loading and running native code of a module dynamically. In particular, each loadable module in CONTIKI is in *Compact Executable and Linkable Format* (CELF) containing code, data, and reference to other functions and variable of system. When a CELF file is loaded to a node, the *dynamic linker* in the core resolves all external and internal references, and then writes the code to ROM and the data to RAM [10].

Let us consider the sample component configuration depicted in Figure 4. We assume that after some time of running application, the `Sampler` component needs to be replaced with the new `Sampler` in which only the implementation of `IReport` interface is changed. Note that `Sampler` is not a REWISE component in this case. Main tasks for performing reconfiguration include: *i*) checking the component to ensure that it is not in interaction with the `Logger` and `Publisher` components before starting reconfiguration, *ii*) saving the state of the component, and *iii*) initializing the new `Sampler` component with the state of the previous `Sampler`.

Our preliminary evaluation includes the comparison of situations depicted in Figure 4 (common approach) and Figure 5 (REWISE approach) in terms of energy consumption for the initial reconfiguration tasks.

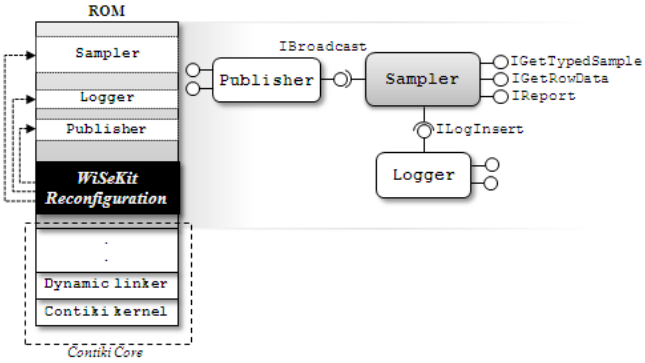

Figure 4. Sample Configuration.

In the common approach, `Sampler`, `Logger`, and `Publisher` are the three components run in ROM as well as the WISEKIT reconfiguration service. Each program's data is also located in RAM. For updating `Sampler` program, the new Sampler_CELF file (2486 bytes) must be transferred to the node by the underlying communication protocol, and then copied into an EEPROM. Next, WISEKIT reconfiguration service performs the three main tasks mentioned in the previous section. Finally, the dynamic linker links, relocates and loads the new `Sampler` code (1364 bytes) into flash ROM.

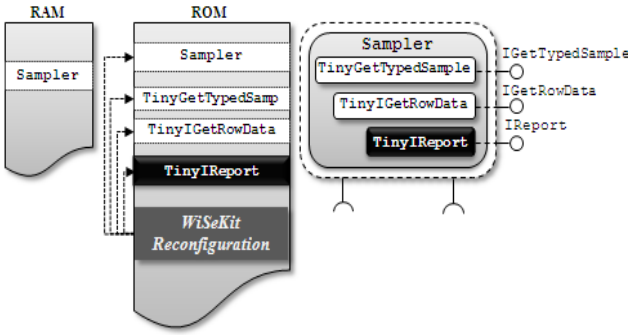

Figure 5. REWISE-based configuration.

As the energy consumption depends on the size of new component, the model of energy consumption will be:

$$E = s_{New\_CELF} \times \left(P_p + P_s + P_l\right) + s_{New\_Sampler} \times P_f + E_{rec}$$

where s_{new_CELF} is the size of new CELF file and P_p, P_s, P_l and P_f are scale factors for network protocol, storing binary, linking and loading, respectively. $s_{New\_Sampler}$ is the code size of new `Sampler`, and E_{rec} is the energy cost of performing reconfiguration. Thus,

$$E = 2486 \times \left(P_p + P_s + P_l\right) + 1364 \times P_f + E_{rec}$$

However, in the REWISE approach, `TinyIReport` becomes the candidate for replacement. Therefore, in this case, the TinyIReport_CELF file (764 bytes) must be transferred to the node, and all mentioned tasks for dynamic loading must be done for the `TinyIReport` component (its code size is 348 bytes).

Note that the overhead of configuration will be reduced in this case because: *i*) safe state for reconfiguring `TinyIReport` is checked with less cost, *ii*) the state preservation step will be skipped, and *iii*) the size of the update code is smaller.

$$E_{new} = 764 \times \left(P_p + P_s + P_l\right) + 348 \times P_f + E_{new\_rec}$$

$$\frac{E_{new}}{E} \approx 0.25 \qquad E_{new\_rec} < E_{rec}$$

Thus, for the given example, the energy consumption in the REWISE model is roughly 75% less than the common model. It is because the size of `TinyIReport` is much smaller than the size of `Sampler`. Basically, the efficiency of using the new model depends on the ratio of the TinyComponent size to the main component size. For smaller values, our approach is more efficient, and for larger values the performance of the proposed model relies on the efficiency of the reconfiguration mechanism of WiSeKit.

6. RELATED WORK

Component Object Model (COM) [14], *Enterprise JavaBeans* (EJB) [15], and the *CORBA Component Model* (CCM) [16] are the most well-known component models in distributed system area. Unfortunately, all these models do not support inherently dynamic reconfiguration of components, because these models provide the basic building blocks for component-based software, and the core design aspects of such models do not consider features, such as reconfigurability of component.

In the scope of reconfigurable component models, FRACTAL has been known as a pioneering model for dynamic software applications [6]. Although FRACTAL is a comprehensive and extensible model for component composition and assembly, its minimal core is a heavy-weighted extensible unit with various features suitable for the large-scale applications needing different degree of reconfiguration for different software granularity. In fact, concepts in REWISE and FRACTAL are different to some extent. For instance, the content part of Fractal is devoted for handling component compositions, while in REWISE we adopt component context as a means to ease the state management during the reconfiguration. Likewise, what has been proposed in OPENCOM component model [7] is a coarse-grained mechanism for dealing with dynamicity in applications.

The first prominent work reported to address reconfigurability for resource-constrained systems is [17]. In this paper, Costa et al. propose a middleware component framework for embedded systems in the context of the RUNES project [22]. Their approach focuses on a kernel providing primary services needed in a typical resource-limited node. Specifically, their work supports customizable component-based middleware services that can be tailored for particular embedded systems. In other word, the middleware itself can be reconfigured, while our proposal tries to give this ability to the application services through underlying middleware services.

Efforts for achieving adaptivity in WSNs have continued by Horr et al [18]. They proposed DAVIM, an adaptable middleware enabling dynamic service management and application isolation. Particularly, their main focus in this work is on the composition of reusable services in order to meet the requirements of concurrently running applications. In fact, they consider the adaptivity from the view of dynamic integration of services, whereas our work tries to make the application services adaptable.

A FRACTAL composition-based approach for constructing and dynamically reconfiguring WSN applications is introduced in [19]. The approach uses π–calculus semantics to unify the models of interaction for both software and hardware components. The novel feature of that approach is its support for a uniform model of interaction between all components, namely communication via typed channels. Although the proposed reconfiguration model is promising, it fails to explain under which conditions a reconfiguration should take place.

An other relevant work in the context of component-based reconfiguration for WSN has recently been reported under the name of the FIGARO framework [20] as an approach for WSN reconfiguration in the RUNES project [22]. The main contribution of FIGARO is to present an approach for determining *what* should be reconfigured *and* where the reconfiguration should take place. The former one is related to runtime component replacement, and the latter is concerned with which nodes in the network should receive an updated code. Although we believe this work provides a promising and tangible way to achieve component reconfiguration for WSNs, we aims at abstracting the reconfiguration mechanism as a middleware level service enhanced with some extra features not mentioned in FIGARO such as policy-based adaptation and distributed adaptation. The component model of FIGARO also fails to support lightweight reconfiguration, which is addressed in our approach by REWISE component model.

7. CONCLUSION AND FUTURE WORK

This paper presented a new way of developing application for WSNs that takes into consideration dynamicity and adaptivity. To this end, we proposed a new development paradigm for efficient dynamic software reconfiguration in the resource-limited networks, such as WSNs. This paradigm is based on a new component model introducing the notion of TinyComponent to achieve a lightweight behavioral-level reconfiguration for WSNs. Using this component model, not only the consistency of reconfiguration is guaranteed, but also the reconfiguration is carried out with a minimum overhead, because instead of updating the whole component, a particular service of the component can be reconfigured.

We are currently focusing on the home-monitoring application as a motivating scenario. In this paper, we described this application briefly without detailing application structure and obtaining its REWISE and non-REWISE components. This application will be analyzed, designed, and implemented based on the concepts and principles we presented for both middleware and application layers. The other open issues regarding the REWISE component model, such as safe state for reconfiguration, has been planned for our future work. The work reported in this paper is a part of our comprehensive solution for self-management in WSNs. Integrating this work with the other work reported in [9,21] is another future direction.

Acknowledgments

This work was partly funded by the Research Council of Norway through the project SWISNET, grant number 176151.

8. REFERENCES

[1] Puccinelli, D., and Haenggi, M., 2005. Wireless sensor networks: applications and challenges of ubiquitous sensing.

IEEE Circuits and Systems Magazine, vol. 5, no. 3, 19-31.

[2] Sohraby, K., Minoli, D., Znati, T., 2007. wireless sensor networks, technology, protocols, and applications. Wiley-Interscience.

[3] Wang, Q., Zhu, Y., and Cheng, L., 2006. Reprogramming wireless sensor networks: challenges and approaches. IEEE Network, 20(3), 48–55.

[4] Akyildiz, I., Kasimoglu, I., 2004. Wireless sensor and actor networks: research challenges, Ad Hoc Networks 2 (4), pp. 351-367.

[5] Open Services Gateway Initiative. http://www.osgi.org

[6] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B., 2006. The FRACTAL component model and its support in Java. Softw., Pract. Exper. 36(11-12): 1257-1284, http://fractal.objectweb.org

[7] G. Coulson et al., 2008. A generic component model for building systems software. ACM Trans. Computer Systems, pp. 1–42, Feb.

[8] Taherkordi, A., Eliassen, F., Rouvoy, R., e-Trung, Q.: ReWiSe: A New Component Model for Lightweight Software Reconfiguration in Wireless Sensor Networks, In: Proc. of 7th IWSSA, LNCS vol. 5333, 415-425, Monterrey, Mexico (2008)

[9] Taherkordi, A., Le-Trung, Q., Rouvoy, R., Eliassen, F., 20089. WISEKIT: A Distributed Middleware to Support Application-level Adaptation in Sensor Networks, The 9th IFIP international conference on Distributed Applications and Interoperable Systems (DAIS'09), LNCS vol. 5523, 44-58, Lisbon, Portugal, June 9-12, 2009.

[10] Dunkels, A., Finne, N., Eriksson, J., and Voigt, T, 2006. Run-time dynamic linking for reprogramming wireless sensor networks. ACM SenSys, Colorado, USA.

[11] Mozer, M., 2004. Lessons from an adaptive home. In: D.J. Cook and S.K. Das, Editors, Smart Environments: Technology, Protocols, and Applications, Wiley, 273-298.

[12] Huebscher, M. C., and McCann, J. A. 2004. Adaptive middleware for context-aware applications in smart homes, 2nd Workshop on Middleware for Pervasive and AdHoc Computing.

[13] Le-Trung, Q., Engelstad, P., Taherkordi, A., Pham, N. H., Skeie, T., 2009: Information Storage, Reduction, and Dissemination in Sensor Networks: A Survey, In: Proc. of the IEEE IRSN Workshop, Las Vegas, US

[14] Microsoft, COM, http://www.microsoft.com/com

[15] Sun Microsystems. Enterprise Java Beans, http://java.sun.com/products/ejb/index.html

[16] OMG. CORBA, Object Management Group, http://www.omg.org

[17] Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G.P., and Zachariadis, S., 2007. A reconfigurable component-based middleware for networked embedded systems. International Journal of Wireless Information Networks, vol 14, No 2, 149-162.

[18] Horr, W., Michiels, S., Joosen, W., Verbaeten, P., 2008: DAVIM: Adaptable Middleware for Sensor Networks, IEEE Distributed Systems Online, vol. 9, no. 1.

[19] Balasubramaniam, D., Dearle, A., Morrison, R., 2008: A Composition-based Approach to the Construction and Dynamic Recon_guration of Wireless Sensor Network Applications, In Proc. of the 7th Symposium on Software Composition (SC)

[20] Mottola, L., Picco, G., and Sheikh, A. 2008. FiGaRo: fine-grained software reconfiguration for wireless sensor networks. In: Proc. of the 5th European Conference on Wireless Sensor Networks (EWSN), Bologna, Italy, LNCS vol. 4913, 286-304.

[21] Taherkordi, A., Rouvoy, R., Le-Trung, Q., Eliassen, F., 2008: A Self-Adaptive Context Processing Framework for Wireless Sensor Networks, In: Proc. of the 3rd ACM MidSens in conjunction with Middleware'09, Belgium, pp. 7-12

[22] http://www.ist-runes.org/