A Java Library for Event-Driven Communication in Power-Manageable Reactive Sensor Nodes

Emanuele Lattanzi and Alessandro Bogliolo
Department of Base Sciences and Fundamentals - DiSBef
University of Urbino - Italy
Email: emanuele.lattanzi, alessandro.bogliolo@uniurb.it

***Abstract*—The energy efficiency of wireless sensor networks strongly depends on the possibility of exploiting the idleness of their nodes. In principle, idle periods could be fully exploited by making use of ultra low power micro controller units (MCUs) and power manageable network interfaces which provide a wide range of sleep states with sub micro Watt power consumption. One of the key issues, however, is to avoid to keep sensor nodes busy when they could be idle, thus reducing the opportunity of dynamic power management. This issue is particularly serious in case of sensor nodes running a virtual runtime environment, since the virtual machine (VM) is seen by the scheduler of the underlying operating system (OS) as a process which is always active in spite of the idleness of the threads running on top of it. On the other hand, the benefits of virtualization in terms of abstraction and usability motivates the development of sensor nodes with power manageable virtual runtime environments. Promising results have been recently achieved in this direction by using a modified version of the Darjeeling VM on top of Contiki OS. This paper moves a step forward by introducing** *VirtualSense*, **an event-driven communication library for the Darjeeling VM which exhibits two distinguishing features. First, it is general enough to enable the implementation of advanced communication protocols in Java. Second, its event-driven nature makes it possible for a Java thread to react to incoming messages without keeping the MCU busy while waiting.**

***Keywords*-Event drive, communication library, reactive sensor, low-power;**

I. Introduction

The energy efficiency of a wireless sensor network (WSN) depends on the capability of its nodes to adapt to time-varying workload conditions by turning off unused components and by dynamically tuning the power-performance tradeoff of the used ones. *Dynamic power management* (DPM) is a wide research field which has brought, on one hand, to the design of power manageable components featuring multiple operating modes and, on the other hand, to the development of advanced DPM strategies to exploit them. The best DPM strategy is the one which meets the performance constraints imposed by the application with minimum power consumption. Apart from the fine tuning of the power-performance tradeoff of the active states, the main power-saving opportunities come from idle periods, which allow the power manager to take advantage to ultra low-power inactive modes. Exploiting idleness is mandatory in wireless sensor nodes, which spend most of their time waiting for external events or for monitoring requests, and which are often equipped with *energy harvesting* modules which promise to grant them an unlimited lifetime [5] as long as their average power consumption is lower than the average power they take from the environment.

A suitable answer to power management needs in WSNs is provided by state-of-the-art ultra low power micro controller units (MCUs), which feature a wide range of active and inactive power states while also providing enough memory and computational resources to run a virtual machine (VM) on top of a tiny operating system (OS). Virtualization adds to the simplicity and portability of applications for WSNs at the cost of increasing the distance between hardware and software, which might impair the effectiveness of DPM both for the limited control of the underlying hardware offered by the virtual runtime environment, and for the limited visibility of the actual activity offered by the VM. In fact, the VM is usually viewed by the scheduler of the embedded OS as a process which is always active in spite of the possible idleness of its threads. Two solutions have been proposed to address these issues. The first one is provided by *bare-metal* VMs, which runs directly on top of the MCU without any OS [6], [7], [8], at the cost of loosing portability. The second one is provided by full-fledged software stacks specifically designed for power manageable sensor nodes in order to make it possible to take DPM decisions directly from the runtime environment and to grant to the OS scheduler full visibility of the idleness of the virtual tasks. This is the approach adopted in this paper, starting from a recently proposed architecture [4] base on Darjeeling VM [2] and Contiki OS [1].

The effectiveness of DPM risks to be impaired, however, by the paradigm adopted for inter-node communication. Although a sensor node is primarily designed to sense a physical quantity and to send a message to the sink to report the measured value, most of the nodes in the network act as routers to relay other nodes' messages towards the sink. Moreover, in self-adapting sensor networks all the nodes have to be able to receive *interest* messages from the sink and to broadcast them to their neighbors in order to be assigned a task and to update their own routing tables

[9]. While all other activities can be either scheduled or triggered by external interrupts, receiving a message is an asynchronous event which needs to be carefully handled. In the Darjeeling VM a thread waiting for a message is suspended and rescheduled at next timer interrupt (by default, after 10ms) to look for an incoming message in the network buffer. This polling mechanism keeps the MCU busy regardless of the idleness of the node.

This problem could be solved by implementing a smarter protocol in the Contiki OS, but this would fail in the attempt of making the sensor node completely programmable in Java. An alternative approach would consist of increasing the timer interrupt in order to grant to the OS enough time to put the MCU into an inactive state, but this would reduce time resolution and increase the average response time of the node.

This paper presents a communication library for the Darjeeling VM which makes available event-driven send and receive primitives to achieve two goals: to provide the generality required to enable the implementation of advanced communication protocols in Java, and to make it possible for a Java thread to react to incoming messages without keeping the MCU busy while waiting. Experimental results show that the proposed library enables the full exploitation of the low-power states of the MCU with a negligible time overhead.

The rest of the paper is organized as follows: Section II presents the architecture of a node with virtual runtime environment, Section III introduces the Java library, Section IV presents a representative case study and some measurements, Section V concludes the work.

II. Node Architecture

We consider the architecture of a sensor node composed of: a power-manageable MCU, an embedded OS (namely, Contiki OS [1]), and a tiny Java-compatible VM (namely, Darjeeling VM [2]).

As briefly discussed in the introduction, the presence of an OS provides a suitable decoupling between HW and SW which makes the approach described in this paper independent of the underlying MCU. Nevertheless, in the following we refer to Texas Instruments' MSP430F54xxa MCUs, which are highly representative of state-of-the-art power-manageable MCUs providing four categories of power states: *Active*, *Standby*, *Sleep*, and *Hibernation*. In Standby mode the CPU is not powered, but the clock system is running and the unit is able to wakeup itself by means of timer interrupts. In Sleep mode both the CPU and the clock system are turned off, so that the unit wakes up only upon external interrupts. In Hibernation even the memory system is switched off, so that there is no data retention and a complete reboot is required at wakeup.

Contiki [1] is a real-time embedded OS particularly suited for sensor nodes. It has an inherent event-driven structure which reduces the overhead of periodic wake-ups by making the interrupt handler aware of the next time at which a process has to be resumed by a timer interrupt. This allows the MCU to go back to sleep without invoking the scheduler in case of premature wakeup. The only exception is represented by processes waiting for external events, which need to be resumed whenever an interrupt arrives.

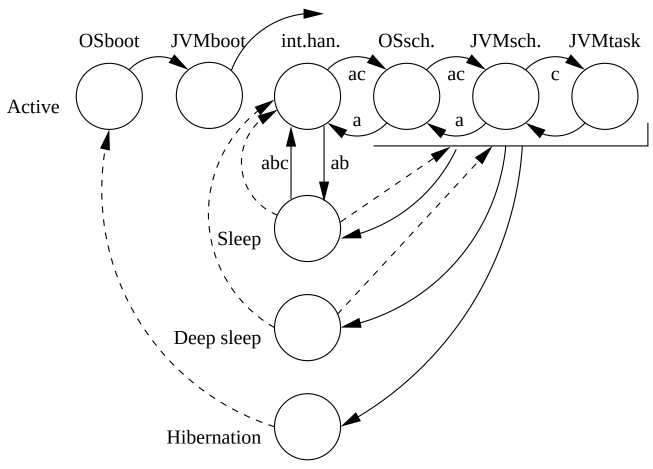

Figure 1. Power state diagram of a power-manageable MCU running the Darjeeling VM on top of Contiki OS.

Darjeeling [2] is a VM for wireless sensor networks which supports a significant subset of the Java libraries while running on 8-bit and 16-bit MCUs with at least 10kbytes of RAM. The VM runs on top of Contiki as a single process, in spite of its multi-threading support. Hence, the VM has its own scheduler to switch among the active threads according to a preemptive round-robin policy. Whenever the running thread is suspended the scheduler waits for the a timer interrupt before resuming execution.

A. Power-manageable virtual sensor node

Figure 1 shows the power-state diagram of a power-manageable MCU running the Darjeeling VM on top of Contiki OS. All the states aligned at the top of the diagram represent the macro steps required at wake-up to resume the execution of a Java thread. Wake-ups can be triggered either by timing interrupts (solid arrows) or by external events (dashed arrows). It is worth noticing that self wake-ups are enabled only in Standby mode, while external events are required to trigger wake-ups from Sleep and Hibernation.

Contiki puts the MCU in Standby whenever all its running processes are waiting for external events or timer interrupts. In order to keep control of the elapsed time it sets a periodic timer interrupt which wakes up the MCU every 10ms to allow the interrupt handler to evaluate if there are processes to be resumed. If this is the case the control is passed to the scheduler, otherwise the MCU is turned off again. As previously stated, the VM needs to be resumed at each timer

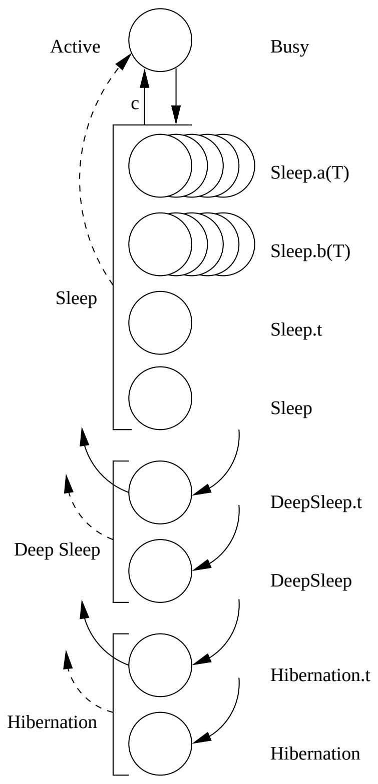

Figure 2. Power state diagram of a power-manageable virtual sensor node [4].

interrupt in order to check for the status of its threads which are not visible from the OS.

Figure 1 makes use of labels a, b, and c to denote the transitions taken upon a timer interrupts in case of: a) VM with no tasks to be resumed, b) OS with no processes to be resumed, and c) virtual task to be resumed. While case c) represents a useful wakeup, cases a) and b) should be avoided in order to save power. To this purpose, a modified software stack was recently proposed [4] which: i) avoids periodic wake-ups by making the OS aware of the time of the next event scheduled by the VM and by tuning the timer interrupt accordingly, ii) supports hibernation by saving and restoring the heap of the VM, and iii) maintains timing information in deeper low power states by means of an external ultra low-power real-time clock.

The modified software stack builds a power manageable virtual sensor node which makes directly available from the Java runtime environment all the power states represented in Figure 2. The first two Standby states are parameterized by the timer interrupt (T) which can be adjusted to explore the tradeoff between power consumption and reactivity. Suffix t is used to denote the usage of the external real-time clock to provide accurate timing information in spite of the lack of the internal clock. Table I reports the power-performance tradeoffs offered by the power states of the software stack

State name	**Power [μW]**	**WUt [ms]**	**WUe [μJ]**	**SDt [ms]**	**SDe [μJ]**
Active	6600μW	n.a.	n.a.	n.a.	n.a.
Standby.a(T)	$4.5 + 153.57/T$	23.41	153.57	–	–
Standby.b(T)	$4.5 + 0.33/T$	23.41	153.57	–	–
Standby.t	$4.5 + 0.3$	23.41	153.57	–	–
Standby	4.5	23.41	153.57	–	–
Sleep.t	*1.5+0.3*	23.41	153.57	–	–
Sleep	*1.5*	23.41	153.57	–	–
Hibernation.t	*0.1+0.3*	560	2312.8	78.8	606.76
Hibernation	*0.1*	560	2312.8	78.8	606.76

Table I
CHARACTERIZATION OF THE POWER STATES OF THE VIRTUAL SENSOR NODE ARCHITECTURE [4] RUN ON A TEXAS INSTRUMENTS' MSP430F2618 POWERED AT 3V AND CLOCKED AT 16MHZ WITH A 4KBYTE VIRTUAL MACHINE HEAP.

run on top of a Texas Instruments MSP430F2618 MCU powered at 3V and clocked at 16MHz. Each inactive state is characterized by: power consumption (Power), wake-up time and energy (WUt and WUe), and by shut-down time and energy (SDt and SDe). Transitions energies (times) lower than 0.01μJ (0.01ms) are not reported. software each power state

B. Communication issues

Communication across the radio channel is handled by the `Radio` class of the Darjeeling VM, which makes available a `receive()` method to be invoked by any Java thread waiting for a message. As soon as the method is invoked, the Java thread is suspended by the scheduler of the VM. If there are no other threads ready to execute, the Darjeeling process is suspended as well and rescheduled by the OS at next timer interrupt (i.e., at most after 10ms). Referring to the state diagram of Figure 1, as long as the message does not arrive, the MCU keeps waking up at each timer interrupt and executing the interrupt handler routine, the OS scheduler, and the VM scheduler before deciding to go back in Standby mode. This is a power-consuming self loop which is labeled with a in Figure 1 and schematically represented by macro state Standby.a(T) in Figure 2. No other low power states can be exploited while waiting for a message.

Looking at Table I, it is worth noticing that power consumption of Standby.a(T) is several orders of magnitude higher than that of Sleep and Hibernation. Moreover, the wake-up time is larger than 10ms, so that the MCU would stay always while waiting for a message unless a longer timer interrupt was set in the modified stack. The minimum timer interrupt which could allow the exploitation of Standby mode is $T = 25ms$.

The event-driven communication library presented in next section solves this issue by enabling the exploitation of all the low-power states of a power-manageable virtual sensor node waiting for incoming messages.

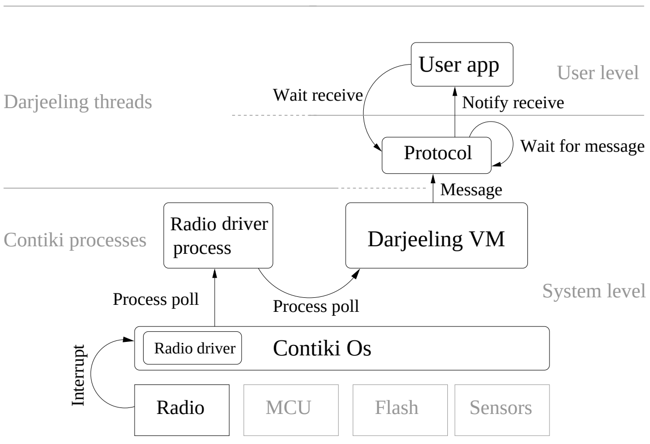

Figure 3. VirtualSense software architecture

III. VirtualSense communication

The software architecture of the proposed communication framework is shown in Figure 3, where arrows are used to represent the event chain triggered by the reception of an incoming packet. The figure points out the interactions between user-level and system-level execution flows, as well as those between Contiki processes (namely, `Radio driver process` and `Darjeeling VM`) and Darjeeling threads (namely, `Protocol` and `User app`).

While waiting for an incoming packet all the processes are blocked and they do not consume any computational resource. When a packet is received by the radio device, the `Radio driver` interrupt handler issues a `PROCESS_EVENT_POLL` for the `Radio driver process` which was waiting for it. At this point the scheduler of Contiki wakes up the `Radio driver process` which: takes the packet from the radio device buffer, forwards it to the Contiki network stack, issues a new `PROCESS_EVENT_POLL` for the `Darjeeling VM`, and releases the CPU while waiting for next packet. The CPU is then taken by the `Darjeeling VM` process, which resumes the execution of the `Protocol` thread which was blocked for I/O. The `Protocol` plays the role of consumer by popping the incoming message from the Contiki network stack, which acts as a buffer in the producer-consumer interaction.

The event chain described so far is general enough to enable the implementation of any kind of communication protocol either within the `Protocol` thread or at application level. Depending on the protocol adopted and on its implementation, received packets can either be handled directly by the `Protocol` thread or be forwarded to the `User app` waiting for them.

Sending a packet is much simpler than receiving it: the `User app` which needs to send a message invokes the `send()` method of the `Protocol`, which puts the packet on the Contiki network stack without involving the `Radio driver process`.

In the following we outline the three packages developed to extend the Darjeeling Java libraries in order to support the event chains described above: i) `javax.virtualsense.radio`, containing the static native methods used to communicate with the radio device; ii) `javax.virtualsense.network`, making communication primitives available to user-level Java threads; iii) `javax.virtualsense.concurrent`, providing synchronization primitives. A simplified class diagram is shown in Figure 4.

A. Radio package

The `radio` package contains the `Radio` class (represented in Figure 4) and some other classes used to handle exceptions. The `Radio` class exports static native methods which directly interact with the platform radio driver and with the network stack of Contiki OS: a method to perform radio device configuration and initialization (`init()`), unicast and broadcast send methods (`send()`, and `broadcast()`), a blocking receive methods (`receive()`, and two methods to get the sender and receiver IDs (`getSenderId()` and `getReceiverId()`). All the methods are protected, in order to be used only through the `Protocol` class, which is part of the `network` package.

Unicast and broadcast send methods make use of the Contiki `unicast_conn` and `broadcast_conn` network connections from the `rime` network stack. The `receive()` method suspends the calling Java thread by putting it in a waiting queue and acquires a lock on the radio device preventing the power manager to shut down the network device. Whenever a radio message is received from the Contiki network stack two different call-backs are activated depending on the nature of the received message: broadcast call-back or unicast call-back. Both call-backs wake up the suspended Java thread, set the `senderId` and `receiverId` attributes, and release the device lock.

B. Network package

The `network` package acts as a middleware layer which lies between the system level radio package and user-level applications. In particular, this package contains an abstract `Protocol` class, providing a communication protocol skeleton, and a `Network` class, providing a public interface to make communication primitives available to user-level threads.

The `Network` implements the *singleton* pattern, so that it can only be instantiated by means of its `init()` method, which can be invoked with or without a given protocol (i.e., an instance of a subclass of `Protocol`. If no protocol is specified, then the `NullProtocol` is used by default. After network initialization, user level threads can call

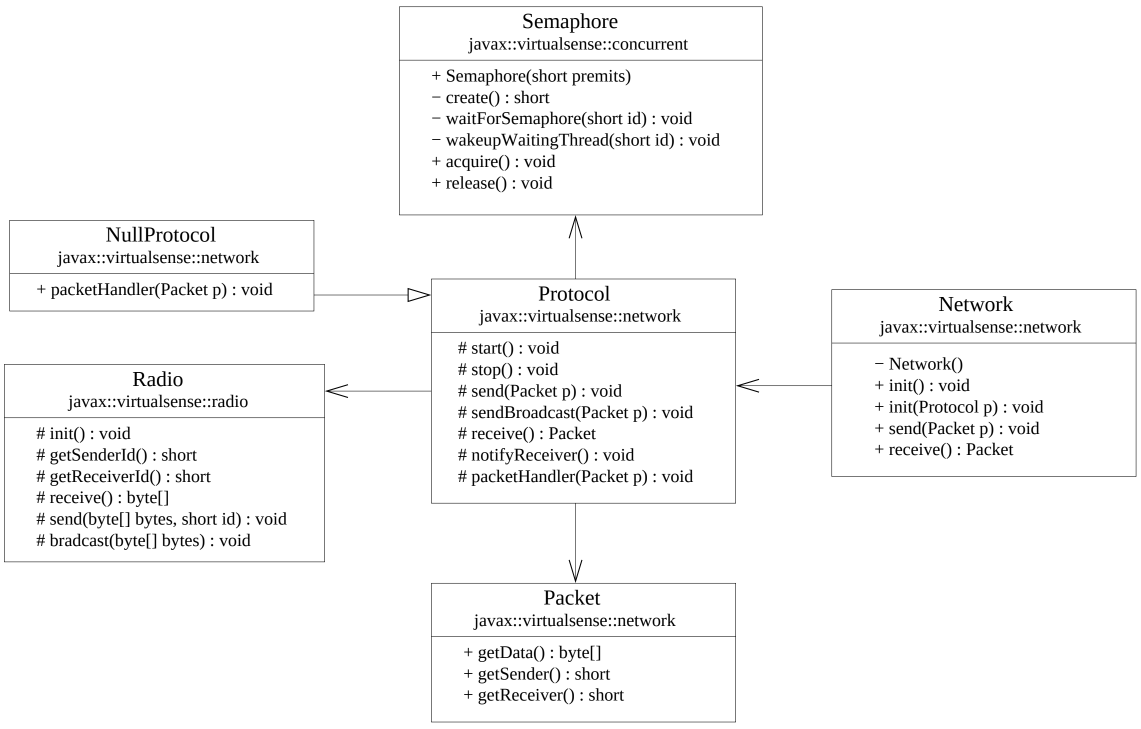
Semaphore
javax::virtualsense::concurrent
+ Semaphore(short premits)
− create() : short
− waitForSemaphore(short id) : void
− wakeupWaitingThread(short id) : void
+ acquire() : void
+ release() : void
NullProtocol
javax::virtualsense::network
+ packetHandler(Packet p) : void
Protocol
javax::virtualsense::network
# start() : void
# stop() : void
# send(Packet p) : void
# sendBroadcast(Packet p) : void
# receive() : Packet
# notifyReceiver() : void
# packetHandler(Packet p) : void
Network
javax::virtualsense::network
− Network()
+ init() : void
+ init(Protocol p) : void
+ send(Packet p) : void
+ receive() : Packet
Radio
javax::virtualsense::radio
# init() : void
# getSenderId() : short
# getReceiverId() : short
# receive() : byte[]
# send(byte[] bytes, short id) : void
# bradcast(byte[] bytes) : void
Packet
javax::virtualsense::network
+ getData() : byte[]
+ getSender() : short
+ getReceiver() : short

Figure 4. VirtualSense communication class diagram. Public methods are denoted means ”+”,l while protected and private methods are denoted by ”#” and ”-”, respectively.

`send()` and `receive()` methods to communicate. These two methods provide a public interface to the corresponding methods of the `Protocol` class.

`Protocol` is an abstract class which has to be subclassed in order to implement specific routing strategies. The class maintains as local properties the routing table and the queue of received packets. In order to decouple system-level and user-level packet reception tasks, the `Protocol` class provides a dedicated thread (instantiated and launched within the class constructor) which runs a loop containing a call to Radio.receive(). The thread is suspended on this call until a packet is received, as described in Section III-A. Upon reception of an incoming packet the thread resumes execution and it calls the `packetHandler()` method, an abstract method that has to be implemented in any Protocol subclass.

Methods `receive()` and `notifyReceiver()` provide the means for using the event-driven reception mechanism from user-level threads. To this purpose, an application which needs to receive a packet from the radio device invokes the `Network.receive()` method which, in turn, calls `Protocol.receive()` which suspends the calling thread on a counting semaphore. Upon reception of a packet to be forwarded to the waiting application, the `Protocol` invokes `notifyReceiver()` to release a permit on the semaphore. From the implementation stand point, the invocation of `notifyReceiver()` has to be placed inside `packetHandler()`, which is the method where the actual routing protocol is implemented. The default `NullProtocol` does nothing but invoking this method to forward to the applications all incoming packets.

C. *Concurrent package*

The `concurrent` package provides a robust and efficient way to manage thread synchronization. In particular the `Semaphore` class implements a standard counting semaphore based on a waiting queue. Any thread waiting for a semaphore permit is suspended by the VM and moved to the semaphore waiting queue. In this way it allows the power manager to shutdown the MCU. As soon as a new permit is available on the semaphore, the waiting thread is woken up by removing it from the waiting queue. Thread suspension and wake up are implemented through native methods `waitForSemaphore()` and `wakeupWaitingThread()`, respectively, which directly interact with the VM scheduler and manage thread displacement.

IV. Case Study and Experimental Results

In this section we show, with a practical example, how to use the communication library presented in Section III. We use as a case study a sensor network programmed to perform a periodic monitoring task: each node in the network senses the target physical quantity once per second and sends the measured value to the sink. The sink is nothing but a sensor ndoe connected to a desktop PC by means of the serial port. All other sensor nodes act also as routers, implementing a self-adapting minimum path routing protocol.

```
01 import javax.virtualsense.network.*;
02
03 public class MinPathProtocol extends Protocol{
04
05   private short minHops = Short.MAX_VALUE;
06   private short epoch   =  0;
07
08   protected void packetHandler(Packet p){
09      if(p instanceof InterestMsg){
10         InterestMsg interest = (InterestMsg)p;
11         if(interest.getEpoch() > this.epoch){
12            this.epoch = interest.getEpoch();
13            super.bestPath = -1;
14            this.minHops = Short.MAX_VALUE;
15         }
16         if(interest.getHops() < this.minHops){
17            this.minHops = interest.getHops();
18            super.bestPath = interest.getSender();
19            interest.setHops(interest.getHops()+1);
20            super.sendBroadcast(interest);
21         }
22      }else if(p instanceof DataMsg) {
23         DataMsg data = (DataMsg)p;
24         if(data.toForward())
25            super.send(data);
26         else
27            super.notifyReceiver();
28      }
29  }// end method
30 }//end class
```

Figure 5. Minimum path algorithm implementation

The sink collects all the measurements and triggers period updates of the routing tables by sending an broadcast *interest* message (`InterestMsg`) to the network according to a *directed diffusion* paradigm [9]. The interest contains a progressive counter, called `epoch`, which is used by the nodes which receives and forward the interest message to verify its freshness. In addition, it contains the number of hops from the sink, which is incremented at each hop to allow sensor nodes to identify the best path. Figure 5 reports the Java code of the `MinPathProtocol` class which extends the `Protocol` and overrides abstract method `packetHandler()` to implement the minimum path directed diffusion algorithm.

Whenever a new packet is received, the `packetHandler()` checks if it contains an interest message (Figure 5, line 09) or a data message (line 22). In case of an interest, its epoch is compared with the previous one (line 11) in order to reset the routing table in case of new epoch (lines 12-14). In the directed diffusion min path protocol the routing table is nothing but the ID of the neighboring node along the best path to the sink. Such an ID is stored in `bestPath`, which is updated with the ID of the sender of last interest message whenever the number of hops anotated in the message is lower than the current value of `minHops` (lines 16-19). In this case the interest message is also forwarded (line 20).

Data packets are either to be forwarded to the sink through `bestPath` (line 25) or to be notified to user-level applications possibly waiting for them (line 27). According to the directed diffusion algorithm sensor nodes never play the role of recipients of data messages. Nevertheless, line 27 has been added in Figure 5 as an example of user-level communication.

A. *Performance overhead*

The proposed architecture was instrumented in order to measure the software overhead introduced by the high-level implementation of the communication protocol.

In particular we measured Contiki and Darjeeling execution times as separate contributions to the reception event-chain starting from the sleep state. Contiki overhead was measured as the time between the reception of a radio interrupt and the corresponding Darjeeling VM process poll. Darjeeling overhead was measured as time between the wake up of Darjeeling VM process and the delivery of the incoming packet to the user-level application. The results obtained in a prototyping board equipped with an MSP430F5435a running at 16MHz where respectively 3.7ms and 14.4ms for Contiki and Darjeeling software overheads resulting in a total overhead of 18.1ms.

On the other hand the software overhead introduced by the proposed Java library in the sending chain was of 3.4ms.

It is worth mentioning that the Java library introduced in this paper allows the power manager to exploit the waiting times to put the sensor nodes either in Sleep or in Hibernation, depending on the DPM policy adopted. For instance, in case of a sensor node acting as a router handling on average two data packets per second, the exploitation of the Sleep state would reduce the power consumption from the 6600μW of the Active state of our MCU to a measured value of 1620μW, which would tend to a few μW as the monitoring rate decreases.

This example shows how the proposed network library allows the programmer to implement a routing protocol with a few lines of code, enabling the full exploitation of the low-power states of the MCU without impairing the reactivity of the sensor node.

V. CONCLUSION

The availability of ultra low-power MCUs able to run a virtual machine makes it possible to design power manageable sensor nodes that can be programmed in Java. The separation between the virtual runtime environment and the underlying MCU has been recently bridged by means of a modified software stack which retains the key benefits of virtualization without impairing the effectiveness of dynamic power management. Any node in a wireless sensor network, however, spends most of its operating time waiting for incoming packets. In spite of the idleness of a waiting node, the capability of reacting to an incoming message is often implemented by means of polling mechanisms which keep the MCU busy and avoid the exploitation of its low-power inactive states. This is what happens with the communication primitives made available by the Darjeeling VM running on top of Contiki OS.

This paper has presented an event-driven communication library, called VirtualSense, which provides energy-aware `send()` and `receive()` methods which allow the MCU to exploit inactive low-power states while waiting for incoming packets and to resume the execution of the Java thread as soon as a packet is received. A simple case study has been presented to show that the proposed library makes it possible (and easy) to implement a communication protocol on top of the Java runtime environment without impairing the effectiveness of dynamic power management in ultra low-power sensor nodes.

ACKNOWLEDGMENT

The authors would like to thank Andrea Seraghiti, Massimo Zandri, and NeuNet Cultural Association (http://www.neunet.it/) for their fundamental contribution to the development of the prototype.

REFERENCES

[1] A. Dunkels, B. Gronvall, and T. Voigt, ”Contiki - a lightweight and flexible operating system for tiny networked sensors”, in *Proc. of the IEEE Conf. on Local Computer Networks*, pp. 455-462, 2004.

[2] N. Brouwers, P. Corke, and K. Langendoen. ”Darjeeling, a Java compatible virtual machine for microcontrollers”, in *Proc. of the ACM/IFIP/USENIX Middleware Conference Companion*, pp. 18-23, 2008.

[3] Texas Instruments. ”MSP430x5xx/MSP430x6xx Family User’s Guide”, URL *http://www.ti.com/lit/ug/slau208j/slau208j.pdf*, 2012.

[4] E. Lattanzi and A. Bogliolo, ”Ultra-Low-Power Sensor Nodes Featuring a Virtual Runtime Environment”, to be presented at IEEE ICC E2NETS-2012, 2012.

[5] E. Lattanzi and A. Bogliolo, ”WSN Design for Unlimited Lifetime”, In Yen Kheng Tan (Ed.), *Sustainable Energy Harvesting Technologies: Past, Present and Future*, InTech, 2011.

[6] Rene Muller, Gustavo Alonso, and Donald Kossmann. 2007. ”A virtual machine for sensor networks”. SIGOPS Oper. Syst. Rev, pp. 145-158, 2007

[7] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. ”Java on the bare metal of wireless sensor devices: the squawk Java virtual machine”. In Proceedings of the 2nd international conference on Virtual execution environments (VEE ’06). ACM, pp. 78-88, 2006.

[8] Philip Levis, David Gay, and David Culler. ”Active sensor networks”. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*, pp. 343-356, 2005.

[9] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. ”Directed diffusion for wireless sensor networking”. IEEE/ACM Trans. Netw. 11, pp. 2-16, 2003.