ReWiSe: A New Component Model for Lightweight Software Reconfiguration in Wireless Sensor Networks

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

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

Abstract. Wireless Sensor Networks (WSNs) are increasingly being deployed for applications with dynamic requirements. Moreover, these applications are likely to be run on nodes with different sensing parameters and capabilities. Addressing these issues in higher layers of WSNs architecture such as middleware and application, beside the consideration of lower layers, is of high importance. *Reconfiguration* of application software is an effective approach for addressing such issues. The special characteristics and limitations of WSNs make the requirements to the reconfiguration mechanism quite different from what has been previously proposed for other types of networks. In this paper, we propose a new software component model, namely ReWiSe, for achieving *lightweight* and *fine-grained* software reconfiguration in WSNs. In this model, a component can be reconfigured at the behavior-level instead of at the component-level. We discuss how the new component model can make the reconfiguration process lightweight in terms of component state preservation, component dependency checking, and new update unit granularity.

Keywords: component model, wireless sensor networks, reconfigurability, adaptivity, middleware

1 Introduction

Deployments of Wireless Sensor Networks (WSNs) have increased considerably over the past few years. There are more applications running on WSN platform, and more technical issues on different aspects arise. Challenges become more serious once WSNs are exploited in the new emerging applications not limiting themselves to a single function called “sense and send” with trivial data processing tasks in the local sensor node [1]. Several efforts have been done to facilitate WSN application development and distribution, where most propose a collection of middleware layer services as a means to expedite application construction. Middleware services are placed atop operating system, provide high level functionalities for application development, and help to mask the distribution and heterogeneity in the network. Recently, numerous middleware layer solutions have been proposed in the context of sensor applications, while most of them are designed for a specific domain of applications [4,5].

The techniques for middleware development are highly dependent on the underlying operating system design method. Since most popular operating systems for sensor nodes such as TinyOS [6] or Contiki [7] are based on component models, contributions to the middleware are mostly inspired from such models. Apart from challenges arising in the middleware such as addressing application domain, programming model, and non-functional service provision, WSN applications themselves become a new challenge when they act dynamically and unpredictably.

The primary tasks of each sensor application is to read sensor data from lower software layers such as operating system or sensor drivers, convert the sensed data to a desired format, and deliver the result to another node in the network according to the network architecture and routing protocol. Moreover, actions such as data gathering and data aggregation in some intermediate nodes might be needed. Therefore, most applications should perform a set of basic tasks common to the entire monitoring applications, with some differences in task details such as data sample rate and sensed data type.

In addition to the basic needs, sensor nodes need to adapt their behaviors and functionalities to cope with changing environmental conditions, and with different capabilities of each individual node in the network. For example, in an application in which a wide range of sensor nodes are deployed, we face a heterogeneous network containing sensors with different sensing parameters, and resource capabilities. As different nodes are expected to run different tasks, software with adaptable functionalities becomes an essential need. Moreover, WSNs are increasingly being used in applications whose requirements are dynamically changed over the time so that it is impossible to predict the future needs [2,3]. The problem becomes more serious when the nodes are deployed in large numbers and inaccessible places. Consequently, individual software updating becomes an impractical solution.

There are a lot of works reported in the literature for addressing dynamic application requirements and behavior, however most of them are proposed in areas such as mobile applications [8,9,10]. Although these works could give some clues for handling dynamicity in WSNs, the special characteristics of WSNs bring several new issues that should be considered as well. Also, a few works have been reported on proposing middleware solution for reconfiguration and adaptation in WSNs [11,12,13]. The main drawback of these proposals is that they assume sensor nodes are powerful enough to run heavyweight reconfiguration and adaptation tasks.

In this paper, we propose a new component model which is more suitable for performing lightweight reconfiguration and adaptation in sensor applications. Using this model, the reconfiguration is limited to the part of a component that really needs to be updated, rather than replacing the whole component with the new one. New concepts in the proposed model simplify the general steps of a typical reconfiguration model. By exploiting this model in designing WSN application software, the reconfiguration program performs its task in a lightweight manner as well as the notion of *behavior reconfiguration* makes it possible to upload only the changed part of software to the nodes, instead of uploading the whole software component. We believe that this model can be also exploited in other application areas such as mobile applications and embedded systems.

The reset of this paper is organized as follows. The next section introduces the concept of reconfigurability in WSNs. In section 3, we describe the ReWiSe component model and its constituents. Then, in Section 4 we discuss how a ReWiSe-based system can be reconfigured, along with a short description of a sample application. Related work is presented in Section 5. Finally, Section 6 gives a summary of the main points and discusses issues for future work.

2 Reconfigurability

Generally, reconfiguration service is responsible for safely reconfiguring from the current set of running application variants to the new set of application variants selected by a professional user of system or existing adaptation mechanism. These changes might include stopping and starting of applications and their components, configuring their connections and parameters, or changing device settings [14]. In the context of WSNs, reconfiguration become fully effective if the reconfiguration service retains fine-grained control over what is being reconfigured, by updating only some selected functionalities to minimize energy consumption. In other word, the ideal way of WSNs application reconfiguration is to limit updates to only the portion of software which really needs to be updated rather than updating the full application image. However, in some sensor platforms, for software updates to be successful, the whole system is required to be restarted after performing reconfiguration.

Designing reconfiguration mechanisms is directly depended on the underlying system structure. For our component-based framework, two main mechanisms could be employed, namely *parameter-scope* and *component-scope* [14]. In the former, component is reconfigured by updating the current application variables. This mechanism supports fine tuning of applications through the modification of component properties values and deployment parameters. Basically, in a typical software component, the behaviors somehow take effects from the current values of system properties. So, in some cases, it is enough to control the behaviors of component by only injecting new values for properties of components. As an example, for component responsible for collecting sample data from sensor, we can update only the value of `SampleRate` attribute in order to control the frequency at which sensors send messages. In the latter, the component can be reconfigured as a whole. Particularly, component-scope reconfiguration allows the modification of service implementation (replacement of component), adding new component, and removing running component. Parameter-scope is an effective way to implement variability, but it is less powerful than component adaptation [8].

In our model, the dynamic reconfiguration mechanisms are described based on software architecture elements known as *component*. The component model defines constructs for structuring the code on the sensor nodes. Basically, every component represents a single unit of functionality and deployment. As illustrated in Figure 1, every component can interact with its outside through *property, interface, and receptacle*. An interface specifies a set of operations provided by a component to others, while a receptacle specifies the set of interfaces a component requires from others. Therefore we are able to reconfigure the system by switching from an old component to a new one implementing the same interfaces. As mentioned before, a few works have been done recently to enable high-level software reconfiguration for

WSN applications [11,12,13]. All these proposals adopt the general common component model as their basic building blocks.

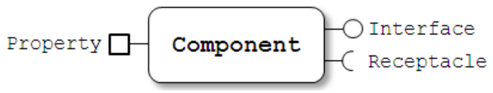

Fig. 1. A typical component model

3 ReWiSe

As mentioned in previous section, component-scope reconfiguration is proposed for manipulating the whole component. In general, four forms of component reconfiguration can be envisaged: adding new component, removing existing component, replacing with a new component and migrating a component. Depending on the form of reconfiguration, reconfiguration procedure should take care of system consistency during reconfiguration. For instance, in component replacement method, the current running component might be interacted with others, so the reconfiguration mechanism should check somehow the dependency of replacement candidate to other components. Likewise, the state of new component should be updated with the last state of old component. Therefore, for each form of configuration, several checking tasks should be carried out in order to keep system integrity and consistency at an optimum level.

Let us consider the sample component configuration depicted in Figure 2, and also assume that `Sampler` component is the candidate for swapping with another new component. Also, we assume that the difference between the old and new version is in the implementation of `IReport` interface. According to what is illustrated in the Figure, four main tasks should be performed to reconfiguration takes place: 1) checking component to ensure that it is not be in interaction with `Logger` component before starting reconfiguration, 2) checking component to ensure that it is not in interaction with `Publisher` component before starting reconfiguration, 3) saving state of component, 4) creating new component and transferring last state to it. Moreover, there are some other small tasks that should be considered for executing a perfect reconfiguration. Although it seems that these steps are necessary in general regardless the underlying platform limitations, we believe that it is possible to skip some of these steps in order to have a lightweight fashion of reconfiguration. Especially, in the context of WSNs, existing limitations such as resource usage and network bandwidth motivate us to have more fine-grained look at the component construct and their interaction model. ReWiSe is a novel definition of software component that is empowered with new features for having more dynamic component model, along with the basic functionalities of a typical component model.

The question which may arise from the above sample is: “why for updating an interface implementation in the `Sampler` component we need to replace the whole current component with new one?” To clarify more, it might be possible to update only the part of component which really needs to be updated rather than replacing it with another one. ReWiSe is going to achieve such a reconfiguration mechanism.

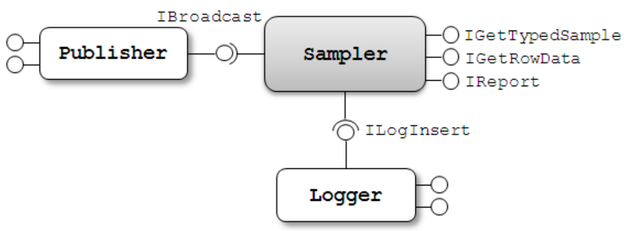

Fig. 2. Sample component configuration

Figure 3 demonstrates the constituents of ReWiSe component model. Like common component models, it is in interaction with others through interface, receptacle, event, listener and property. In fact, the outer white box is what we proposed previously many times in literature [15]. As you move through different doors of outer box to the inner structure, all things are the same as before except the *interface door*. Actually, for interface, instead of reaching to the real implementation of interfaces we face new very small component including "only and only" the real implementation of that interface. Let us call these components *TinyComponent*. For each interface exposed from outer box, we have a corresponding TinyComponent implementing that interface. In ReWiSe, every component is surrounded with *Interface Interceptor Wrapper* to route requests from other components to the associated TinyComponent. In next section, we describe the concepts and feature of ReWiSe in detail.

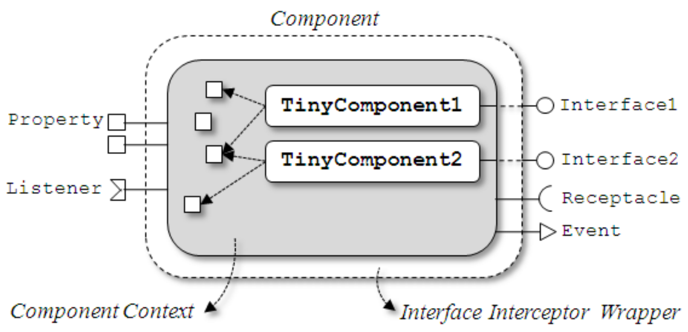

Fig. 3. ReWiSe Component Model

3.1 TinyComponent

TinyComponent is the representative of interface implementation. For each interface of a component, there is a corresponding TinyComponent containing just one method that realizes the interface. The underlying runtime system treats with TinyComponent as entity carrying out what the interface supposed to do no more. New issues arise related in interaction of TinyComponent with other elements inside the component, while its connection with world outside the main component is in the same fashion as the previous models. Firstly, unlike direct interface implementation in main component which has access easily to the component properties, TinyComponent is located out of scope of main component, so TinyComponent is not able to reach to the properties of main component. This problem is resolved by passing a pointer of main component to TinyComponent, thereby the scope is reachable for TinyComponent through a variable containing pointer to the main component. Secondly, it is about how TinyComponents can interact among themselves, like what is occurred between methods in a common component models. Actually, like the first problem, this case is originated from the fact that the scope is different now. The proposed *main component pointer* is still able to cope with this problem, because this variable is used as a proxy for everything we want out of the TinyComponent. In the next section, we discuss in more details how *Interface interceptor Wrapper* facilitates such a calling among TinyComponents.

3.2 Interface Interceptor Wrapper

To get effectively the benefits of TinyComponent unit, ReWiSe should be enhanced with a mechanism capable to route requests for a specific interface to the corresponding TinyComponent. Since TinyComponents are going to become the new candidate for replacement in the forthcoming reconfiguration mechanism, component can not maintain the name of TinyComponent in a hardcode manner. Therefore, interceptor wrapper is responsible for handling dynamically the references to TinyComponents. For the first time of executing a service, the wrapper reads from its *mapping configuration file* the name of TinyComponent implementing the service and then caches the reference to the corresponding TinyComponent in its local data. Afterward, the other requests for that service will be automatically forwarded to the assigned TinyComponent. The configuration file contains a set of structured data indentifying the name of corresponding TinyComponent for each interface.

In addition to taking the responsibility for directing the requests from other components, interface interceptor wrapper undertakes the task of handling requests inside the component. As mentioned in the previous section, TinyComponents are likely to call each others, like what happens in common component models.

3.3 Component Context

One of major challenges in reconfiguration mechanisms is how to preserve the state of component during the reconfiguration. Since the replacement level in ReWiSe is demoted to the TinyComponent, the replacement candidate does not miss its state during the reconfiguration time. *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 component reference defined in the TinyComponent scope.

4 ReWiSe-based Reconfiguration

Let us go back to the sample component configuration described above, and examine how we can carry out the reconfiguration request based on what has been proposed in ReWiSe component model. Figure 4 illustrates the new situation, where component interior design is tailored according to ReWiSe model. Similarly, we assume that `Sampler` is a candidate component for the replacement. According to our proposal, for each interface a TinyComponent, implementing the interface, will be assigned.

In the sample configuration, we focus on `Sampler` component, and examine what will happen during the reconfiguration. Unlike the previous execution in which the details of what updated inside the component was not a concern, in the new model, the configuration mechanism goes into the details of what should be really modified. The reconfiguration logic is still the same: changing only the `IReport` implementation. As shown in Figure 4, `IReport` is the candidate interface for changing its implementation. Obviously, at the first step the interaction of `Sampler` with other components should be checked to make sure that the component is in a safe state for removing. Checking method in ReWiSe is simpler than what was done previously, as depicted in Figure 5, the dependency of `TinyIReport` to others should be checked instead of checking dependency of whole main component to others (later in this section the checking mechanism is explained). After reaching to the safe point of replacement, rationally the state of component should be considered. Component context in ReWiSe, as the state preserver makes the reconfiguration process easier. In fact, the state of component will be preserved during the component lifetime, regardless how many reconfigurations are requested and run.

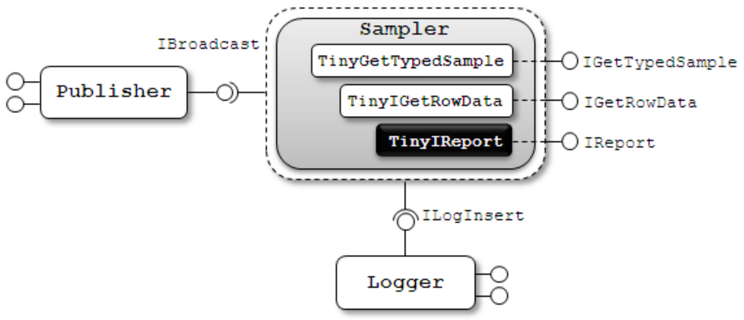

Fig. 4. Sample component configuration based on ReWiSe model

Basically, the three main design concepts of ReWiSe keep the component in a high reconfigurability degree. Firstly, the notion of TinyComponent makes the behavioral-level configuration possible; thereby if a portion of component (interface implementation) needs to be repaired we shouldn't change the whole. Another advantage of using TinyComponent is simplifying the process of checking component interaction with others by checking only the interactions of candidate TinyComponent with others. Secondly, the interface interceptor wrapper undertakes switching from old TinyComponent to new one. As mentioned, a mapping configuration file is attached to the wrapper for identifying the map between interface and corresponding TinyComponent name. Note that if the name of new TinyComponent is the same as before, the mapping file remains unaltered. Finally, the considerable improvement is taken palace in component state management. In fact, in previous models, the stateful 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 component lifetime.

4.1 Safe State for Reconfiguration

The safe state for performing reconfiguration is defined as a situation in which all instances of a component are temporarily idle or not processing any request. It means that in a typical component model the interactions among components must be monitored in order to determine when the target component reaches a configurable state. But, in ReWiSe, interface interceptor wrapper is enhanced with a *farFromSafety* variable for each exposed interface. This variable will be updated in the *application scope* each time that a request is reached. Before calling a service, *farFromSafety* is increased by one and after running service, this variable is decreased by one. Thus, at the arbitrary time of application execution, the value of *farFromSafety* for a specific interface indicates the number of TinyComponent instances currently running in the memory. Once this value reaches zero, all other request for that interface will be blocked and the reconfiguration unit accommodates new TinyComponent. Afterward, the blocked requests are processed within the new TinyComponent.

4.2 A Typical ReWiSe-based Application

We are currently developing a component-based middleware for WSN application with respect to enabling reconfiguration and adaptation. The target application for evaluating both ReWiSe and middleware is a *home monitoring application* [2]. In this application, we deploy five different types of sensor node including temperature, smoke detector, occupancy detector, in-bed detector and humidity. Based on the activities usually done in a specific room, relevant sensor types will be deployed, e.g., kitchen will be equipped by three smoke detector sensors, two temperature sensors and one humidity sensor.

As different sensor types are likely to have their own software components for running relevant application logic, at network deployment time each sensor is programmed with the core middleware services atop operation system. After successfully deployment of sensors, according to the type of each sensor the other necessary application components should be transferred remotely to each node. During the application running time, variant scenarios of component reconfigurations might be happened, e.g., if temperature sensors report unusual data for a while the smoke detector sensor should increase the sample rate and report more precise data. There are many challenges, out of scope of this paper that should be addressed in both middleware design and application scenario. Currently, we are designing and implementing the middleware, called WiSeKit, based on ReWiSe atop Contiki operating system. Simultaneously, we are investigating the detail specifications of a home monitoring application requirement to map them to a collection of software components including non-configurable model and ReWiSe model. The underlying configuration and adaptation services in WiSeKit realize the goal of lightweight reconfiguration in WSNs [16].

5 Related Work

Componet Objecy Model (COM) [17], enterprise JavaBeans(EJB) [18], and the CORBA Componet Model [19] are 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 [20]. Although Fractal is a comprehensive and extensible model for component composition and assembly, the minimal core is a heavyweight extensible unit with various features suitable for the large scale applications needing different degree of reconfiguration for different software granularity. Moreover, 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 state management during the reconfiguration. We believe that even a simplified model extracted from Fractal could not meet the requirements for a lightweight reconfiguration. Likewise, what has been proposed in OpenCom component model [21] is a coarse-grained mechanism for dealing with dynamicity in applications.

6 Conclusions and Future Work

This paper has presented a new component model that is tailored for dynamic reconfiguration for resource-limited networks such as WSNs. The component model introduces the notion of TinyComponent for achieving a lightweight behavioral level reconfiguration for WSNs. By using this component model, not only a consistent reconfiguration is guaranteed, but also the reconfiguration is carried out with a minimum overhead, because instead of swapping the whole component the needing-reconfiguration portion of component will be updated.

In this paper, we assumed that reconfiguration is limited to the TinyComponents, whereas in some cases, the main component containing TinyComponents needs to be reconfigured as a whole. We should consider the reconfiguration of main component in terms of state preservation and safety checking. The reconfiguration manager, as one of primary services in WiSeKit middleware, is another ongoing work bringing new challenges such as handling component life cycle and autonomous reconfiguration. The other possible future work is exploiting ReWiSe in other environments such as mobile or embedded systems.

References

1. Puccinelli, D., Haenggi, M.: Wireless Sensor Networks: Applications and Challenges of Ubiquitous Sensing. IEEE Circuits and Systems Magazine, vol. 5, no. 3, pp. 19--31 (2005)
2. Mozer, M.: Lessons from an Adaptive Home. In: D.J. Cook and S.K. Das, Editors, Smart Environments: Technology, Protocols, and Applications, Wiley pp. 273--298 (2004)
3. Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., Anderson, J.: Wireless Sensor Networks for Habitat Monitoring. In: Proc. of First ACM Workshop on Wireless Sensor Networks and Applications (WSNA) (2002)
4. Henricksen, K., Robinson, R.: A Survey of Middleware for Sensor Networks: State-of-the-art and Future Directions. In: Proc. of the international Workshop on Middleware for Sensor Networks, Melbourne, Australia, (2006)
5. Taleghan, M., Taherkordi, A.,Sharifi, M., Kim, T.: A Survey of System Software for Wireless Sensor Networks. In: Proc. of IEEE International Workshop on Wireless Ad Hoc, Mesh and Sensor Networks (WAMSNet'07), Korea, (2007)
6. Hill J., et al.: System Architecture Directions for Networked Sensors. In: Proc. of International Conference on Architectural Support for Programming Languages and Operating systems (ASPLOS), Cambridge, MA, (2000)
7. Dunkels, A., Grönvall, B., and Voigt, T.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proc. of the First IEEE Workshop on Embedded Networked Sensors, (2004)
8. Alia, M., Hallsteinsen, S., Paspallis, N., Eliassen, F.: Managing Distributed Adaptation of Mobile Applications. In: Proc. of 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS) (2007)
9. Rouvoy, R., Eliassen, F., Floch, J., Hallsteinsen, S., Stav, E.: Composing Components and Services using a Planning-based Adaptation Middleware. In: Proc. of 7th Intl. Symp. on Software Composition (SC'08), Springer LNCS, p. 16, Budapest, Hungary (2008)
10. Grace, P., Blair, G., Samuel, S.: ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. In Proc. of International Symposium of Distributed Objects and Applications (DOA '03), Catania, Sicily, (2003)
11. Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G.P., Zachariadis, S.: A Reconfigurable Component-based Middleware for Networked Embedded Systems. International Journal of Wireless Information Networks, vol 14, No 2, pp. 149--162, (2007)
12. Mottola, L., Picco, G., Sheikh, A.: 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, pp. 286-304 (2008)
13. Horré, W., Michiels, S., Joosen, W., Verbaeten, P.: DAVIM: Adaptable Middleware for Sensor Networks. IEEE Distributed Systems Online, vol. 9, no. 1 (2008)
14. McKinley, P. : Composing Adaptive Software. Computer, vol. 37, no. 7, pp. 56--64 (2004)
15. Crnkovic I., Larsson, M.: Building Reliable Component-based Software Systems**.** Artech House, (2002)
16. Taherkordi, A.: WiSeKit: A Component-based Middleware for Adaptive and Reconfigurable WSNs applications, Technical report, University of Oslo, (2008)
17. Microsoft, COM, http://www.microsoft.com/com
18. Sun Microsystems. Enterprise JavaBeans, http://java.sun.com/products/ejb/index.html
19. OMG. CORBA, Object Management Group, http://www.omg.org
20. Bruneton, E., Coupaye, T., Stefani, J.: The fractal component model specification. http://fractal.objectweb.org/specification
21. Coulson, G., Blair, G., Grace, P., TaÃ¯ani, F., Joolia, A., Lee, L., Ueyama, J., Sivaharan, T.: A Generic Component Model for Building Systems Software. ACM Transactions on Computer Systems (TOCS) 26(1) (2008)