Valentine : a dynamic and adaptative operating system for wireless sensor networks

Natacha Hoang[1], Nicolas Belloir[1], Cong-Duc Pham[1] and Severine Sentilles[2]

[1] LIUPPA
University of Pau
F-64013 Pau
Email:{natacha.hoang, nicolas.belloir,cpham}@univ-pau.fr
[2] Mälardalen University Departement of Computer Science and Electronics
PO Box 883, SE-721 23 Västerås, Sweden
Email: severine.sentilles@mdh.se

Abstract. We present in this paper an approach allowing dynamic reconfiguration in wireless sensor networks. This proposition is based on two complementary works. Firstly we propose a new component-based operating system (OS) for wireless sensors networks, allowing dynamic administration of components at runtime. This OS will be generated from an OS generator called Think. Secondly, we discuss and present Valentine, a specific mechanism for dynamic reconfiguration in the constrained context of wireless sensor networks.

1 Introduction

In many disaster relief scenario (earthquake, flooding...) or environmental monitoring applications, it is usually very difficult to get accurate and up-to-date local information that would definitely help the organization of rescue and/or help scientists to collect data in order to better understand the causes of a given physical phenomenon. The traditional way of getting such local information is by means of specialized devices, called sensors, which are usually connected to a fixed data acquisition system. For instance, air quality and more generally environmental monitoring networks follow this architecture. Although efficient, these infrastructures are heavy, mostly static, and are extremely difficult to deploy dynamically on a large-scale basis.

With the recent advances in microelectronics research these sensors are now able to communicate wirelessly with other sensors in the range of several meters and have also embedded processing and storage capabilities, taking place within the context of the pervasive computing, more known under the name of *ubiquitous computing* [1]. It is therefore possible to build a collaborative network of sensors on a vast geographical area. These wireless sensor networks (WSN) could be deployed more rapidly because they do not need any wired or fixed infrastructures. A survey on sensor networks could be found in [2]. It is foreseen that these autonomous sensor networks will open new perspectives for monitoring applications and most importantly disaster relief applications where relevant information must be obtained as quickly as possible.

However, the integration of such sensors inside the physical world is not an easy task. New problems are essentially generated by the severity of the constraints inherent in their resources. One can quote the conservation of the memory space or also the problem of energy management. Some solutions have already been proposed to some of these problems. For example, TinyOS [3], the reference operating system, fills very little memory thanks to the many optimizations such as the use of a component-based architecture.

On another side, Component-Based Software Engineering [4] (*CBSE*), is now recognized for the development of both flexible and well structured applications, meeting in particular needs for reconfiguration and administration. However even if TinyOS presents similar concepts to those present in CBSE, some important are lacking. In particular, no support is proposed for dynamically reconfigure an application which has been identified as fundamental in *autonomic computing* [5]. In the CBSE context, dynamic reconfiguration allows to replace a component by another in a running application. Such action can originate from several reasons. For example it can be necessary to substitute a badly implemented component, i.e. not carrying out the target functionalities or carrying them out in an incorrect way, or to add to this component news functionalities. Such a process can be performed only if the component is in a *stable state*, i.e that it is not used anymore. If it is not the case, reconfiguration could lead to an irremediable crash of the system. In the WSN context, dynamic reconfiguration becomes an important feature for reorganizing a deployed network or adding new functionalities to several sensors.

Within this context, we propose in this paper to investigate operating systems handling dynamic reconfiguration for WSN. We describe current OS limitations in section 2 and we propose in section 3 the design of a new operating system for WSN with the Think model [6]. Our model for dynamic reconfiguration in WSN is presented in section 4. Finally, section 5 presents some conclusions and directions for future works.

2 Limits of current operating systems for dynamic reconfiguration

2.1 Overview of some existing operating systems

There have been many proposals for implementing dynamic reconfiguration in sensor networks. Initially, propositions for dynamic reconfiguration were based on a *full image update*. TinyOS ,which is the *de facto* standard for WSN, is an operating system for sensor networks that generates a binary image of the entire application. We will present later on the limitations of TinyOS. Deluge [7] is a networked bootloader and dissemination protocol that process full image upgrades of TinyOS applications. Mate [8] is a *virtual machine* architecture for the resource constrained sensor devices allowing to reconfigure programs running on TinyOS nodes. Nevertheless, it has significant computational overhead.

Other approaches such as SOS [9] and Contiki [10] are based on *modular binary module*. The idea is to separate kernel from code modules. The architecture

of SOS consists of dynamically-loaded module and a statically compiled kernel. Metadatas contains module information and a linker script is used to place a specific module at the corresponding place. Contiki is a lightweight operating system with support for dynamic loading and replacement of individual programs and services. [11] presents a solution based on a SOS kernel and implements a virtual machine on top of this kernel.

Nones of these approaches provide the flexibility of CBSE. We will present on the advantages of CBSE in the section section 3.

2.2 Limits of TinyOS for dynamic reconfiguration

TinyOS addresses the problem of limited memory thanks to many optimizations : memory is allocated statically at compile-time, the nesC implementation model does not provide the concept of function pointers ... Consequently, dynamic allocation of resources is not allowed and this is one of the TinyOS's limitations. Moreover, [12] points out the difficulty to implement this property on sensors (or *motes*) equipped with TinyOS and proposed to use a less restrictive operating system allowing dynamic allocation. Even though a solution was proposed, it is too heavy to be implemented due to the employed mechanisms to overcome the static nature of the system and most particularly the use of intermediate components to manage binding reassignment.

Besides, in TinyOS, the concept of component disappears when the system image is generated. Thus, if we consider the reconfiguration property and if we assume its feasibility, this implies that if we want to modify the implementation of one component, like the communication subsystem for example, it is not possible to replace only this component. A full image of the operating system must be reinstalled.

This underlines another problem of the TinyOS model. Indeed, in WSN, sensors are used as relay node for data propagation. However, it is clear that a full system image is much bigger than a partial one. Therefor, if we assume the existence of a reconfiguration mechanism for a mote running TinyOS, this implies to "cut" this system image into several packets for transmission to the sensor. The high number of packets puts high load in the network, thus, consuming more energy.

Nevertheless, TinyOS is based on some efficient features for WSN. The first of them is the event-driven execution model. Indeed, such model is often used in embedded systems because it allows, firstly, to generate a little memory footprint and, secondly, to control more easily the scheduling activities. A second important feature is the scheduling policies used to allocate a process to the processor. TinyOS uses a FIFO queue : (i) it is a simple algorithm to implement ; (ii) activity time of a mote have to be the shortest and tasks with long duration will be considered as marginal ; (iii) as we are in a "single-user" system, a task monopolizing the processor is an acceptable situation. But in the context of reconfigurable sensor this policy would become too constraining at it will explained in section section 4.1.

3 Toward the Valentine component-based operating system

The previous section shows that none of the operating systems presented and especially the TinyOS model, meet the future requirements of complex applications in term of reconfiguration capability. In Software Engineering, the component paradigm is now recognized for being the most powerful concept to build flexible, scalable, generic and reconfigurable applications. Components run on specific frameworks or middlewares supporting all the internal mechanisms needed for component administration. These middlewares hide the complexity of the operating system and are lacking in WSN in order to allow the development of complex application softwares [13]. A recent article [14] points out that "there is a still a long way to go for a perfect middleware for WSN to really exist". But we think that the first step before having this kind of middleware is to provide an operating system fully supporting a component-based operating system and a mechanism of dynamic reconfiguration adapted to sensor network.

We propose to address the problem of dynamic reconfiguration in WSNs using the Think framework. Think is a software framework for component-based operating system kernels [15]. It enables OS architects to implement any new operating system kernels from components of arbitrary size. Despite a resource consumption more important than with TinyOS, using the Think model is very interesting for creating new OS with dynamic capabilities on a component-based architecture. We present here the design of such an operating system model for sensors combining both TinyOS well-tried aspects and dynamic component-based development, through the Think framework.

3.1 From hardware to OS components

We firstly present the different hardware elements which constitute the sensor and how they interact when designing our operating system, each OS component will correspond to a hardware element. A sensor has a microprocessor, communication I/O devices (network, sensor board, serial interface) and memory. Figure 1 shows how these elements are connected together. Most of our figure are described in UML.

The *processor* is the core of the system. It is constituted of an *Arithmetic and Logical Unit* (ALU), a *control unit* to use instructions stored in memory, a *clock* and a *bus* connecting these different components. Most processors used in sensors are currently *AVR* and *ARM*, both based on a RISC architecture. AVR processors[3] belong to the microprocessor family based on an Harvard architecture which stores program and data separately. ARM processors are widely used in embedded systems like PDAs.

Input-Output devices (I/O devices) are the links the sensor has with the environment, the others sensors and possibly the users. The transmitter-receiver (network), the connection with the development framework (serial interface)

[3] `http://www.atmel.com/`

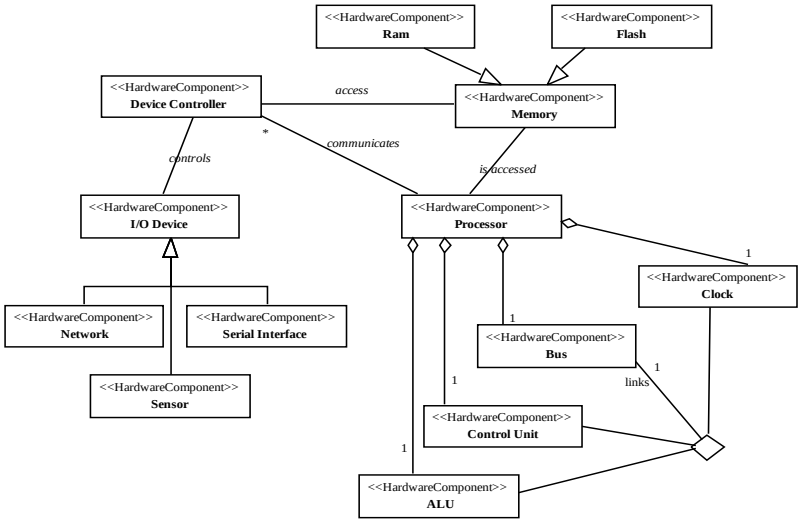

Fig. 1. "Componentization" of a sensor in UML

or the sensor board belong to this category. Each kind of I/O device has its own controller named *device controller* which is in charge of these devices and commands them. It allows to make them autonomous. The processor is informed of the end of an I/O operation by an interruption generated by the controller. Finally, *memory* enables to store both data and instructions required at runtime. thus processor and device controllers can access to it.

3.2 Design considerations

It is necessary to determine the way the operating system uses these components to realize the functionalities of the application. Thus, in a traditional system the various functionalities are implemented by lightweight processes, called threads, which are allocated to the processor by a scheduler. However using threads for sensor networks seems to be rather expensive since each sensor must store in memory a copy of the execution context during all its lifetime. Thus such mechanisms consume too much memory resources and is consequently to consider sparingly. Moreover, blocking mechanisms, such as waiting for a message arrival, will not be allowed in order to not prevent the execution of other processes. This choice is implemented through a queue mechanism.

We must determine the better adapted type of operating system kernel in the context of our study. A monolithic kernel is a kernel architecture where the entire kernel is run in kernel space in supervisor mode. Obviously, a monolithic kernel is too bulky to be able to be used on sensors. On the other hand exokernels

are tiny, since the proposed functionalities are limited to ensure protection and the multiplexing of resources. Consequently they seem better adapted and the application will directly get access to the needed hardware components.

Having described the basic elements constituting our model[4] we must now determine how these various elements will interact. Whereas a component-based model typically consists in a client-server model, a client interface asks for the execution of a request, i.e. of a function, on a server interface of another component, a sensor network is in nature strongly event-driven. This particularity would be considered and our operating system mut integrate both interaction mechanisms.

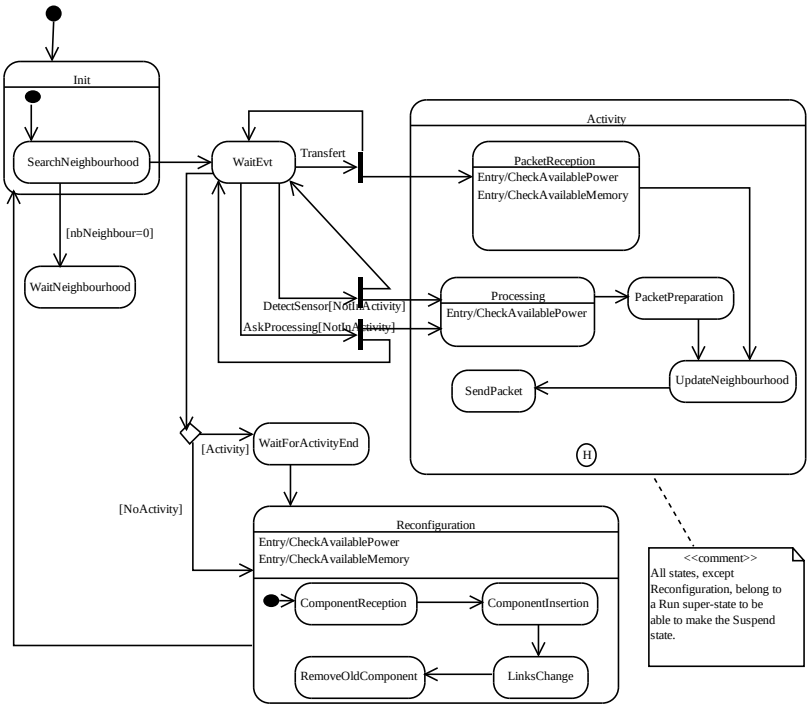

Fig. 2. The states of a sensor

In order to illustrate importance of the event mechanism in WSN, the state diagram of a sensor is presented at the Figure 2. We can see that the `WaitEvt` and the `WaitNeighbourhood` states allow to save energy by waiting for event. An event is always at the origin of the activity of the sensor. This figure highlights

[4] For more details on the model, please consult [16].

the coexistence of two behaviors : `Waiting for an event` and `Activity`. During a processing task, like aggregation of data, an event can occur. For example, the battery can announce that it is too weak to continue to supply the sensor. Thus, it must be possible to recover these events to treat them. Two processes are then needed to manage both waiting of an event and activities in progress in the sensor. It involves a second problem: how both event-driven and component functional aspects (function calls in particular) could existing together?

3.3 Analysis of dynamic reconfiguration in Think

Think [17] proposes a mechanism of dynamic reconfiguration based on the concept of controllers such as presented in the Fractal model [18]. Fractal[5] is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure systems and applications. It is developped by the INRIA[6] and France Telecom R&D. A Fractal component is formed out of two parts: the non-functional part, called *controller* and the functional part, called *content*. The *controller* is used to control and manage components. There are various controller, one for each kind of control. For exemple, the `LifeCycleController` manages the beginning and the end of a component, the `BindingCintroller` manages the links between components.

In [17], reconfiguration is considered at the component level. The objectives for this mechanism are *genericity* and *flexibility* in order to support : (i) all existing models of dynamic reconfiguration ; (ii) no modification of the functional aspect of the components ; and (iii) minimization of runtime overload memory. Moreover, only the components that need to be reconfigured have this mechanism.

To carry out these objectives, the description of dynamic reconfiguration is constrained by both *architecture* and a *set of rules*. Both of them allow to determine whether reconfiguration is realizable or not as well as the necessary steps for the component substitution. These two functionalities will be implemented through a controller containing a representation of the architecture as well as reconfiguration procedures. This new controller, called `ReconfEngine`, is connected to the existing Fractal controllers as depicted by the Figure 3. The system to be reconfigured can either be a primitive component, a composite component or a complete application.

To implement this new controller, a new interface was defined that describes three operations : `request()` allowing the initiator of the reconfiguration to submit a new description of the system configuration ; `resume()` to avoid blocking on waiting of a stable state ; `cancel()` to cancel the current reconfiguration and to return in the initial configuration. It implies that the reconfiguration controller stores the current configuration of the system.

[5] `http://fractal.objectweb.org/`
[6] `http://www.inria.fr/`

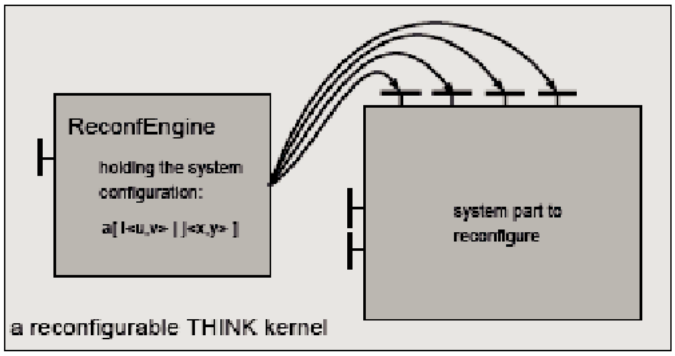

Fig. 3. Links between the reconfiguration controller and the system to reconfigure

Moreover, the Fractal controller `LifeCycleController` was upgraded with two new methods: `suspend()` to place the component and its sub-components in a stable state and `resume()` allowing to cancel the reconfiguration in progress and to restart the normal activity of the component.

[17] proposes some functionalities in the following way:

1. An initiator asks for a reconfiguration by calling the function `request()` of `ReconfEngine` and passing the new configuration as argument.
2. `ReconfEngine` checks if reconfiguration can be carried out. If not, it rejects it and stops. At this step, the components and the links having to be reconfigured are known.
3. `ReconfEngine` binds to the control interfaces of the components having to be reconfigured.
4. Then for each component to be reconfigured, `ReconfEngine` requires obtaining a stable state by calling the method `suspend()`.
5. When the stable state is reached, `LifeCycleController` reactivates `ReconfEngine` by using the function `resume()`.
6. `ReconfEngine` finally sets up the new configuration:
   (a) the new components are loaded in memory,
   (b) and/or the binds are reconfigured.
   (c) the states are transferred between the components
   (d) the components which are not used any more are unloaded from memory.
7. Reconfiguration is thus finished and the new configuration is saved.

Before reconfiguring the application, components must be in a stable state. [17] proposed to determine a stable state for a component using the same mechanism of interception that those used in Julia[7] to carry out this task : it consists in

[7] Julia is the reference implementation of the Fractal component model in Java. See `http://fractal.objectweb.org/julia` for more details

interposing *proxies* between the external and internal interfaces of a component in order to count the incoming and outgoing calls to this component. When the counter reaches zero, there is no more process in progress in the component, so there is no more activity. The component is in a stable state. This approach has performance problems since each *proxy* takes up additional memory capacity. It also prevents the use of *by-pass* techniques which make it possible to directly call an interface of a sub-component thanks to a reference to it. Without this optimization, such a call requires three memory indirections.

In conclusion, the model for dynamic reconfiguration proposed for Think in [17] is too generic to satisfy the inherent constraints of sensor networks. A modification of this proposal is thus necessary.

4 A model for dynamic reconfiguration

4.1 Presentation of the model

In this section, we present our proposition to integrate dynamic reconfiguration into WSN.

In order to solve the problem presented in section 3.2 regarding the capability to merge both event-driven and component functional aspects policies, each component requires an "event handler" for processing incoming events. That implies to have a new structure of component, presented in Figure 4.

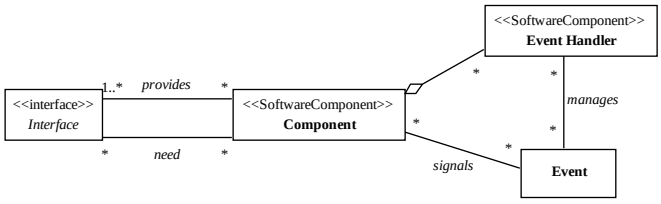

Fig. 4. Event-driven component model

The implementation of the "event handler" as a sub-component could facilitate later modifications of the system since only the handler, and the optional links between the listened events and the handler, will be modified if new events must be considered or if the existing set of managed event must be modified.

Our proposition is that our operating system will be based on an event-driven model functioning in the following way : lower level components announce events to the higher level components which are able to treat them. Higher level components can ask lower level components for occurring tasks through

function calls. In the same way, components in the same level also communicate by function calls. Figure 5 shows this mechanism.

The previously chosen scheduler, a FIFO queue, imposes that all tasks are treated in order of arrival. Thus, an event arrival does not imply its immediate execution. A solution to this problem in the use of a "priority queue" in which events will have a priority higher than function calls. In this case, when an event occurs, the scheduler will place it at the head of the queue or, after other tasks having the same priority.

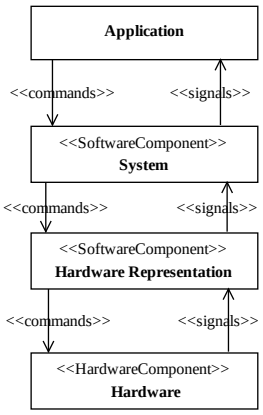

Fig. 5. Functioning of the system

In summary, we do not use a "traditional" model of operating system but the event-driven model of execution which we described previously. Indeed, this model retrieves not only most of the characteristics which made the success of TinyOS but authorizes dynamic allocations of component thus allowing advanced mechanisms such as dynamic reconfiguration. The behavior of our OS is illustrated in the Figure 6 and could be described as following :

1. Initialisation of the system.
2. Wait for event.
3. Arrival of an event causes an activity. The system is held nevertheless on the "waiting of event" state.
4. The system can be stopped by user intervention and if the user reactivates it, it restarts in the state where it had stopped.

By referring on this event-driven model, searching of a stable state is an expensive operation. In our system, at a given moment, only one task is running. That implies that when it finishes, the system is stable. Thus it carries out the

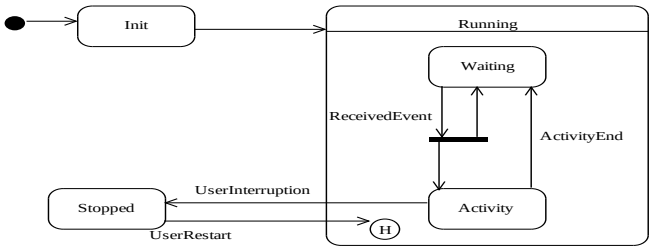

Fig. 6. The states of the system

next task in the scheduler or it recovers on waiting for event. Moreover, if one sees reconfiguration as an event, it is possible to assign to it a higher priority compared to the other tasks and thus to allow that the next task selected by the scheduler is reconfiguration. Suppression of the stable state is possible by the operating mode of sensors. Indeed, it is recommended that the sensor performs its task as soon as possible and recovers on standby in order to save the consumption of the resources. Thus, a sensor must spend most of its time on standby. Moreover, a constraint of our model is to prohibit blocking functions. One can thus consider that if the reconfiguration event occurs during the execution of a task, this one will take a relatively short time to finish and the reconfiguration will be able to take place.

We must compare the cost of waiting for the activity end with the cost of searching for a stable state for each component. Nevertheless, elements can be brought to prove the interest of this solution. Firstly, the *by-pass* technique, which constitutes one of the optimizations used by Think, can be preserved. Secondly, it is not necessary anymore to interpose *proxies* to count the incoming and outgoing calls of a component and the extension of `LifeCycleController` can also be removed. Thus, it is possible to save memory compared to the solution proposed in [17].

The second modification that we can consider relates to the tests determining if reconfiguration can be carried out. Considering the little capacity of memory available, it does not appear necessary to preserve this mechanism on sensors because it requires to store not only the description of the architecture but also the rules authorizing the transformation of this architecture. It is however possible to store only the rules allowing the transformations. Indeed, thanks to the Fractal model, somme means of component introspection are introduced through controllers, in particular the `ContentController`, the `BindingController` and the `Component`. It is then possible to obtain the overall structure but this mechanism increases the processor use. The time of establishment of a new architecture will be inevitably function of the architectural complexity of the sensor as for example in the presence of sub-components.

We thus recommend a delocalization of the whole mechanism. A *sink node* or a *base station*, having less rigorous constraints, could carry out this functionality. It could be constructed from a more powerful mote. Thus within the context of a reconfiguration, the *base station* for example would process in the following way :

1. Locally, it contains description of the sensor architecture as well as rules of applicable reconfiguration. It allows it to test if reconfiguration is feasible, and to establish the procedure to modify the sensor.
2. Once the reconfiguration procedure established (which link(s) to modify, which component(s) to remove, which component(s) to add, ...), the *base station* checks the component(s) presence on the sensor.
   (a) If the component(s) to add are already presents on the sensor, it transmits only to `ReconfEngine` the reconfiguration request with the procedure,
   (b) else, it transmits new component too or, if a closer sensor contains it, localization of this one in order to obtain a copy from it.

4.2 Limitations and future works

Some limitations result from our model. The first one relates to the increase in message exchanges on the network. If the initiator of reconfiguration is for example a sensor, the request for reconfiguration must be transmitted to the *base station* to be treated. Then the reconfiguration is transmitted from the base station to the sensor concerned. When the configuration is realized, the sensor must inform the *base station* that the operation correctly proceeded. This example shows that three messages are needed whereas the solution of [17] requires only one : initialization of reconfiguration. Nevertheless, practically reconfiguring an application or a sensor node would not be a regular activity. So increasing the rate of message exchanges from 1 to 3 seems to be an acceptable sacrifice if it result in providing an efficient reconfiguration mechanism.

The second limitation relates to the centralized aspect of the mechanism. Indeed, intuitively if the *base station* breaks down, it is not possible anymore to reconfigure the system. Moreover a problem appears if for example, a validation message of reconfiguration is lost. The *base station* can then consider that reconfiguration failed and then either considers the sensor as destroyed, or considers that it has returned in its initial configuration whereas this latter has been effectively modified. This case can occur because the reliability of exchanges is not a condition imposed in sensor networks. To solve this problem, it is possible to insert some recovering mechanisms, but this would add some overhead. We are waiting for experiment and measurement to better solve this limitation.

At the moment this work is at the final development phase. We encountered some error with generating the operating system on AVR processor. This is due to internal error in Think and we are working on the Think generator in order to solve this problem. The next step will concern the evaluation of our proposal with some tests in order to determine the overhead of this approach

and compare it to the other operating systems allowing dynamic reconfiguration. These tests are designed to : firstly determine the amount of memory needed by OS; secondly, evaluate memory capacity and occupancy rate of processor if one removes the description of local architecture; and thirdly carry out a comparison with the delocalized mechanism. That will thus make it possible to determine if it is better to save memory capacity or the traffic load in the network.

5 Conclusions and future work

In this paper, we discussed about dynamic reconfiguration in the field of WSN. This property is very important for the next generation of applications which would run on WSN. We explained that the component paradigm and middleware supporting it are the best support for deploying modern applications but are still not supported by WSN. We highlighted that existing operating systems for WSN are not adapted. Particulary TinyOS because the component based architecture is broken by its specific optimization. We proposed to create a new specialized operating system for WSN, called Valentine, based on the CBSE results, and using Think, a powerful generator of operating systems. Finally, we discussed about the characteristics of this new operating system in order to provide a support for dynamic reconfiguration and we proposed a specific model for dynamic reconfiguration.

References

1. Weiser, M.: Some Computer Science Issues in Ubiquitous Computing. Communications of the ACM **36** (1993) 75 – 84
2. Khemapech, I., Duncan, I., Miller, A.: A survey of wireless sensor networks technology. In: PGNET, Proceedings of the 6th Annual PostGraduate Symposium on the Convergence of Telecommunications, Networking & Broadcasting. (2005)
3. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for network sensors. In: Proceedings of Ninth International Conference ASPLOS, Cambridge, MA, USA (2000)
4. Szyperski, C., Gruntz, D., Murer, S.: Component Software – Beyond Object-Oriented Programming. 2nd edn. ACM Press. Addison-Wesley, New York, NY (2002)
5. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Computer **36** (2003) 41–50
6. The Think Project: Think home page (2006) `http://think.objectweb.org/`.
7. Hui, J.W., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale. In: Proceedings of the 2nd international conference on Embedded networked sensor systems, Baltimore, MD, USA (2004) 81–94
8. Levis, P., Culler, D.: Maté: a tiny virtual machine for sensor networks. In: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, San Jose, California (2002) 85–95
9. Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In: Proceedings of the 3rd international conference on Mobile systems, applications, and services, Seattle, Washington, USA (2005)

10. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I), Tampa, Florida, USA (2004)
11. Balani, R., Han, C., Rengaswamy, R.K., Tsigkogiannis, I.: Multilevel software reconfiguration for sensor networks. ACM Conference on Embedded Systems Software (EMSOFT) (2006)
12. Kogekar, S., Neema, S., Eames, B., Koutsoukos, X., Ledeczi, A., Maroti, M.: Constraint-guided dynamic reconfiguration in sensor networks. In: Proceedings of ISPN 04, Berkeley, CA, USA (2004)
13. Blumenthal, J., Handy, M.: Wireless Sensor Networks - New Challenges in Software Engineering. In: Proc. of the IEEE Emerging Technologies and Factory Automation Conference, Lisbon, Portugal (2003) 551–556
14. Hadim, S., Mohamed, N.: Middleware challenges and approaches for wireless sensor networks. IEEE Distributed Systems Online **7** (2006)
15. Fassino, J.P., Stefani, J.B., Lawall, J., Muller, G.: Think: A software framework for component-based operating system kernels. In: In the USENIX Annual Technical Conference, Monterey, CA, USA (2002) 73–86
16. Sentilles, S.: Architecture logicielle pour capteurs sans-fil en réseau. Master report, University of Pau, Pau, France (2006) http://www.univ-pau.fr/ belloir/DOC/Sentilles-Rapport_final.pdf.
17. Polakovic, J.: Dynamische rekonfiguration in think. Master's thesis, Universität Karlsruhe (2004) Master report.
18. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal Component Model and its Support in Java. Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems **36** (2006) 1257–1284