Protocol Awareness: A Step Towards Smarter Sensors

Hoel Iris, François Pacull
CEA-LETI MINATEC Campus
17 rue des Martyrs, 38000 Grenoble, France
Email: hoel.iris@cea.fr ; francois.pacull@cea.fr

***Abstract*—Low power consumption and reliability are two important properties in the wireless sensor network area. The approach presented here to improve these aspects is to use a rule-based middleware enforcing a coordination protocol on top of the communication protocols imposed by the different wireless sensor networks. In addition, we move the callee side of this protocol from the gateway to the sensors/actuators in order to make them able to directly respond to this protocol.**
The high-level coordination protocol brings on the one hand the control from the application side the activities (sleep/awake) of the sensors and on the other hand the transactional processing of operations involving a group of sensors / actuators. This has a positive impact on the consumption and on the reliability.

***Keywords*-Coordination Middleware; Smart Sensor; Transaction; Power Consumption.**

I. INTRODUCTION

A Wireless sensor network (WSN) is a set of sensors or actuators connected together through a wireless connection. WSNs are nowadays commonly used in various domains because the absence of wire helps the deployment and decreases the installation cost.

The main components of a sensor/actuator are some sensing or acting units, a micro-controller, a transceiver and a power unit.

Basic objectives of sensor networks were accuracy, flexibility, cost effectiveness and ease of deployment. As these properties are now taken for granter, low power consumption and reliability are the next steps to consider.

The paper presents our approach to improve these aspects through a rule-based middleware [1] enforcing a coordination protocol on top of the communication protocols imposed by the different WSN. The high-level coordination protocol brings on the one hand the control from the application side the activities (sleep/awake) of the sensors and on the other hand the transactional processing of operations involving a group of sensors / actuators.

The paper is organised as follows. Section II introduces the problem to solve. We describe in Section III our approach based on the high level coordination protocol enforced by our middleware and the design of smart sensors able to react smartly to this protocol. Section IV presents the hardware we considered. In Section V we exemplify our approach both for power consumption and reliability. We conclude in Section VI.

II. PROBLEM TO SOLVE

The usual way actuator/sensor wireless networks are designed does not help to solve the two problems of *reliability* and *power consumption* as described here after.

A. Reliability

Most of the time the wireless communication policy in WSNs is only best effort. In addition, sensors are battery powered for autonomy reasons and they may stop their activities or decrease the wireless signal strength just because the battery is partially or totally discharged. Thus, there is no guaranty that a sent message is eventually received.

A traditional way to enforce reliability in an asynchronous system in presence of failures is to embed the operations involving the sensors/actuators within transactions [2]. For instance, we consider to open a windows (actuator) and to store this information in a database responsible for keeping the state of the system. If this is not embedded in a transaction, it may happen that the change in the database is done while the actuator is not reachable or on the contrary that the window is opened but the database is locked for some external reasons. As a transaction unrolls a two-phase commit (2PC) protocol if an actuator is not reachable in the first phase (i.e. reservation) then the transaction is aborted preventing the database update.

Obviously, this makes sense only if the transaction protocol really involves the actuator and not only the gateway otherwise a failure may happen in between the gateway and the actuator after the commit of the transaction.

We propose to implement smart sensors/actuators that directly provide the required transactional capabilities.

B. Power consumption

Power consumption is the sum of the powers consumed for sensing and computing activities plus the power used for the delivery of information through the radio. In general, both of these parts can be slept when they are not used. However, the main issue is precisely to define when they are not used. Indeed, we can distinguish two categories of sensors. The first one, based on an alarm (e.g. presence detector) can be easily slept until something external happens. On the contrary, the second type (e.g. temperature sensor) regularly emits the information without any idea if it is useful or not for the application. Among the improvements proposed, we may find systems that offer configuration facilities allowing the user to

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-208-0

define the delay between two emissions. It is also possible to avoid to emit the information if it did not change or if the change is inside a given interval that can be configured [3]. However, even if these efforts are noticeable it is far to be optimal because in general the user has little information to efficiently configure the sensors in advance since most of the knowledge appears during the execution of the application

Part of the solution is to let the initiative of the application to interrogate the sensors (pull mode) rather than trying to optimise its emission rate (push mode).

C. Our approach

Our approach is based on the combination of a rule-based middleware which coordinates the actions of a group of software components through a high level protocol. This protocol, thanks to a limited yet effective set of primitives allows on the one hand to control the interrogation of the sensors from the application side and on the other hand to embed a set of operations on sensors, actuators and software components within transactions.

In addition, we have designed sensors/actuators aware of this protocol and then able to behave like first class components of our middleware. In other words, we replaced the classical approach where the sensors are accessed via a gateway with no control on what is actually done beyond the gateway to an architecture unrolling our protocol until the physical devices. The transport layer of the sensor network is in this case a vehicle for our coordination protocol.

The usage of this protocol together with sensors aware of this protocol propose a response to the both mentioned issues: power consumption and reliability.

We describe in the next sections the middleware and its protocol and the global design of our sensors/actuators both at the software and hardware levels.

III. Architecture approach: software

A. Coordination Middleware

The middleware provides a uniform abstraction layer that eases the integration and coordination of the different components (software and hardware) involved in WSNs. It relies on the *Associated memory* paradigm implemented in our case as a distributed set of bags containing resources (tuples). Following Linda [4] approach the bags are accessed through the three following operations:

- `rd()` which takes as parameter a partially instantiated tuple and returns a fully instantiated tuple from the bag whose fields match to the input pattern;
- `put()` which takes as parameter a fully instantiated tuple and insert it in the bag;
- `get()` which takes as parameter a fully instantiated tuple, verifies its presence in the bag and consumes it in an atomic way.

The bag abstraction can encapsulate real tuple spaces but also databases, services, event systems, sensors and actuators. From the sensor point-of-view, a couple of bags map a set of sensors and contains the resources corresponding to basic information: e.g. tuples (sensorid, value, timestamp) or (sensorid, type) allows to model all the data and metadata required to manipulate sensors through the `rd()` or the `get()` operations. For the actuators, the `put()` operation is used to introduce tuples under the form `(actuatorid, function, parameter1, parameter2)`. Once inserted, the bag can actually trigger the correct action on the physical actuator with the appropriate parameters.

The tree operations `rd(), get() and put()` are used by the *Production rules* [5] to express the way these resources are manipulated in the classical *pre-condition* and *performance* phases. The rules are enacted by dedicated components called *coordinators*.

Precondition phase: It relies on a sequence of rd() operations to find and detect the presence of resources in several bags. This can be be sensed values, result of service calls or states stored in tuplespaces or databases.

The particularity of the precondition phase is that:

- the result of a `rd()` operation can be used to define some fields of the subsequent `rd()` operation
- a `rd()` is blocked until a resource corresponding to the pattern is available
- a `rd()` operation at the right hand side of a blocked `rd()` is not active and will invoke its bags only when the previous `rd()` receives a response.

This mechanism will be used to access the sensors only when it is required by the application.

1) Performance phase: It combines the three operations `rd(), get() and put()` to respectively verify that some resources found in the precondition phase are still present, consume some resources and insert new resources. In this phase, the operations are embedded in *distributed transactions*. This particularity ensures several properties that go beyond traditional production rules. In particular it ensures that

- we can verify that the important conditions responsible for firing the rule (precondition) are still valid in the performance phase.
- the different involved bags are effectively all accessible.

These properties can improve reliability provided that the sensors are aware of the coordination protocol.

B. Protocol aware sensor

In order to make the sensors/actuators capable to understand the coordination protocol we need to implement at the sensor level the different operations that will be invoked during the enactment of the rules. In addition, we need to implement stubs that encapsulate the communication layer in order to provide to the coordinator the way to invoke transparently the remote bags.

Stub: We briefly describe how the stub mechanism is used in our middleware in figure 1. When the coordinator needs to access a given bag, it asks the nameserver in order to obtain the stub that will be used to invoke the `rd()`, `get()` and `put()` operations.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-208-0

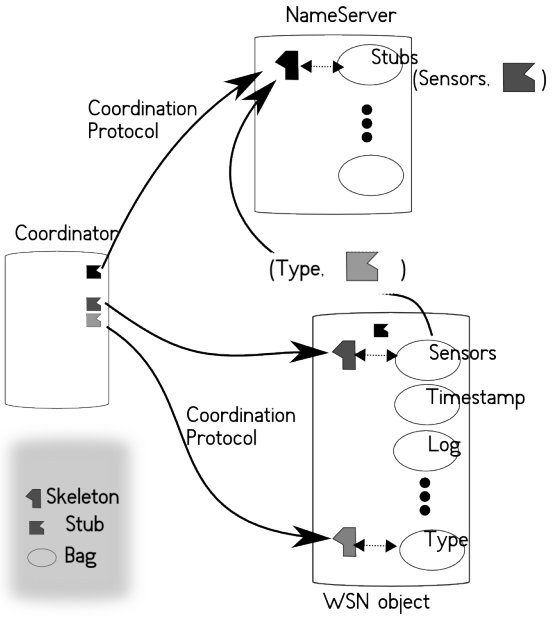

Fig. 1. stub

The name server stores the information related to the stubs as resources of one of its bag called `Stubs`. A stub is inserted by each bag in the name server at starting time. Once obtained by the coordinator the latter just calls the appropriate primitives via the stub and has no idea about the way they are implemented. This means that the transport layer is completely hidden at this level. A corresponding skeleton able to receive and decode the requests coming through the stub is present at each bag.

A stub allows to invoke two distinct `rd()` operations: one for the precondition phase that may block when no corresponding resource is currently available and one for the performance phase which is decomposed into a `pre_rd()` and `commit_rd()` or `abort_rd()` according to the 2PC used to enforce the transaction. The `get()` and `put()` operations are also decomposed for the same reasons.

Another point concerns the management of the wake up of the sensor since we consider that the sensor is in sleep mode by default, and wakes up before calling a primitives of the protocol. Then a signal, whose role is to wake up the sensor, is sent to it before any request. Moreover, we consider that the sensor returns in sleep more right after the termination of the `rd()` and the `commit_*()` or `abort_*()`. Different type of signals are discussed further.

Skeleton: at the sensor side: We consider now, the treatment to be done at the micro-controller side for operation of the precondition and performance phases.

The only operation required in the precondition phase is the `rd()` operation. It has the following behaviours.

If a resource is available (i.e. a value may be read and returned immediately) then the read is non-blocking. This corresponds to sensors like temperature or humidity measuring most of the time a physical value. The resource based on the sensed value is directly returned to the stub call and nothing else has to be done.

If a resource is not available then the read needs to be blocked until a resource becomes available. This corresponds to sensors like presence sensor or threshold detector that are bind to an alarm. As it would not be efficient on a power consumption point of view to let the sensor alive just to block the `rd()` operation, we implement it in a different way. The value returned to the `rd()` warns the calling stub that nothing is currently available and that the sensor will take the initiative to send the resource when it becomes available. At the stub level, we just block. When the sensor receives the alarm (e.g. detection of a presence), which wakes it up, the resource is returned to the stub along with an identifier which allows the stub to retrieve to which `rd()` it corresponds. The resource remains stored locally in RAM at the sensor level for the performance phase. The sensor can return in sleep mode and the stub can end the process corresponding to this `rd()`.

For the performance phase, during the `pre_*()` operations, the sensor has to store intermediate informations that will be used during the `commit_*()` or `abort_*()` operations: i.e. the considered resource passed as parameter.

For the `pre_rd()` and `pre_get()` the concerned resource, if available, is locked to prevent any other `pre_*()` operation coming from other transactions to be accepted. If the resource may be locked, `ok` is returned and the lock status is locally stored. Otherwise `notok` is returned.

For the `pre_put()` we need to verify that the operation is possible. We can verify for instance that the actuator can be manipulated. Accordingly, `ok` or `notok` is returned.

If a `commit_*()` is invoked then the considered operation is effectively done: nothing for a `commit_rd()`, the destruction of the resource for a `commit_get()` and the action associated to the actuator parametrised according to the resource for a `commit_put()` If an `abort_*()` is received then nothing is done. In all cases, the intermediate informations (locks, ...) are cleaned.

C. Model of a sensor/actuator network object

To define a prototype object that encapsulates a sensor/actuator network we can consider the following set of bags required for its management.

For the sensors part we have the following bags.

- `Sensors`: stores the sensor id and the value read
- `TimeStamp`: stores the id and last reading time
- `Log`: stores the id and reading time
- `Type`: stores the id and its type (i.e.: temperature, ...)

For the actuators part we have specific bags for the different types of actuators. These bags implement for the `put()` method the actual action to be realized to act on the physical actuator.

With a classical approach, this object is located at the gateway level as shown on Figure 2 with the first object.

With a smart sensor approach, we have decomposed the set of bags in order to let some of the bags at the gateway level and to move the others at the micro-controller side.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-208-0

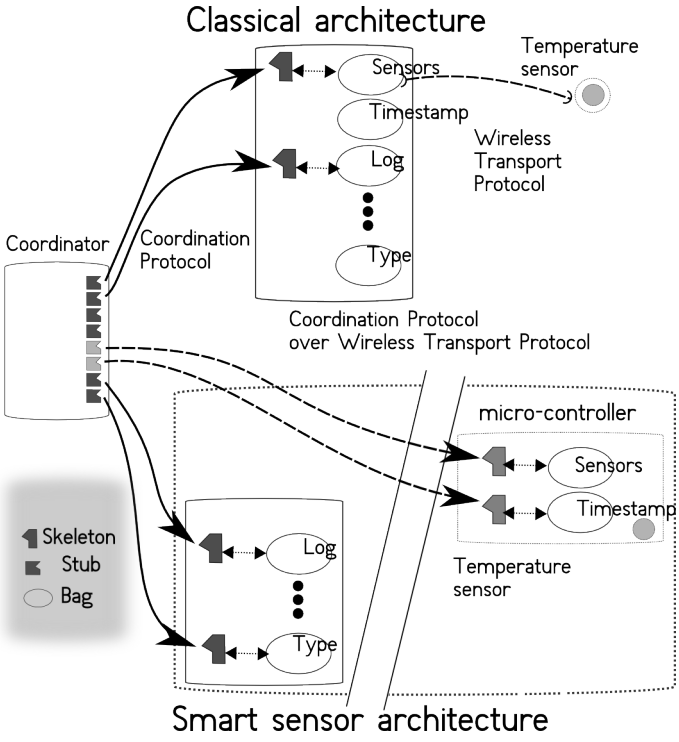

Fig. 2. Sensor network object: the smart sensor approach

`Sensors` and `TimeStamp` bags which contain respectively the value associated to a sensor and the timestamp corresponding to the last sensed value are implemented on the micro-controller side. All the other bags are implemented at the gateway side. The main reason is these bags contain information that either records the sensed value or store some status updated very sparsely. Thus, in order to save memory space and power-on time on smart sensor we keep these bags on the gateway level and we implement the bags which makes sense at the micro-controller side. For instance, if we need to access the log of the successive values read by a temperature sensor, it is not required to ask the micro-controller itself, this can be done at the gateway level. On the contrary, the current temperature needs to be asked to the micro-controller.

For an actuator, we implement the bag which is responsible for the real action on the environment on the micro-controller side since we want to ensure the transactional property. Some verifications are done during the `pre_put()` operation. For instance we can verify that the rotation we would like to do on a motor is possible without damage. We could also verify that the command is sensible: to ask to close a window that is already close is typically something that is not normal. We can also verify that the actuator is physically able to do the requested command or that the remaining energy is enough to perform the action. In case of trouble, a `notok` is returned.

IV. Architecture approach: hardware

We present in this section the hardware details corresponding to the different smart sensors we implemented.

A. The platforms

We have considered three embedded platforms to experiment different levels of performance at the microcontroler level and different communication standards: *OpenPicus Flyport (Wifi-802.11)* [6], *Atmel SAM3N-EK (802.15.4)* [7] and *Arduino Pro 3.3V (802.15.4)* [8]. These three architectures are widely used for their low power and low cost characteristics. They all three offer easy-to-use development environments avoiding most of cross-compilation problems and microcontroller driver development part.

OpenPicus Flyport (Wi-Fi): This platform is based on a Microchip MRF24WB0MA/RM Wi-Fi chip and a Microchip PIC 24F 16bits processor.

The board embeds a small web-server, running on top of FreeRTOS operating system. This web server allows to encapsulate remote function call through simple URLs over HTTP protocol with a mechanism closed to cgi-scripts.

This is very similar to what is done in our middleware for the default transport layer within the stub. This allowed us a straight forward implementation of the smart sensor. This is the main reason for the choice of this board.

For the integration of physical sensors/actuators, the board offers eighteen digital I/Os and four analog inputs. Digital pins can be mapped between one I2C, one SPI, four UARTs and nine PWMs. Three different external interrupts can wake up the Flyport from standby mode.

The associated communication link is a classical Wi-Fi which allow a very easy integration in existing installation with a range that may cover the entire floor of a building.

Atmel SAM3 (802.15.4): We have used a SAM3N-EK board that was available in our laboratory with a Cortex-M3 32bits based micro-controller. This board offered a convenient experimentation framework and as the Atmel environment allows, with a same API, to target every ARM and AVR microcontroller belonging the Atmel family the solution is quite generic. This is the reason why we used this platform. The ARM Cortex-M3 core is used by many constructors, including ST Microelectronic, Atmel and Texas Instrument, to build low power but still powerful chips.

The board offers 64 I/Os including PWMs, I2Cs, SPIs, UARTs and ADCs. Some I/Os are connected at on-board LCD, buzzer, SD card host, user switch, RS232 connector or touch sensors. Sixteen different external interrupts can wake up SAM3N-EK from standby mode.

We use for the communication Digi-XBee series one modules [9], which provides IEEE 802.15.4 wireless networking protocol. A module is connected to the board via one UART and an other is plugged on the gateway via an FTDI breakout board. We use them in peer-to-peer mode. The data rate is $250Kbit/s$ on the $2.4GHz$ RF band. The maximum frame length is one hundred bytes. This is less than the Wi-Fi solution but with an adhoc encoding it is efficient enough to allow serialized procedure calls to hold into a single frame.

Arduino (802.15.4): The Arduino board we used is based on Atmel AVR 8bits architectures. It is open source and supported

by a large community. This is the reason that conducted us to use this board.

We chose the Pro 3.3V version due to the reduced on-board electronics and a working voltage that corresponds directly to XBee modules we use for the communication. The XBee module is connected to the micro-controller via the UART interface like for the Atmel board.

The board offers fourteen digital I/Os and six analog inputs. Digital pins support one UART, six PWMs and one SPI. Analogue pin could also be used for adding one I2C interface. Two different external interrupts can wake up Arduino from standby mode.

B. Power consideration

Boards: Power consumption of considered micro-controllers are under $10\mu A$ at $3.3V$ in standby-mode, while it is at least $5mA$ in run mode. However, in the case of the SAM3-EK Atmel board, the complete board is a development board for which at least one electronic device was not designed for low power usage. Thus, we did not considered this board for the power consumption experiment. On the contrary, openPicus Flyport and Arduino Pro 3.3V boards are designed for low power, in particular in standby-mode. The respective consumption measured with a full charge lithium-ion battery $(4.1V)$ with wireless unit in the same state than the micro-controller are summarized in Table 1.

Micro-controller+wireless	standby	run mode
Arduino + Xbee	206μA	57.1mA
OpenPicus Flyport + Wifi	97μA	127.5mA

Table 1: consumption of the full board.

Wireless communication unit: XBee (802.15.4) and Wifi (802.11) have been considered: They both offer power down mode to control their power state from the micro-controller. Respective consumptions are in Table 2

Wireless module	standby	communication
Xbee	10μA	50mA
Wifi	0.1μA	120mA

Table 2: consumption of the wireless communication unit

The first is more expensive in standby mode but can be integrated to a larger number of micro-controllers. The second is more expensive in communication mode but it handles a bigger amount of data with a higher communication range.

Radio control: Primitives `radio wake up` and `radio go sleep` are called during the board `power down` and `power up` procedures. In this way, when micro-controllers are in run mode, the wireless communication link is available. We could have used smarter radio power control but since we are considering that the board should be sleeping most of the time we did not go further in this direction.

Sleep mode and wake up events: As in sleeping mode, the consumption is divided by at least one hundred, our approach is to force the sensor to be in sleeping mode almost all the time. It is wake up on demand either because the coordinator send a signal before invoking the appropriate primitive or because an alarm is triggered by a physical sensor linked to the board. Both events raise a different interrupt received by the micro-controller and thus they are treated accordingly in order to execute the appropriate code (See Section III-B).

Wake up signal: For the wake up signal sent by the coordinator (gateway side) we envisaged different solutions that could be used. The first is based on a simple modulated IR signal that can be used if the gateway is in the same location than the sensor. The second is based on a less expensive wireless signal [10] (e.g. $433Mhz$) that can be used to this purpose. Finally, the third may consider the new passive RFID technology working in a range of 15 meters. This problem is open and still under investigation and in our first experiment we only used the IR solution.

V. EXAMPLES

First example: consumption aspect

We consider a simple application where we collect external temperature in order to control the heating system according to the variation of the temperature and the speed of this variation. Then, we have an algorithm that defines according to a window of sensed temperatures the most appropriate time to make the next set of measures. Basically, if the temperature does not change a lot we increase the delay between two measures and we decrease it if the variation increase. It is impossible to compute in advance the time when the measurement need to be done.

The classical way is to ask the sensors to send the information each $\delta(t)$ and to sleep the rest of the time. With $\delta(t) = 5mn$, we have $24 * 12$ measures sent by each sensor to the gateway per day. Most of them will not be used at all and for some of them we have inaccuracy due to the acquisition rate that does not match exactly the algorithm.

If we use a rule triggering the interrogation of the sensors upon the insertion of a resource by the algorithm, we first remove the inaccuracy and second we can decrease the activity of the sensors to the exact required number of measures. We consider that the number of really useful measures is only 50 per day. The ratio is 0.17 (5 times less).

The average power consumption PC is given by

$$PC = R * C_{RunMode} + (1 - R) * C_{StandbyMode} \quad (1)$$

with R the run time ratio $R = \frac{RunTime}{TotalTime}$
We have measured that a run period (wake-up, send, sleep) takes 0.04 seconds for the Arduino and 1 second for the Flyport. It gives the following ratio for classical (C) and smart sensor (S) implementations.

$$R_C = \frac{24 * 12 * 1}{24 * 60 * 60} = 0,333\%$$

$$R_S = \frac{50 * 1}{24 * 60 * 60} = 0,058\%$$

Reported to equation (1) we obtain with the figures of Table 1 for the Flyport board the both implementations.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-208-0

$C_C = 33310^{-5} * 127.5 + (1 - 333.10^{-5}) * 0.091 = 0.516mA$
$C_S = 57.810^{-5} * 127 + (1 - 57.8.10^{-5}) * 0.091 = 0.165mA$

and for the Arduino

$$R_C = \frac{24 * 12 * 0.04}{24 * 60 * 60} = 0,0133\%$$

$$R_S = \frac{50 * 0.04}{24 * 60 * 60} = 0,00231\%$$

$C_C = 13310^{-6} * 57.1 + (1 - 13310^{-6}) * 0.206 = 0.2135mA$
$C_S = 23.110^{-6} * 57.1 + (1 - 23.110^{-6}) * 0.206 = 0.2073mA$

If we consider a battery capacity Cap, the following formulae gives the saved time in hour.

$$H = \frac{Cap}{C_S} - \frac{Cap}{C_C} \quad (2)$$

With a capacity $Cap = 1300mAh$ for the battery we save $H = 7892 - 2521 = 5371\ hours$ i.e. 244 days for the Flyport and $H = 6270 - 6087 = 183\ hours$ i.e. 8 days for the Arduino. The battery allowing respectively 328 days for the Flyport and 261 days for the Arduino.

Micro-controller+wireless	Classical	Smart	Gain
Arduino + Xbee	253 days	261 days	3%
OpenPicus Flyport + Wifi	105 days	328 days	68%

Table 3: Battery autonomy for both approach.

The interesting result is that solving the power consumption issue only acting on the consumption of the wireless communication unit is probably not the unique research direction. Indeed, with the classical approach the Arduino is far better thanks to its very low consumption when running while with the smart sensor approach the Flyport is far better the Arduino thanks to its very low consumption in standby. This means that using a simpler to deploy communication protocol (i.e. Wifi) is affordable if used in a smarter way.

Second example: transactional aspect

We consider a mobile robot equipped with a pan-tilt camera that follows a moving object. We have an external software component that computes from captured images the next positions of the robot and the camera in order to have the object centred. New positions are inserted in the respective bags as parameters of the motors controlling the pan, the tilt and the robot.

We have different competing rules moving the robot and the camera. They use a common ticket resource which ensures that only one of them will be performed. This means that we can have the object centred by moving the robot or by moving the camera. If the rule moving the camera is aborted because for instance the requested move for the pan cannot be physically done. Then, the rule moving the robot can be triggered to compensate the default of the camera. On the contrary, if the robot is blocked and thus cannot move, the rule controlling the camera may be considered. The possibility to propose alternative treatments to solve a problem that can take into account physical limitation of an actuator is provided almost for free at the middleware level.

VI. Conclusion

We have presented an approach to improve power consumption and reliability in the wireless sensor network area. This approach is based on a high level rule-based middleware that coordinates the operations involving the sensors and actuators. This offers the possibility to activate the sensors with a signal (external to the communication) when they are really needed and thus let them most of the time in standby mode to reduce power consumption. In addition the coordination protocol allows to embed actions on sensors and actuators within transactions to improve reliability. We have implemented three different smart sensor boards able to understand the coordination protocol of our rule based middleware.

Finally our contribution has been illustrated from a power consumption and reliability improvement point of view with two basic examples that show the possibilities offered by our approach.

Acknowledgment

This work has been partially funded by the FP7 SCUBA project under grant nb 288079.

References

[1] L.-F. Ducreux, C. Guyon-Gardeux, S. Lesecq, F. Pacull, and S. R. Thior, "Resource-based middleware in the context of heterogeneous building automation systems," in *38th Annual Conference of the IEEE Industrial Electronics Society (IECON 2012)*, 2012.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing, 1987.

[3] S. Feizi, G. Angelopoulos, V. Goyal, and M. Medard, "Energy-efficient time-stampless adaptive nonuniform sampling," in *Sensors, 2011 IEEE*, oct. 2011, pp. 912 –915.

[4] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, pp. 444–458, April 1989.

[5] T. A. Cooper and N. Wogrin, *Rule Based Programming with OPS5*. Morgan Kaufmann, July 1988.

[6] " OpenPicus Flyport Documentation Repository ," http://http://www.openpicus.com/site/downloads/downloads, 2012.

[7] " SAM3N-EK Documentation Repository ," http://www.atmel.com/tools/SAM3N-EK.aspx?tab=overview, 2012.

[8] " Arduino board Documentation Repository," http://arduino.cc/en/Main/ArduinoBoardPro, 2012.

[9] " XBee-PRO 802.15.4 OEM RF Modules," http://http://www.digi.com/xbee, 2012.

[10] S. J. Marinkovic and E. M. Popovici, "Nano-power wireless wake-up receiver with serial peripheral interface." *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 8, pp. 1641–1647, 2011.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-208-0