WiSeKit: A Distributed Middleware to Support Application-Level Adaptation in Sensor Networks

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

[1] University of Oslo, Department of Informatics
P.O. Box 1080 Blindern, N-0314 Oslo
{amirhost,quanle,rouvoy,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. Applications for *Wireless Sensor Networks* (WSNs) are being spread to areas in which the contextual parameters modeling the environment are changing over the application lifespan. Whereas *software adaptation* has been identified as an effective approach for addressing context-aware applications, the existing work on WSNs fails to support context-awareness and mostly focuses on developing techniques to reprogram the whole sensor node rather than reconfiguring a particular portion of the sensor application software. Therefore, enabling adaptivity in the higher layers of a WSN architecture such as the middleware and application layers, beside the consideration in the lower layers, becomes of high importance. In this paper, we propose a distributed component-based middleware approach, named WiSeKit, to enable adaptation and reconfiguration of WSN applications. In particular, this proposal aims at providing an abstraction to facilitate development of adaptive WSN applications. As resource availability is the main concern of WSNs, the preliminary evaluation shows that our middleware approach promises a lightweight, fine-grained and communication-efficient model of application adaptation with a very limited memory and energy overhead.

Keywords: wireless sensor networks, distributed middleware, adaptation, reconfiguration.

1 Introduction

The challenges for application development on WSNs are becoming as prominent as the issues concerning sensor hardware, network architecture, and system software. It is because the new emerging applications for WSNs do not limit themselves to a single function called *"sense and send"* with trivial local data

processing tasks [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 based on the context information surrounding them [3].

In such an environment, in addition to the basic tasks, an application needs to adapt its behavior to cope with changing environmental conditions, and different capabilities of each individual sensor node in the network. As different nodes are expected to run different tasks, software with dynamic adaptable functionalities becomes an essential need. Moreover, for applications deployed to a high number of nodes in inaccessible places, individual software updating becomes an impractical and inefficient solution.

As the common types of sensor nodes are still suffering from resource scarceness, researchers have not been willing to consider the application code on sensor node as adaptive software. This is because, on one hand, the typical adaptation frameworks mostly come with a high level of complexity in the reasoning process and reconfiguration mechanism. On the other hand, most of the software development paradigms for WSN application are not able to support reconfigurability due to the lack of modularity, such as in the case of script programming. Moreover, the lack of operating system level support for dynamic reconfiguration is the other critical challenge in the way of achieving application-level adaptivity in WSNs. Recently, operating systems, such as Contiki [4], have considered this issue by supporting dynamic binding and loading of software components.

A few works have been reported in the literature that address adaptation for embedded and sensor systems. In [6,7,8], the main contribution is to provide adaptivity at the middleware-level (not application-level) in order to make the network-level services reconfigurable and replaceable. In [9], a small runtime support is proposed over Contiki to facilitate dynamic reconfiguration of software components. Although it promises to achieve application-level reconfigurability, the level of abstraction is low and it does not propose a general framework supporting all essential aspects of application adaptation. In fact, it plays the role of component reconfiguration service in a typical adaptation framework.

The performance of the adaptation middleware depends on two major factors. The first is the *reconfigurability degree* of software modules. In a highly reconfigurable model, the update is limited to the part of the code that really needs to be updated instead of updating the whole software image. We term this feature *fine-grained reconfiguration*. The second is the mechanism by which a module is reconfigured. In this paper, we concentrate on the latter, whereas the former has been discussed in [5] by proposing a new component model, called *ReWiSe*, for lightweight software reconfiguration in WSNs.

In this paper, we present a novel distributed middleware approach, named WiSeKit, for addressing the dynamicity of WSN applications. 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 hide the

complexity of the technical aspects of adaptation. The adaptation decision logic of WiSeKit is also inspired from the hierarchical architecture of typical WSNs in order to achieve the adaptation goals in a light-weight and efficient manner.

The rest of this paper is organized as follows. In Section 2, we demonstrate a motivating application scenario. The basic design concepts of our middleware proposal are described in Section 3. In Section 4, the WiSeKit middleware is proposed with a preliminary evaluation presented in Section 5. Related work is presented in Section 6. Finally, Section 7 concludes the paper and gives an outlook on future research.

2 Motivating Application 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 [10,11]. 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 through actuators [3].

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 for detecting 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 tasks. 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.

Besides the above concerns, the recent requests for *remote home monitoring*, which enables the owner to check periodically 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 you have many of these scenarios that should be supported simultaneously by the application on a resource-limited node. Moreover, at the same time you need to maintain the relation between the context elements and reason timely on a change. Obviously, supporting all these requirements during application runtime needs an abstract middleware approach to address the dynamicity and adaptivity challenges w.r.t. the unique characteristics of WSNs.

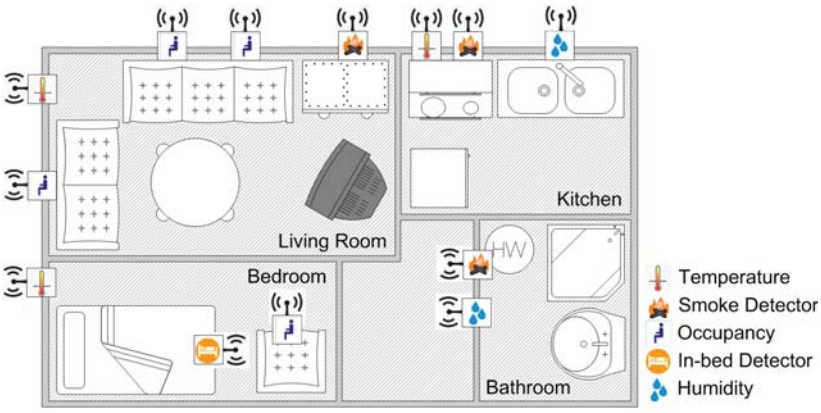

Fig. 1. Description of the home monitoring system

3 Basic Design Concepts

In this section, we describe the basic design concepts of WiSeKit middleware.

Adaptation Time. Basically, adaptation can be performed in two manners: *statically* and *dynamically*. Static adaptation relates to the redesign and reconfiguration of application architectures and components before deployment of the application. Dynamic adaptation is performed at application run-time due to the changing needs, resource and context conditions. We adopt dynamic adaptation in our proposal because most WSN applications are expected to work seamlessly for a long time and mostly deployed in inaccessible places.

Adaptation Scope. Two popular adaptation mechanisms are introduced in the literature [12], [13]: *parameter adaptation* and *component adaptation*. Parameter adaptation supports fine tuning of applications through the modification of application variables and deployment parameters, while component adaptation allows the modification of service implementation (replacement of component), adding new components, and removing running components. We explain later in this paper why and how our middleware supports both of these mechanisms.

Adaptation Policy. In general, there are three different approaches for identifying a policy: situation-action rules [13,14], goal-oriented [15] and utility-based [16]. The two latter techniques represent high level forms of policies, while the former specifies exactly what to do in given situations. As the adaptation policies of most WSN applications can be described easily through a set of conditional statements, WiSeKit follows the situation-action rules approach. Situations in our proposal are provided from the framework we proposed in [17]. This framework proposes a *context processing model* to collect data from different sources (environment, application, sensor resources, and user) and process them for the use of adaptation reasoning service.

Fine-grained Reconfiguration. Adaptation reasoning specifies through which mechanism (either parameter-based or component-based) which parts of the application should be reconfigured. As the major cost of reconfiguration in WSNs is in transferring the new update code across the network, fined-grained reconfiguration becomes very important. Note that fine-grained reconfigurability should be supported by the underlying system software.

Hierarchical Adaptation. As the sensor nodes are mostly organized in a hierarchical way [18], our proposal distributes the adaptation tasks 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 node.

4 WiSeKit Adaptation Middleware

WiSeKit aims to provide a set of APIs at the middleware level of WSNs in order to make an abstraction layer that formalizes and simplifies the development of adaptive WSN applications. In general, WiSeKit is characterized by the following features:

- *Local-adaptivity*: an application running on the sensor nodes has the possibility of identifying its adaptation policies. The APIs exposed at the middleware layer are able to read the policy object and maintain the application components' configuration according to the context information gathered periodically from both sensor node and application.
- *Intermediate-observation*: using WiSeKit, we can specify adaptation requirements for a region of the network, *e.g.*, a floor or a room in a building. At this level, we can specify high-level adaptation policies through WiSeKit APIs provided at more powerful nodes such as cluster head or sink node.
- *Remote-observation*: the end-user or the agent checking the application status locally through sink interface or remotely via a web interface might need to specify his/her own requirements regarding adaptation.
- *Component-based reconfiguration*: updates in WiSeKit can take place both at component attribute level and at component level. WiSeKit expects application developers implement predefined interfaces for components which are subject to reconfiguration. We present later in this section the signature of such interfaces and the mechanisms for reconfiguration.
- *Distribution*: The heterogeneity of WSNs in terms of the node's resource capabilities and functionalities necessitates support for distribution at the middleware layer in order to achieve the above goals and also optimize network resources usage. WiSeKit is within all nodes types built up over a set of *Core Services* which provides an infrastructure for distribution.

Figure 2 illustrates the complete logical architecture of the WiSeKit middleware distributed over the different node types. It shows how the adaptation

services are located in different node types and mapped to a typical WSN architecture. At the left side of the figure, sensor node features a set of services for realizing *Local-adaptivity* and *Component-based reconfiguration*. Next to the sensor nodes, the cluster head has the responsibility of *Intermediate-observation* to observe data and analyze it in terms of adaptation required within the scope of a cluster. Finally, at the right side of Figure 2, WiSeKit in the sink node is able to retain the "whole" WSN application in a high degree of adaptivity via *Intermediate-observation* and *Remote-observation*. Therefore, the end user of the application can specify his/her own adaptation needs through the APIs provided within the sink node. Middleware services in different nodes interact through core services customized for each type of node. The details of WiSeKit services within each type of node are explained in the rest of this section.

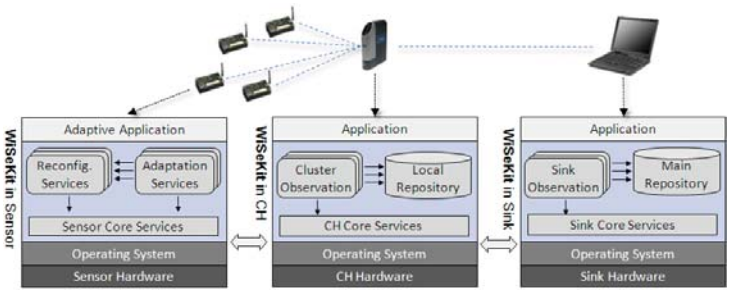

Fig. 2. WiSeKit in the hierarchical WSN architecture

4.1 Sensor Side

To address the middleware requirements of adaptive applications, we need first to explore the desired structure of an application deployed over the sensor nodes, then the adaptation middleware services will be discovered accordingly. Figure 3 illustrates a sample configuration of application components for a home monitoring application. There are three main aspects that should be considered for application development.

Firstly, the components which are subject to reconfiguration should implement the relevant interfaces. In general, four types of reconfigurations are likely to happen during the application runtime, including: *i)* replacing a component with a new one, *ii)* adding a new component, *iii)* component removal, and *iv)* changing the values of component member variables. For each type of reconfiguration the relevant interface(s) should be implemented. We explain later in this section the name and specification of those interfaces.

Secondly, as shown in the figure, the deployable package should include a predefined *policy file* describing situation-actions rules. It is one of the main sources of *local* adaptation decision. The local decision is limited to changing the values of component member variables, while the decision of full component image replacement is made by the cluster head. It is because the decisions for replacing or

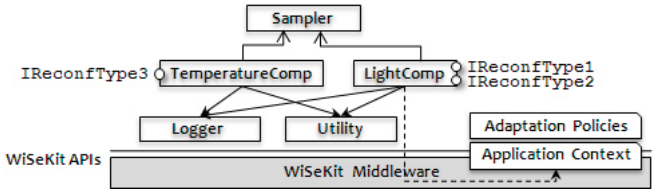

Fig. 3. A sample component configuration for an adaptive home application

adding components fall in the category of heavyweight reconfiguration requests. Such a decision should be assigned to a node being aware of what happens in the scope of a cluster. Moreover, sometimes we need to send a component image to a set of sensor nodes having similar attributes or responsibilities.

Finally, *Application Context* is a meta-data reporting application-related contextual information. Application components can update it when a contextual event is detected through APIs provided by WiSeKit. The content of application context is used together with sensor context information against the situations described in the policy object to check whether any adaption is needed.

Figure 4 describes the architecture of our adaptation middleware for sensor nodes. As shown, WiSeKit addresses three main areas of adaptation concern.

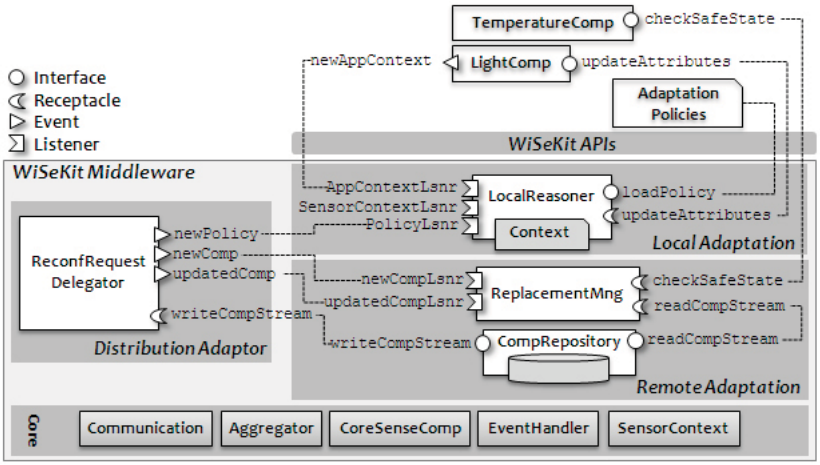

Fig. 4. WiSeKit services in the sensor node

Local Adaptation is in charge of carrying out local parameter adaptation requests. LocalReasoner, as the main service receives both the adaptation policies of the application and context information, then it checks periodically the situations within policy file against application context and sensor context for adaptation reasoning. Upon satisfying a situation, the corresponding action,

changing the values(s) of component attribute(s), is performed via calling the `updateAttributes` interface of the component and passing the new values.

Remote Adaptation is concerned with adapting the whole component. In fact, the corresponding cluster node performs the reasoning task and notifies the sensor node the result containing which component should be wholly reconfigured. The key service in this part is ReplacementMng. Upon receiving either `newComp` or `updatedComp` event, it reads the component image from CompRepository, loads the new component and finally removes the image stored by CompRepository from the repository.

After loading the component image, the current component is periodically checked to identify whether it is in a *safe state of reconfiguration* or not. The safe state is defined as a situation in which all instances of a component are temporarily idle or not processing any request. There are several safe state checking mechanisms in the literature [19], [20]. In some solutions, safe state recognition is the responsibility of the underlying reconfiguration service, whereas in the other mechanisms this checking is assigned to the component itself. We adopt the second method because of its low overhead. Therefore, WiSeKit expects from each reconfigurable component to implement the `checkSafeState` interface.

Distribution Adaptor provides a distribution platform for adaptation decision and accomplishment. Specifically, it is proposed to address three issues: *i)* the possibility of updating adaptation policies during application lifespan, *ii)* receiving the result of high-level adaptation decision from cluster head, *i.e.*, the image of a component, and *iii)* providing an abstraction for distributed interactions between WiSeKit services. ReconfRequestDelegator reads the data received through the Communication service, checks whether it encompasses any event such as new policy, new component, or updated component, and finally unmarshals the content and generates the corresponding event object.

The bottom part of middleware is decorated with the *core* to provide an infrastructure for distribution as well as the utility and common services. The Communication service has the responsibility of establishing connection to the other nodes in the hierarchical structure. This service not only sends the data, but also receives the reconfiguration information (component image or policy). Aggregator is a service for performing aggregation of data received from CoreSenseComp. EventHandler handles events generated by the services within the middleware. The context information related to the sensor hardware and system software is reported by SensorContext service as an `newSensorContext` event.

4.2 Cluster Head Side

Based on *hierarchical adaptation*, when the context information of a sensor node is not sufficient to make an adaptation decision, the cluster head attempts to decide on an adaptation based on the data received from sensor nodes in its own cluster. Similarly, if the current information in the cluster head is not enough for the adaptation reasoning, the final decision is left to the sink node, *e.g.*, in our motivation scenario, if the occupancy sensors detect a movement in the living room, the cluster head notifies the temperature sensors to reduce the sampling

rate. Figure 5 illustrates both the structure of WSN application and the WiSeKit architecture over the cluster head. The WiSeKit services within the cluster head make the high-level adaptation decisions through processing *application context model* and *cluster-level adaptation policies*.

The context model defines the possible contextual situations by describing the relations between the application data gathered by sensor nodes [17]. For example, "room occupied" is a situation that can be deduced from checking the data values of both occupancy sensors and light sensors in a room. The cluster-level adaptation policies are described in the same way as for sensor nodes (situation-action). However, in this case, the situations are those defined in the context model. The action also include loading a new policy or a new component in some selected nodes.

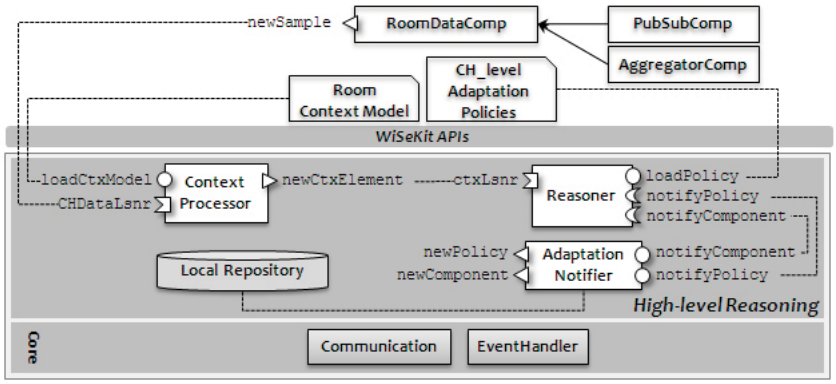

Fig. 5. WiSeKit in the cluster head

As depicted in Figure 5, WiSeKit aims at addressing the high-level reasoning issues within the cluster head. To this end, the middleware services expect from the application to provide: *i)* the context model, and *ii)* the adaptation policies. In this way, WiSeKit processes at first the context model along with the data received from sensor nodes in order to find out the current context situation(s) of environment, then the Reasoner service checks whether any adaption is needed. In fact, this service analyzes the adaptation policies based on the current context information, thereby it decides on update notifications, *i.e.*, either new policy or new component. If Reasoner makes the decision of a component update, AdptationNotifier loads the binary object of new component from the *Local Repository* and multicasts it along with the required meta-information to the nodes in its vicinity. AdaptationNotifier is also responsible for forwarding the adaptation requests of the sink node to the cluster members. We assume that the local repository of cluster head contains all versions of a component that might be needed during the application lifespan.

4.3 Sink Side

WiSeKit in the sink node is designed in the same way as it is proposed for the cluster head. The main differences between the sink node and the cluster head in the context of our middleware are in two aspects. Firstly, the scope of network covered by the sink node is the whole sensor network, while the cluster head has only access to the information retrieved within a cluster. Therefore, the global adaptation, the highest level of adaptation decision, takes place only in the sink node, where it has access to the status of all clusters. Secondly, the sink node is able to receive end-user preferences regarding to the adaptation requirements.

The component repository within the sink node contains all versions of all components. As the sink node is a powerful node with sufficient resources for processing tasks and storing application components, WiSeKit in the sink node has the ability of reasoning on the sophisticated adaptations and providing different versions of a component according to the adaptation needs.

The communication service within the core of sink provides the following functionalities: *i)* global context information exchange between the sink and the cluster heads, *ii)* code distribution, and *iii)* internetworking between WSNs and external networks, *e.g.*, the Internet. While the context information can be piggybacked into either the code distribution or routing protocols to reduce the signaling overhead, the internetworking provides more flexible facilities to manage and control WSNs remotely.

5 Preliminary Evaluation

As our adaptation middleware is customized for each type of node, the evaluation should take into account many performance figures. At first, we need to evaluate each type of node separately, then the effectiveness of WiSeKit should be assessed for all nodes together. As considering the evaluation for all nodes is a huge work, this paper focuses only on middleware performance in the sensor node as the critical part of our proposal, while evaluating the whole adaptation middleware is a part of our future work.

The efficiency of our approach for sensor node can be considered from the following performance figures:

- The *memory overhead* of middleware run-time, with respect to both program and data memory. The former can be evaluated by measuring the size of binary images after compilation. The latter includes the data structures used by the programs.
- The *energy usage* during adaptation, which refers to the energy overhead of running an adaptation task.
- The *communication overhead* between sensor nodes and cluster head in the presence of middleware for a typical adaptive application.

We chose the Instant Contiki simulator [22] to measure the overhead of memory. The prototype implementation shows the memory footprint for reconfiguration program and its data is no more than 3 Kbytes in total. As most of

sensor nodes are equipped with more than 48 Kbytes of program flash (TelosB node), WiSeKit does not impose a high overhead in terms of memory. It should be noted that this cost is paid once and for all, regardless of the amount of memory is needed for the application components. There is also an application level memory overhead for the description of adaptation policies and implementing the reconfiguration interfaces (`checkSafeState`, `updateAttributes`, etc.). This cost depends directly on the degree of application adaptivity. Moreover, the amount of memory used by CompRepository varies with respect to the number of new components downloaded simultaneously in the sensor node. As WiSeKit removes the image of a component from repository when loading it to the memory, this overhead is kept at a very low level in the order of zero.

For measuring energy consumption, we assume that our hypothetical WSN application is similar to the configuration depicted in Figure 6 and Sampler is the replacement candidate. The main reconfiguration tasks include: *i)* checking the old Sampler to ensure that it is not in interaction with the other components before starting reconfiguration, *ii)* saving the state of old Sampler, *iii)* creating the new one and transferring the last state to it.

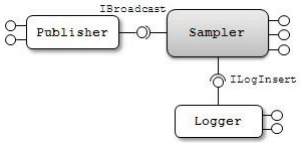

Fig. 6. Sample configuration

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 core resolves all external and internal references, and then writes code to ROM and the data to RAM [21]. For our sample configuration, the Sampler_CELF file (764 bytes) must be transferred to the node, and all mentioned tasks for dynamic loading must be done for the Sampler program (its code size is 348 bytes). As the energy consumption depends on the size of new update, the model of energy consumption will be [21]:

$$E = S_{New\_CELF} \times (P_p + P_s + P_l) + S_{New\_Sampler} \times P_f + E_{safeStateCheck}$$

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 $E_{safeStateCheck}$ is the energy cost of performing reconfiguration. Concretely, we obtain the following energy consumption for the considered adaptation scenario:

$$E = 764 \times (P_p + P_s + P_l) + 348 \times P_f + E_{safeStateCheck}$$

In this equation, we take into account the overhead of checking safe state (dependencies to the other two components). We believe that this value is very low compared to the first part, which is the reconfiguration cost imposed by Contiki.

To measure the communication overhead, we assume a scenario in the living room of home application in which the "occupancy" of context changes occasionally. According to the monitoring rules of home, when the room is empty the temperature sensors should report the temperature of room every 10 minutes. Once the room is occupied the temperature sensors should stop sensing until the room becomes empty again.

According to this scenario, when the room is occupied, ContextProcessor within the cluster head observes the new context and Reasoner notifies the relevant sensor nodes to stop sampling. WiSeKit does not impose any communication cost for context detection because it piggybacks the current value of a node's attributes at middleware layer of cluster head. Therefore, the communication cost is limited to sending policy objects to stop/restart sampling task.

Three parameters should be taken into account to measure the overhead of communication: *i)* sampling rate (r), *ii)* context consistency duration (c), and *iii)* number of context changes during a particular period of time (k). For the hypothesis scenario, if a room is occupied for two hours during one day, we have: $r = 10$ min, $c = 120$ min, and $k = 1$. In this case, the temperature sensors do not send the data for two hours, thus the number of communication for one day (24 hours) is:

$$\begin{aligned} N_{total} &= N_{forWholeDay} - N_{occupiedTime} + N_{WiSeKitOverhead} \\ &= (24 \times 60)/r - c/r + k \times N_{policySending} \\ &= 144 - 12 + 2 = 134 \end{aligned}$$

Therefore for this case the number of saved communications is 10. Generally, we can evaluate that the saved number of communications is:

$$\begin{cases} N_{saved} = c/r \times k - 2 \times k \\ 1 \leqslant k \leqslant 24/y \end{cases}$$

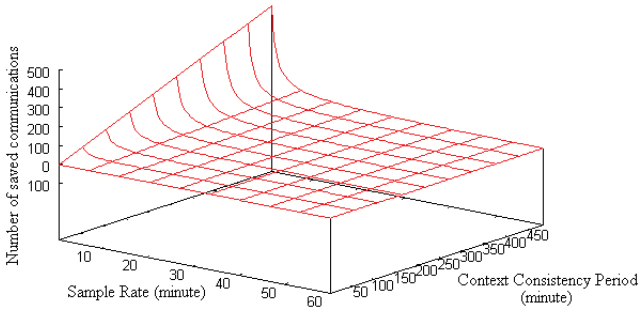

Fig. 7. Number of saved communications for a sample home monitoring scenario

Figure 7 shows the saved number of communication. As the consistency period of new context is increased and sampling rate is decreased, more number of communications will be saved. This is because the middleware prevents a sensor node to send data during the period of new context activation.

6 Related Work

The first prominent work reported to address reconfigurability for resource-constrained systems is [7]. In this paper, Costa et al. propose a middleware framework for embedded systems. 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 words, this approach enables reconfigurability at the middleware level, 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 [6]. 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 simultaneously running applications. In fact, they consider the adaptivity from the view of dynamic integration of services, whereas our work tries to make the services adaptable.

A fractal composition-based approach for constructing and dynamically reconfiguring WSN applications is introduced in [23]. 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 reconfiguration model in [23] is promising, it fails to explain under which conditions a reconfiguration should take place.

The most relevant work in the context of reconfiguration for WSN has been reported recently under the name of FiGaRo framework [9]. The main contribution of FiGaRo is to present an approach for *what and where* should be reconfigured. The former one is related to runtime component replacement, and latter is concern with which nodes in the network should receive update code. In fact, FiGaRo provides a run-time support for component loading, while our approach proposes a generic solution which includes all of-the-shelf adaptation services besides the feature of run-time component loading.

7 Conclusion and Future Work

In this paper, we proposed WiSeKit as a middleware solution making adaptation and reconfiguration of WSN application software possible. We categorized our proposal into three different layers according to the hierarchical architecture of WSN and presented WiSeKit features for each type of node. The hierarchical adaptation decision of WiSeKit conforms the hierarchical architecture of WSNs

so that based on the resource availability in a node as well as the portion of the network covered by a node, adaptation and reconfiguration are performed.

This paper focused only on adaptation for the portion of application running on sensor nodes, while the part of application deployed on cluster head and sink may need to be adapted as well. This issue will be addressed in 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 [5], [17] is another future direction. Developing a complete home monitoring application based on the proposed middleware is also included in the plan for future work.

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

References

1. Puccinelli, D., Haenggi, M.: Wireless Sensor Networks: Applications and Challenges of Ubiquitous Sensing. IEEE Circuits and Systems 5(3) (2005)
2. Costa, P., et al.: The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In: Proc. of PERCOM (2007)
3. Akyildiz, I.F., Kasimoglu, I.H.: Wireless Sensor and Actor Networks: Research challenges. Ad Hoc Networks Journal 2(4), 351–367 (2004)
4. Dunkels, A., Grnvall, B., Voigt, T.: Contiki-A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: Proc. of EmNetS-I (2004)
5. Taherkordi, A., Eliassen, F., Rouvoy, R., e-Trung, Q.: ReWiSe: A New Component Model for Lightweight Software Reconfiguration in Wireless Sensor Networks. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM 2008 Workshops. LNCS, vol. 5333, pp. 415–425. Springer, Heidelberg (2008)
6. Horré, W., Michiels, S., Joosen, W., Verbaeten, P.: DAVIM: Adaptable Middleware for Sensor Networks. IEEE Distributed Systems Online 9(1) (2008)
7. Costa, P., et al.: A Reconfigurable Component-based Middleware for Networked Embedded Systems. Journal of Wireless Information Networks 14(2) (2007)
8. Grace, P., Coulson, G., Blair, G., Porter, B., Hughes, D.: Dynamic reconfiguration in sensor middleware. In: Proc. of the 1st ACM MidSens, Australia (2006)
9. Mottola, L., Picco, G., Sheikh, A.: FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks. In: Verdone, R. (ed.) EWSN 2008. LNCS, vol. 4913, pp. 286–304. Springer, Heidelberg (2008)
10. Huebscher, M.C., McCann, J.A.: Adaptive middleware for context-aware applications in smart homes. In: Proc. of the MPAC, Canada, pp. 111–116 (2004)
11. Mozer, M.: Lessons from an Adaptive Home. In: Smart Environments: Technology, Protocols, and Applications, pp. 273–298. Wiley, Chichester (2004)
12. Poladian, V., Sousa, J.P., Garlan, D., Shaw, M.: Dynamic Configuration of Resource-Aware Services. In: ICSE, pp. 604–613. IEEE Computer Society, Los Alamitos (2004)
13. Garlan, D., et al.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
14. Lutfiyya, H., et al.: Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, p. 185. Springer, Heidelberg (2001)

15. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–52 (2003)
16. Kephart, J.O., Das, R.: Achieving Self-Management via Utility Functions. IEEE Internet Computing 11(1), 40–48 (2007)
17. Taherkordi, A., Rouvoy, R., Le-Trung, Q., Eliassen, F.: A Self-Adaptive Context Processing Framework for Wireless Sensor Networks. In: Proc. of the 3rd ACM MidSens in conjunction with Middleware 2009, Belgium, pp. 7–12 (2008)
18. Le-Trung, Q., Engelstad, P., Taherkordi, A., Pham, N.H., Skeie, T.: Information Storage, Reduction, and Dissemination in Sensor Networks: A Survey, In: Proc. of the IEEE IRSN Workshop, Las Vegas, US (2009)
19. Paula, J., et al.: Transparent dynamic reconfiguration for CORBA. In: Proc. of the 3rd International Symposium on Distributed Objects and Applications (2001)
20. Zhang, J., Cheng, B., Yang, Z., McKinley, P.: Enabling safe dynamic component-based software adaptation. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 194–211. Springer, Heidelberg (2005)
21. Dunkels, A., Finne, N., Eriksson, J., Voigt, T.: Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks. In: Proc. of ACM SenSys (2006)
22. `http://www.sics.se/contiki/`
23. Balasubramaniam, D., Dearle, A., Morrison, R.: A Composition-based Approach to the Construction and Dynamic Reconfiguration of Wireless Sensor Network Applications. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 206–214. Springer, Heidelberg (2008)