Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2014, Article ID 814510, 9 pages
http://dx.doi.org/10.1155/2014/814510

Research Article

Supporting Preemptive Multitasking in Wireless Sensor Networks

Emanuele Lattanzi, Valerio Freschi, and Alessandro Bogliolo

Department of Basic Sciences and Foundations, University of Urbino, Piazza della Repubblica 13, 61029 Urbino, Italy

Correspondence should be addressed to Emanuele Lattanzi; emanuele.lattanzi@uniurb.it

Received 12 April 2013; Revised 19 December 2013; Accepted 20 December 2013; Published 6 February 2014

Academic Editor: Frank Ehlers

Supporting the concurrent execution of multiple tasks on lightweight sensor nodes could enable the deployment of independent applications on a shared wireless sensor network, thus saving cost and time by exploiting infrastructures which are typically underutilized if dedicated to a single task. Existing approaches to wireless sensor network programming provide limited support to concurrency at the cost of reducing the generality and the expressiveness of the language adopted. This paper presents a java-compatible platform for wireless sensor networks which provides a thorough support to preemptive multitasking while allowing the programmers to write their applications in java. The proposed approach has been implemented and tested on top of VirtualSense, an ultra-low-power wireless sensor mote providing a java-compatible runtime environment. Performance and scalability of the solution are discussed in light of extensive experiments performed on representative benchmarks.

1. Introduction

Multitasking is a method which allows multiple tasks to be concurrently performed in the same time period by a shared processing unit. In microprocessor systems, the *scheduler* of the operating system (OS) decides to which task the shared resources have to be assigned at any given time in order to maximize the overall throughput and meet the performance constraints of the applications. While multitasking is common practice in general-purpose computer systems, it is not commonly supported in *wireless sensor networks* (WSNs) made of lightweight nodes subject to tight power and resource constraints.

On the other hand, the growing need for context awareness and ambient intelligence is rapidly pushing the deployment of WSNs and the development of applications designed to run on top of them. Supporting the concurrent execution of multiple applications on a shared WSN is an attractive perspective in terms of cost sharing, deployment time, and sustainability. In principle, a multitasking WSN could be deployed once and for all in a given place and then made available to end users and software developers as a common platform to run any kind of context-aware applications, possibly focused on different physical quantities, targeting different portions of the network, and belonging to different users. This can be useful not only to deploy shared test beds allowing different research groups to run comparative experiments, but also to enable a thorough sharing of real world sensor networks among different users independently running their own tasks. For instance, a WSN equipped with CO_2 sensors could concurrently execute tasks to collect data about air quality, control HVAC equipment, trigger fire alarms, and estimate the number of people in a room. The applications could be independently developed by different users and dispatched to the same nodes.

Such a scenario requires a specific support both at network level (dynamic code deployment) and at node level (dynamic application management and concurrent execution). Dynamic code deployment in WSNs has been an active research topic for many years, leading to several suitable solutions that are currently available in the market [1]. Node-level concurrency, on the contrary, has received much less attention so far and no ultimate solutions have been proposed. Existing approaches can be classified into two

main categories: WSNs managed as acquisitional distributed databases [2, 3] and WSNs based on scripting languages [4–7]. Both of them provide some form of concurrency at the cost of reducing the flexibility and the expressiveness of the query/scripting language adopted.

This paper presents a java-compatible WSN platform providing a thorough support to node-level multitasking without asking the programmers to relinquish the benefits (in terms of usability, portability, and expressiveness) of high-level object-oriented programming. The platform has been implemented as an extension of *VirtualSense* [8], an ultra-low-power wireless sensor mote featuring a java-compatible runtime environment.

The rest of the paper is organized as follows: Section 2 provides a survey of existing approaches to multitasking in WSNs; Section 3 provides a minimum background on VirtualSense and on the components of its software stack; Section 4 presents the programming model; Section 5 outlines the proposed architecture; Section 6 reports and discusses experimental results; Section 7 concludes the work.

2. Related Work

The state of the art of multitasking WSNs can be illustrated by referring to the broader research area of WSN programming, where existing approaches can be classified into three main categories, depending on their granularity: (i) node-level approaches, (ii) group-level approaches, and (iii) network-level approaches [9].

Group-level and *network-level* programming models provide a set of programming primitives to handle either a group of nodes or the entire network as a single abstract programmable machine. Acquisitional distributed database systems, such as Cougar and TinyDB, fall in this category [2, 3]. The WSN is seen at application level as a database managed by a distributed query processor which runs on the nodes. The sensing/monitoring tasks are expressed as acquisitional queries that have to be written in a declarative SQL-like language. In this scenario, concurrency is provided to some extent by the gateway, which combines the queries possibly coming from different users before launching them on the network.

Node-level programming models are mainly based on ad hoc abstractions of the underlying hardware, conceived to grant to the programmer a suitable control of it. Representative examples of this kind of platforms are provided by *TinyOS* [10] and *Contiki* [11], programmed, respectively, in *nesC* and *C*, which have become *de facto* standards for sensor network programming. Both of them support multithreaded applications, but they do not provide specific support to interapplication concurrency.

The most common approaches to node-level concurrency are based on context-specific execution environments with their own scripting languages for application programming. In particular, *SensorWare*, *Agilla*, and *Servilla* are agent-based platforms which support the concurrent execution of scripting-defined agents [4, 6, 7]. Similarly, concurrency is supported by *Melete* [5], an extension of the *Maté* project which provides a Virtual Machine (VM) capable of executing *TinyScript* applications.

All the above-mentioned solutions suffer from several limitations which prompt further research efforts in the field of multitasking WSN: (i) they make use of domain-specific programming languages with limited expressiveness, (ii) they provide limited degrees of freedom in terms of communication protocols, (iii) they do not support multithreaded services/applications, and (iv) they do not guarantee cross-platform portability of the code.

A significant step towards the preemptive multitasking support for WSNs has been recently made by *SenSmart* [12], a multitasking operating system for sensor networks which rewrites and recompiles in a single binary all the *NesC* applications to be executed concurrently with the patches needed to guarantee appropriate preemptive multitasking semantics. Binary rewriting is performed on the base station and the executable file is then dispatched to the sensor nodes for execution. *SenSmart* eludes classification into node-/group-level approaches in that it performs group-level preliminary operations to prepare the executable to be run on each node.

This paper makes a step forward towards node-level multitasking by providing a thorough support to preemption on top of a java-compatible virtual machine, thus making concurrency completely transparent to the programmer without any code patching or rewriting.

3. Background

This section provides a minimum background on VirtualSense [8], the HW/SW platform adopted in this paper, and on the two main components of its software stack, namely, Contiki [11] and Darjeeling [13].

3.1. VirtualSense. VirtualSense is an open-hardware ultra-low-power sensor module featuring a java-compatible virtual runtime environment. VirtualSense is based on Contiki OS [11] and on the Darjeeling VM [13], suitably modified in order to make it possible for a java programmer to fully exploit the low-power states of the underlying microcontroller unit, a Texas Instruments' MSP430F5418a in the latest version [8].

There are four categories of power states made available by VirtualSense: *active*, which is the only state in which the CPU is running; *standby*, where the CPU is not powered, but the clock system is running and the unit is able to wake up itself by means of timer interrupts; *sleep*, where both the CPU and the clock system are turned off, so that the unit wakes up only upon external interrupts; and *hibernation*, where even the memory system is switched off, so that there is no data retention and a complete reboot of the OS is required at wake-up together with a restore of the VM heap.

The average power consumption of VirtualSense is of about 13 mW in processing mode and 66 mW in transmit mode, while the consumption reduces to 14.67 μW in standby mode, 1.32 μW in sleep mode, and 0.36 μW in hibernation, with wake-up times ranging from 25 ms (from standby and sleep) to about 500 ms (from hibernation). VirtualSense

motes can execute typical WSN tasks with an average power consumption of a few micro Watts.

VirtualSense provides an API, made available by the `PowerManager` class, which makes it possible for java programmers to control the power states of the MCU and of all the power manageable peripherals (refer to Section 5.5 for details).

3.2. Contiki. Contiki is a real-time embedded OS particularly suited for sensor nodes [11]. It provides 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 wake-up.

The only inefficiency of this reactive behavior is represented by processes waiting for external events, which need to be resumed whenever an interrupt arrives, regardless of the nature of the event, in order to be possibly ready to process it.

The Contiki programming model is based on *protothreads*. A protothread is a memory-efficient programming abstraction that combines features of both multithreading and event-driven programming to attain a low memory overhead. The kernel invokes the protothread of a process in response to an internal or external event. Contiki does not enforce preemption; rather, it assumes protothreads to be cooperatively scheduled. This means that a Contiki process must always explicitly yield control back to the kernel at regular intervals by invoking either `PROCESS_WAIT_EVENT ()` or `PROCESS_YIELD()`.

3.3. Darjeeling. Darjeeling is a VM for wireless sensor networks which supports a significant subset of the java libraries with minimum hardware requirements: 8 bit/16 bit MCUs with at least 10 kbytes of RAM [13].

The VM supports execution of a single java application at a time, which can instantiate multiple threads. Darjeeling VM is implemented as a single Contiki process equipped with its own scheduler, switching among the active threads according to a time-sliced Round-Robin policy. In particular, whenever the running thread exhausts its time quantum it is suspended to allow the scheduler to resume execution and to grant the resources to another thread.

4. Programming Model

Sensor nodes are usually devoted to run one application at a time. Sensor nodes which provide a built-in runtime environment execute a script taken either from the executable nonvolatile memory or from an external eeprom, while *platform-centric* nodes [9] require the application to be stored on the executable nonvolatile memory and linked (either statically or dynamically) to the firmware. In both cases, *over-the-air* (OTA) programming allows the application to be replaced whenever needed [1].

In VirtualSense, an application consists of one or more java files containing at least a class implementing the entry-point method signature which is `public static void motemain()`. Byte code verification and transformation are done off-line by a tool called *Infuser*, which takes in input multiple class files and statically links them in loadable modules called *infusions*. VirtualSense exploits this paradigm to compile into a single infusion file the entire application, which can be made of several class files and possibly contains several threads. The infusion file is ready to be transferred, installed, and executed on the target nodes.

```
(1) import javax.virtualsense.actuators.Leds;
(2)
(3) public class Blink
(4) {
(5)
(6)   public static void motemain()
(7)   {
(8)        boolean state=true;
(9)        while(true)
(10)       {
(11)          for (int i=0; i<3; i++)
(12)          {
(13)               Leds.setLed(i,state);
(14)               Thread.sleep(1000);
(15)          }
(16)          state=!state;
(17)       }
(18)   }
(19) }
```

Algorithm 1: Java code of a simple blink application.

The java code of a simple led-blinking application is reported in Algorithm 1. Compared to a standard java application the only change concerns the signature of the `main()` method that must be `motemain()`.

Multiple applications (i.e., infusion files) can be simultaneously installed and executed on the same node. Any application instance running on top of the VirtualSense VM is called *Task*. The multitasking support provided by VirtualSense allows multiple tasks to execute concurrently and share the CPU time. In particular, each java thread belonging to a task is mapped into a *native thread* following the *one-to-one* mapping model. Each native thread is then scheduled by the VM as a single entity which competes for taking the CPU according to a typical *system contention scope*. This results in a general concurrency among all java threads belonging to all tasks.

The programming model is schematically shown in Figure 1, where tasks are represented on top of a layered architecture composed of (i) the VirtualSense Hardware, (ii) the Contiki OS, (iii) the VirtualSense runtime, including the Darjeeling VM and all the core libraries of VirtualSense which are not part of the API, and (iv) the VirtualSense libraries, made available to the applications through specific APIs.

Arrows represent applications and commands which can be received and possibly forwarded at each node. Although

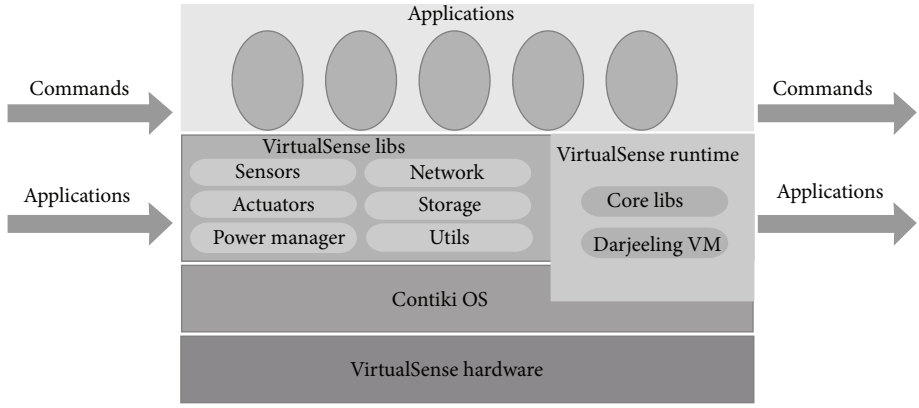

Figure 1: Schematic representation of the programming model of VirtualSense.

over-the-air programming is out of the scope of this work, the programming model is general enough to support both the viral diffusion of the same application across the WSN and the selective installation on target nodes. Once the applications are installed on the target nodes, the commands provide a remote interface to load, launch, stop, and unload them, as detailed in the following section. It is worth noting that commands can be issued either by the user (through the gateway) or by another application possibly running on a different node.

5. Architecture Design and Implementation

The isolation and protection needs of the concurrent tasks running on the same node require the platform to provide to each task a dedicated execution space, a dedicated network interface, and a preemptive task scheduler. The multitasking support of VirtualSense provides

(1) an application manager able to install and remove applications from a nonvolatile memory space,

(2) a task manager which can load, start, stop, and unload a task,

(3) a network dispatcher which manages network connections and dispatches network data to/from the appropriate thread belonging to a particular task,

(4) a preemptive task scheduler capable to interrupt running tasks and implementing a CPU-time sharing policy.

Figures 2, 3, and 4 provide a schematic representation of the VirtualSense architecture, where Contiki processes are represented as circles, VirtualSense tasks are represented as dashed rectangles, and java threads are represented as dark rectangles. Four Contiki processes are shown in the figures: the application manager, the task manager, the network dispatcher, and the Darjeeling process, with four user-level tasks running on top of it: `Task 0`, `Task 1`, and `Task n` together with the `VirtualSense Main Task`. For the sake of explanation, the three figures report the same scheme with different highlights, corresponding to the focus of the following subsections.

5.1. Application Manager. The VirtualSense application manager is highlighted in Figure 2. It is a Contiki process which receives through the network the applications (i.e., infusion files) to be installed or upgraded on the node and stores them in the application memory. A table is used to maintain application data, including ID, name, version, state, and address of the infusion file. In current implementation both the infusion files and the application table are stored in the flash memory of the MCU.

5.2. Task Manager. Once an application has been installed on the node, the task manager is responsible for loading it into the VM (`LOAD`), starting (`START`) and stopping (`STOP`) execution, and unloading (`UNLOAD`) it from the VM. The task manager is implemented as a separate Contiki process which is listening to the network for incoming commands. Whenever a command is received, it is notified to the `VirtualSense Main Task` which executes it, while an execution response is sent back by the task manager. The task management chain is highlighted in Figure 3. In case of a `LOAD` command, the `VirtualSense Main Task` reads from the application table the memory address of the corresponding infusion file. If a valid entry is found, the application is loaded in main memory and a new task is created (`create task`). A `START` command causes the execution of the `motemain()` method of the corresponding task, while a `STOP` command breaks the execution of the task. Since each task can have more than one children thread, the stop command is forwarded to all of them. Once a task has been stopped, it can be either unloaded from main memory by means of an `UNLOAD` command or restarted from scratch by issuing of a new `START` command.

5.3. Network Dispatcher. The VirtualSense network dispatcher, shown in Figure 4, is responsible for managing communication between the tasks running on top of the VM and the underlying network interface. In particular, each user-level thread which needs to send or receive a packet through the network has to ask the network dispatcher to initialize the connection (`network init`). Network initialization is nothing but a dynamic allocation of a network port to

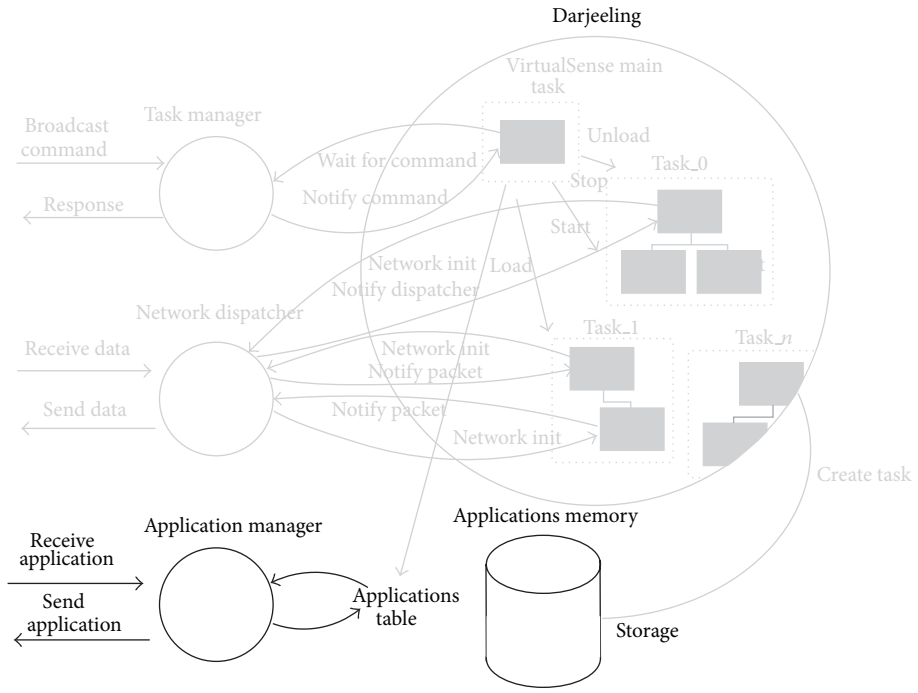

Figure 2: Abstract representation of the VirtualSense architecture highlighting the application manager interacting with the application table and memory.

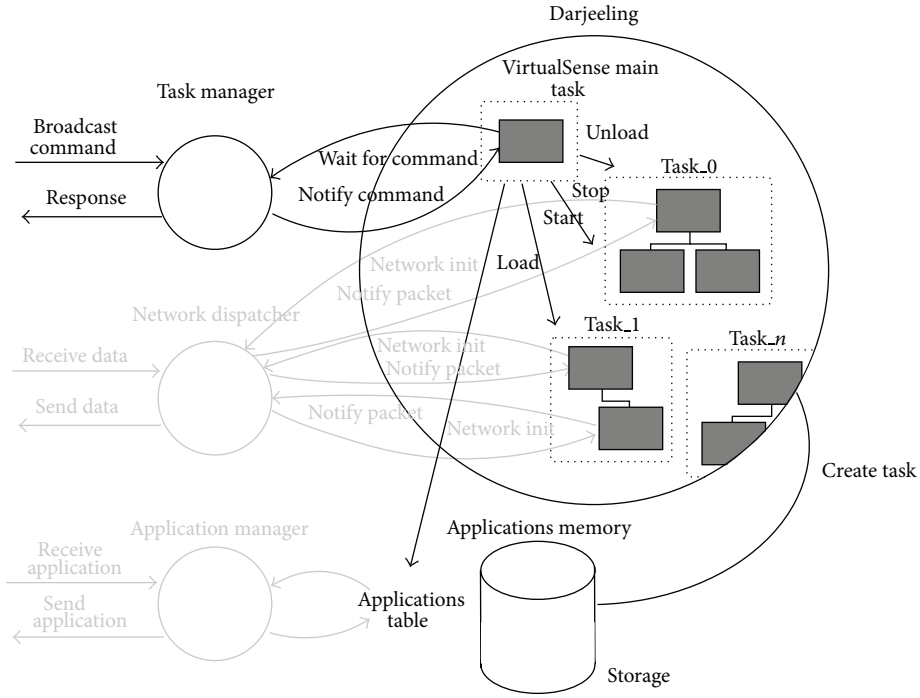

Figure 3: Abstract representation of the VirtualSense architecture highlighting the task manager and several running tasks.

a given thread in order to provide a private communication channel on top of a shared interface. The network dispatcher maintains a port assignment table and notifies the target thread (`notify packet`) whenever a network packet is received on the port assigned to it.

5.4. Preemptive Scheduler. VirtualSense schedules its running tasks by means of a preemptive time-sliced Round-Robin policy (RR). Preemption is performed by the VM by interrupting the execution of a task after a fixed number of bytecodes. The number of bytecodes determines the average

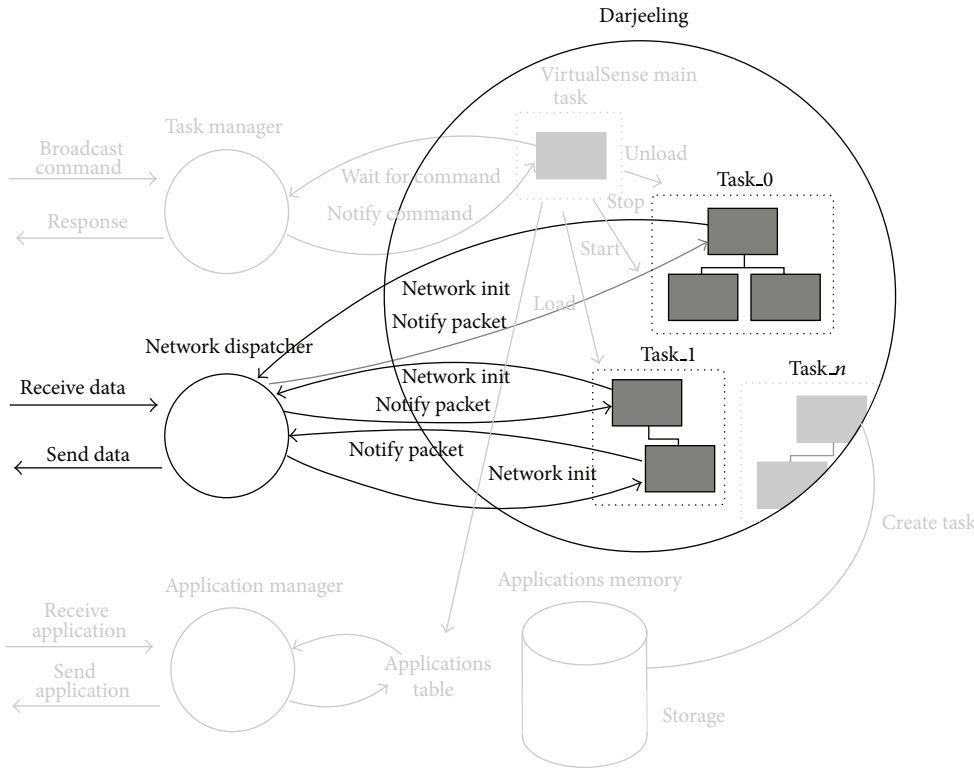

Figure 4: Abstract representation of the VirtualSense architecture highlighting the network dispatcher.

frequency of preemption points. Due to the variable time required to execute each bytecode, the corresponding time slice is not a constant. In traditional desktop systems, this is overcome by rigidly defining the time slice by means of a timer interrupt which, in case of VirtualSense, would cause a higher overhead. At each preemption point, the state of the current task is saved and a new scheduling decision is taken which can lead to a context switch.

5.5. Exploitability of Low-Power Modes. As described in Section 3.1, VirtualSense provides several low-power modes enabling the full exploitation of the MCU low-power states. To make it possible to develop advanced DPM algorithms while working on top of the VM, VirtualSense provides a `PowerManager` class which exports methods for changing the MCU frequency and for triggering transitions to any low-power state, possibly specifying the wake-up time and setting an external RTC to issue self-wake-ups. In the same way, VirtualSense provides java methods to control the wireless communication interface and all the power-manageable modules of the MCU, namely, the serial peripheral interface (SPI), the interintegrated circuit (I2C), the universal asynchronous receiver/transmitter (UART), and the analog-to-digital converter (ADC).

In a traditional mote without multitasking support, the running task is allowed to manage the entire platform by choosing the appropriate power mode and by turning on and off the modules according to its needs. In order to support multitasking, power management in VirtualSense has been deeply modified in order to grant to all running tasks concurrent access to the power-manageable resources. In particular, each power-manageable resource is assigned with a power-state *counting lock* which has to be acquired by each task before it starts to use it and released as soon as it does not need it any more. The VM scheduler, at each invocation, launches a software routine which tries to put each power manageable resource into its deepest low-power state compatible with the counting locks. In particular, it is possible for a resource to enter a low-power state only if there are no higher power states with nonnull counting locks. The counting lock acquisition/release mechanisms are embedded into the native methods used to handle power-manageable resources, so that they are transparent to the java programmer.

Current implementation supports the following locks: `ACTIVE_lock`, `STANDBY_lock`, `SLEEP_lock`, `RADIO_lock`, `ADC_lock`, `UART_lock`, `MAC_lock`, `I2C_lock`, and `SPI_lock`. Each native thread in its running state automatically locks the active state in order to prevent the scheduler fron putting the MCU in any low-power mode. Whenever a running thread suspends its execution by means of a `Thread. sleep` invocation, it automatically releases the `ACTIVE_lock`, which is decremented. If the `ACTIVE_lock` reaches zero, the VM scheduler is allowed to test the lock of next low power state, namely, `STANDBY_lock`, and so on.

6. Experimental Results

This section describes the benchmarks used to characterize the VirtualSense platform and discusses the results of performance measurements made on an instrumented prototype of the VirtualSense mote.

The experiments were designed to demonstrate the capability of VirtualSense to support concurrent execution of multiple tasks on top of ultra-low-power sensor nodes. Experimental results show that the behaviour of the proposed platform is consistent with that of typical general-purpose concurrent systems. In the context of WSNs, although expected, results are not obvious because of the tight energy and computational constraints of the motes. Remarkably, the experiments reported in the following subsections provide evidence that Virtualsense motes make it possible to simultaneously share computational and sensing resources among multiple tasks as it usually happens in multitasking computer systems.

6.1. Benchmarks. The performance of the multitasking support provided by VirtualSense was evaluated on a set of synthetic benchmarks representative of three classes of typical WSN applications: (i) periodic tasks; (ii) CPU-bound tasks; (iii) reactive tasks.

Periodic tasks are common monitoring tasks which sample a physical quantity of interest by periodically reading a value from a sensor. The java code of the monitoring task is reported in Algorithm 2. The `executeMonitor` method which is called the `motemain` of the benchmark is parameterized in the sampling period and in the number of total samples to read. The execution of the corresponding task completes when the `executeMonitor` method returns. This task makes use of the CPU at every period just for the time required to perform the read and then schedules an autowake-up after `period` milliseconds and goes to sleep.

CPU-bound tasks are computation intensive tasks the performance of which is mainly affected by the performance of the CPU. In our benchmark suite CPU-bound tasks are represented by an implementation of the MD5 message-digest algorithm, a cryptographic hash function that takes in input a string and returns a 128 bit hash [14]. Each task computes the MD5 hash function of a given set of strings. Without a preemptive scheduler each task, once started, would keep the CPU busy until the end of its execution.

Reactive tasks spend most of the time waiting for incoming events without keeping the CPU busy. In our benchmark suite, reactive tasks are represented by a single thread task waiting for incoming network packets. Each thread of a task is assigned with a port and it does nothing but notifying the reception of a packet on its port by blinking a led.

6.2. Performance Measurements. The performance achieved by a multitasking system in the execution of a task is usually expressed in terms of *turnaround time*, which takes into account both the time in which the task is running (processing time) and the time in which it is waiting to be executed (waiting time). In the instrumented prototype of VirtualSense, turnaround measurements are directly computed by the scheduler of the Darjeeling virtual machine as the time interval between task submission and task completion.

```
(1) public int executeMonitor(int samples,
(2)                           int period)
(3) {
(4)     int avg = 0;
(5)     for(int i = 0; i < samples; i++){
(6)           avg+=Temperature.getValue();
(7)           Thread.sleep(period);
(8)     }
(9)     return avg/samples;
(10) }
```

Algorithm 2: Java code of the monitor task used in the case study.

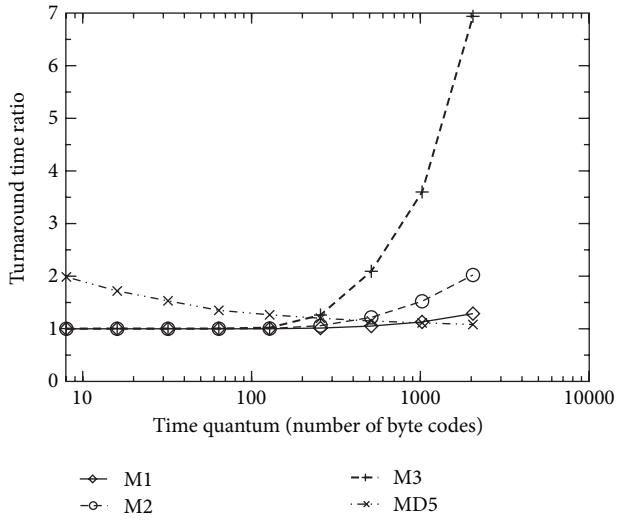

Figure 5: Turnaround time versus the size of the time quantum.

Since the turnaround time of a task is affected by the concurrent execution of other tasks on the same node, the relative impact of multitasking on the performance of a WSN application can be expressed in terms of *turnaround time ratio*. For a given benchmark, composed of a pool of concurrent tasks, the turnaround time ratio of a specific task is the ratio between two measurements of its turnaround time, measured with and without the concurrent tasks.

Figure 5 reports the turnaround time ratio as a function of the scheduler time quantum, expressed as the number of bytecodes to be executed before preempting a task. Data refer to a pool of 6 tasks: three monitoring tasks with sampling periods of 200 ms (M1), 50 ms (M2), and 10 ms (M3) and three instances of an MD5 task.

As expected, the turnaround time of the CPU-bound tasks decreases for increasing values of the time quantum because of the reduced number of context switches they are subject to. At the contrary, monitoring tasks are negatively affected by the size of the time quantum, since the reduced

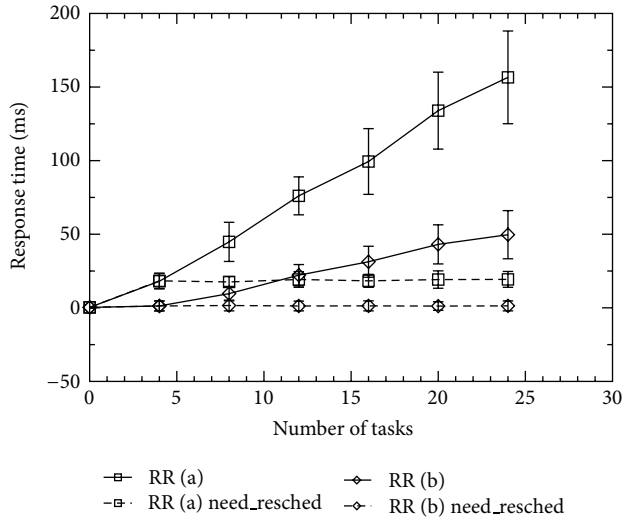

Figure 6: Response time versus number of running tasks.

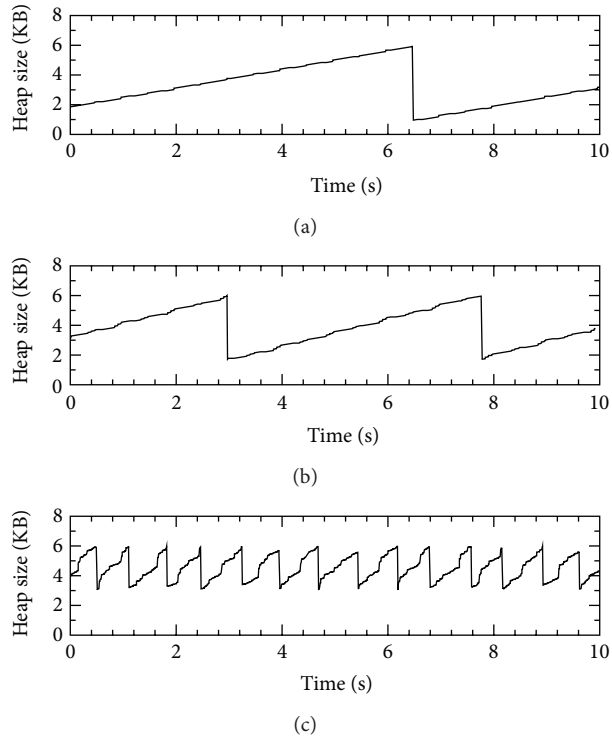

Figure 7: Heap size plotted during tasks execution while varying the number of running tasks: (a) 3 running tasks; (b) 6 running tasks; (c) 12 running tasks.

density of preemption points makes it hard for the tasks to take the CPU exactly when needed to sample the target sensor. As a result, monitoring tasks fail to meet their sampling deadlines and they complete their execution in a longer time. This effect gets worse as the sampling rate increases, so that M3 is the task with the worst performance.

In order to test the impact of multitasking on the responsiveness of a sensor node, a reactive task was installed and run on the mote together with an increasing number of concurrent CPU-bound tasks. The VirtualSense VM was instrumented to measure the response time of the task as the difference between the time at which it obtains the CPU and the timestamp of the triggering event (i.e., an incoming network packet). Figure 6 plots the response time of the reactive task as a function of the number of concurrent CPU-bound tasks. Each point represents the sample average of 10 repeated measures, the standard deviation of which is also reported in the graph. The experiments were repeated for two different values of the time quantum: (a) 1024 bytecodes and (b) 256 bytecodes. Solid lines refer to the results achieved on a VM implementing a pure Round-Robin scheduling policy. As expected, the greater the number of the concurrent tasks, the greater the response time of the reactive task. Moreover, for a given number of concurrent tasks the impact of multitasking on the response time is proportional to the size of time quantum.

Dashed lines report the results of the same experiments repeated by using the `need_resched` flag to tell the scheduler that the task has received an asynchronous event, so that it needs to be rescheduled as soon as possible to process it, in spite of the RR policy. The use of the `need_resched` flag significantly reduces the response time, making it almost independent of the number of concurrent CPU-bound tasks. In fact, the delay incurred by the reactive task in this case depends only on the time quantum, which is the time interval between preemption points.

The size of the heap of the VM was monitored during the experiments to evaluate the scalability of the proposed approach in terms of system requirements. Figure 7 shows how the size of the heap varies over time in case of (a) 3 running tasks, (b) 6 running tasks, and (c) 12 running tasks. When there are more concurrent tasks, the heap size grows faster and the garbage collector intervenes more often to clean up the memory. It is also worth noting that the total amount of space which can be freed by the garbage collector depends on the number and size of the running tasks, so that the baseline of Figure 7(c) is significantly higher than that of Figures 7(a) and 7(b).

7. Conclusion

This paper has presented a modified version of the VirtualSense [8] software stack which provides a thorough support to preemptive multitasking in ultra-low-power wireless sensor networks. VirtualSense features a java-compatible virtual machine which allows the programmer to write applications in java, taking advantage of a specific API which grants full control of the power management features of the underlying hardware. The multitasking support presented in this paper is completely transparent to the programmer, thus maintaining full compatibility with existing VirtualSense applications and with the multitasking nature of the Darjeeling VM. Tasks can be dispatched to the target nodes over the air as *infusion*

files and then remotely controlled by issuing across the network LOAD, RUN, STOP, and UNLOAD commands which are executed by the task manager running on each node. Preemption is transparently managed by a modified version of the scheduler of the Darjeeling VM, which executed the task for a given number of bytecodes (time quantum) before granting the CPU to another task. A special component, called network dispatcher, creates private communication channels for each thread of each task on top of a shared network interface, thus granting complete independence to the applications, which could even implement their own routing protocols.

Performance and scalability have been tested by running synthetic benchmarks on VirtualSense motes instrumented in order to measure the turnaround time and the time to completion of each task, while also monitoring the size of the VM heap. Both the time quantum and the number of concurrent tasks have been used as sweep parameters during the experiments to provide to the reader the elements required to evaluate the applicability of the proposed approach.

In summary, the experimental results show that the preemptive multitasking support provided by VirtualSense has a limited impact on performance, which scales smoothly and can be tuned by setting the time quantum used for preemption. Moreover, a `need_resched` flag can be associated with reactive tasks in order to schedule them as soon as their triggering events occur, without waiting for their RR turn. This makes the time to completion of time-critical applications almost independent on the number of concurrent tasks.

Current work is focused on the implementation of a message-passing mechanism providing intertask communication among concurrent applications running on the same mote.

Acknowledgments

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

Conflict of Interests

The authors declare no conflict of interests.

References

[1] C.-C. Han, R. Kumar, R. Shea, and M. Srivastava, "Sensor network software update management: a survey," *International Journal of Network Management*, vol. 15, no. 4, pp. 283–294, 2005.

[2] W. F. Fung, D. Sun, and J. Gehrke, "COUGAR: the network is the database," in *Proceedings of the ACM SIGMOD International Conference on Managment of Data*, p. 621, Association for Computing Machinery Press, 2002.

[3] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: a tiny aggregation service for ad-hoc sensor networks," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation, ACM SIGOPS Operating Systems Review (OSDI '02)*, vol. 36, no. 1, pp. 131–146, 2002.

[4] A. Boulis, C.-C. Han, and M. B. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," in *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys '03)*, pp. 187–200, Association for Computing Machinery, New York, NY, USA, 2003.

[5] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. Lebrun, "Supporting concurrent applications in wireless sensor networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, pp. 139–152, New York, NY, USA, 2006.

[6] C.-L. Fok, G.-C. Roman, and C. Lu, "Agilla: a mobile agent middleware for self-adaptive wireless sensor networks," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 3, article 16, 2009.

[7] C.-L. Fok, G.-C. Roman, and C. Lu, "Servilla: a flexible service provisioning middleware for heterogeneous sensor networks," *Science of Computer Programming*, vol. 77, no. 6, pp. 663–684, 2012.

[8] E. Lattanzi and A. Bogliolo, "Virtualsense: a java-based open platform for ultra-low-power wireless sensor nodes," *International Journal of Distributed Sensor Networks*, vol. 2012, Article ID 154737, 16 pages, 2012.

[9] R. Sugihara and R. K. Gupta, "Programming models for sensor networks: a survey," *ACM Transactions on Sensor Networks*, vol. 4, no. 2, article 8, 2008.

[10] P. Levis, S. Madden, J. Polastre et al., "Tinyos: an operating system for sensor networks," in *Ambient Intelligence*, pp. 115–148, Springer, 2004.

[11] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki—a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN '04)*, pp. 455–462, IEEE Computer Society, 2004.

[12] R. Chu, L. Gu, Y. Liu, M. Li, and X. Lu, "Sensmart: adaptive stack management for multitasking sensor networks," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 137–150, 2013.

[13] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a feature-rich VM for the resource poor," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*, pp. 169–182, Association for Computing Machinery, New York, NY, USA, 2009.

[14] R. Rivest, "The md5 message-digest algorithm," United States, 1992.