Configuration and tuning of sensor network applications through virtual sensors *

Paolo Corsini, Paolo Masci, Alessio Vecchio
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
Via Diotisalvi 2, 56100 Pisa, Italy
firstname.lastname@ing.unipi.it

Abstract

Writing software for sensor networks requires full understanding of the physical phenomenon under observation. Nevertheless, in many cases knowing the details of the area of operation is difficult or unfeasible. This leads to the necessity of configuring and tuning the application executed on the sensor nodes even after the network has been deployed. We present a solution where a layer of computation is inserted between the application and the sensing equipment. This layer acts like a tiny interpreter and can be used to customize the behavior of already running applications. Experimental validation of the architecture and examples of use are also shown.

1. Introduction

A sensor network is a wireless network composed of a large number of communicating nodes used to monitor a physical phenomenon. Each node consists of an embedded microcontroller with a small amount of memory, a battery, a wireless transceiver, and may be equipped with various sensing hardware (light, temperature, etc.). The network is self-organizing and multi-hop communication is used to transport the collected data to a monitoring base station.

In this paper we explore the issues regarding configurability and tuning of the application executed on sensing nodes possibly after they have already been deployed. We propose a software component, called *virtual sensor*, that can be used to tune up already running applications by means of small programs sent from the base station. The virtual sensor is presented within the context of a TinyOS-based architecture. Hence, we first recall some concepts of TinyOS, then we present *VirtuS*, a prototype implementation of the virtual sensor for TinyOS.

*This work is partially supported by Fondazione Cassa di Risparmio di Pisa, Italy (SensorNet Project).

2. TinyOS and nesC

TinyOS [9] is an open-source operating system designed for wireless sensor networks. Applications that run on the TinyOS platform, as well as TinyOS itself, are written with nesC [1], a component-oriented extension of the C programming language. With nesC, programmers can define new components using a C-like syntax, and connect together existing components to create other components or applications (the act of connecting together components is called "wiring"). Each component declares input and output functions, called *commands* and *events*, that are used in the wiring process. Commands and events are usually grouped into *interfaces*, i.e. labeled sets with a given type.

Our system is heavily based on sensor-drivers that in TinyOS are represented by components declaring two interfaces: ADC, that provides analog to digital converter functions, and SplitControl, that provides initialization functions. The ADC interface is composed by a command and an event [1]: the *getData()* command is called by other components to initiate a reading, and the *dataReady()* event is signaled by the sensor-driver when the acquisition is completed. The SplitControl interface is composed by three commands (*init()*, *start()*, *stop()*) that can be used by other components to initialize, turn on/off the sensor component, and three events (*initDone()*, *startDone()*, *stopDone()*), signaling the completion of the commands.

3. The virtual sensor concept

Configuration and tuning of already deployed applications can be a necessity because of several factors: the a-priori knowledge of the area of operation is incomplete, recalibration of sensing equipment, changing conditions, etc. Reprogramming the network by hands is usually unfeasible,

[1] The ADC interface includes another command, *getContinuousData()*, that can be used by applications to initiate a series of ADC conversions. For the sake of clarity, this possibility is not included in the description of the virtual sensor implementation.

Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)
0-7695-2520-2/06 $20.00 © 2006 IEEE

due to the possibly large number of nodes. But also sending the binaries of the application through the network is not a good choice because of the high energy drain of network reprogramming.

An alternative solution can be obtained by using a computation layer that modifies the application behavior. This layer, that we call *virtual sensor*, is programmable at runtime and is able of executing different adaptation schemes that may change during the application lifetime. The rationale under the virtual sensor is that execution flows of application for sensor nodes are mostly guided by values gathered from the environment. As a consequence, a way to tune up an application can be obtained by modifying its "perception" of the environment. Basically, the virtual sensor behaves like a tiny interpreter and transparently changes the application behavior by pre-processing data gathered by the sensors and mapping these values into new values.

The virtual sensor is inserted between the user-defined application components and the sensor driver. The virtual sensor provides the same interfaces of standard sensor-drivers, thus it can be incorporated into existing applications with minimal changes to their source code. This enables to use the virtual sensor also under existing complex applications such as TinyDB [8] (for example to provide homogeneous readings when nodes in the same network are equipped with different sensing boards).

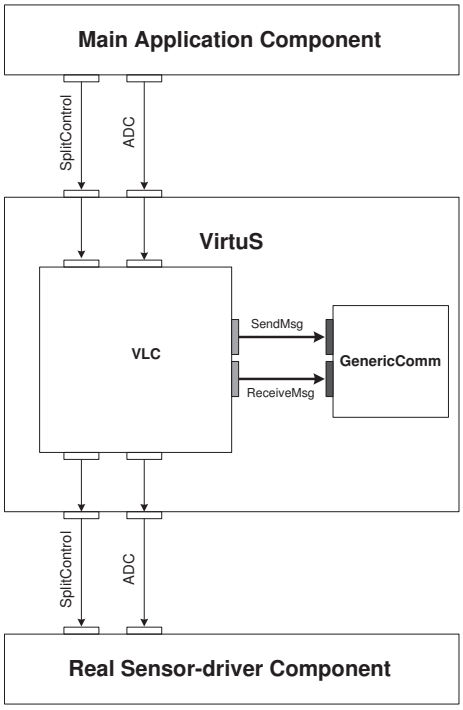


Figure 1. The architecture of VirtuS (some details are not shown).

3.1. VirtuS

VirtuS is an implementation for TinyOS of the virtual sensor. It provides the following features:

- the available instruction set is easily configurable at compile-time and enables developers to generate compact program codes;
- the executed program can be dynamically changed from the base station through mechanisms that disseminate the code into the network;
- the size of the virtual sensor component is in the order of few KBytes.

VirtuS, as depicted in Figure 1, is implemented by a software component that provides the ADC and SplitControl interfaces, the same interfaces exported by standard sensor-drivers. Every time the application invokes the *getData()* command, VirtuS executes a program. On program completion, the final value is returned to the application through the *dataReady()* event. In turn, VirtuS is connected to the SplitControl and ADC interfaces of the real sensor-driver component, whose job is to gather real values.

The computation layer of VirtuS is implemented within a component called *Virtual Layer of Computation* (VLC). The VLC is connected to GenericComm, a component of the TinyOS library, which provides networking functionalities in order to i) receive the packets containing the programs, ii) forward the programs to those nodes that are not in the communication range of the base station, iii) communicate with the VLC of neighbor nodes (e.g., to request the sampled values or to reply to a request).

Programs for VirtuS are usually few bytes long, thus they can be transmitted within a single packet. For this reason VirtuS embeds a component that provides simple mechanisms for disseminating programs into the network. Basically, when VirtuS has to be reprogrammed, the packet that contains the program is injected from the base station. Then, the packet is retransmitted by each node and the network is flooded. This enables the reprogramming of all nodes. A selective reprogramming of a single node is also possible. In fact, the packet that contains the program includes a field that can be used to specify the address of a specific node.

4. VirtuS programming

VirtuS is implemented as a stack-based machine. Some of the instructions include an immediate operand, while others operate only on the values on the top of the stack. The instructions can be grouped into 4 categories:

Registers and Stack The use of registers and stack makes easy to store/retrieve temporary values and fetch operands. The stack can be used also as a parameter passing buffer across different functions. Examples of these instructions are `LOAD r`, that copies the value of register `r` onto the stack, and `POP`, that removes the topmost value from the stack.

Arithmetic Besides basic arithmetic operations, such as sum and subtraction, VirtuS includes functions to compute frequent high-level operations, for e.g. the average value `AVG`, the minimum `MIN` and the maximum `MAX` value.

Branching The use of branching instructions enables the execution of different instruction sequences depending on runtime conditions. An example of a branching instruction is `GOTO` *l*, that jumps to instruction number *l*.

Data acquisition These instructions are specialized for getting data from sensors, for example `ADCGET` gets the ADC reading of the real sensor. Another example of instruction for data acquisition is `ADCREQ`, that requests the ADC reading from the neighbor nodes.

As an example, a simple program is shown here. The program performs temporal aggregation by sampling the ADC every second, and then averaging four samples:

```
0:PUSH 10    (push the sampling period, expressed in 100's of millis,
              onto the stack)
1:PUSH 4     (push the number of requested samples onto the stack)
2:ADCGET     (pop two operands from the stack, periodically get
              the ADC samples as specified by the operands,
              and place the sampled values onto the stack)
3:AVG        (compute the average of the values on the stack)
4:RETURN     (program end)
```

5. Examples of use

The virtual sensor can be used in different scenarios. In the following sections we show some examples of use.

5.1. Temporal aggregation

The virtual sensor architecture can be used to filter spikes of sensed data before returning the value to the main application. This simple idea can be useful to obtain more meaningful measures in several scenarios, especially when the physical attribute under observation is characterized by large fluctuations. For example, let us suppose that nodes have been deployed inside a forest, where they are used to measure the intensity of light. It may happen that sensing hardware is exposed to direct sunlight or to shadow for short intervals, due to the movement of trees caused by the wind.

5.2. Spatial aggregation

A key point for an efficient in-network processing is removing the noise. Noise may have different causes: hardware malfunctions (e.g. sensor failures), errors in software specifications (e.g. unsatisfactory sampling method), local conditions of the physical environment. In many cases, a single node is not able to detect if it is properly sensing the environment by simply inspecting its data. Techniques like temporal correlations are not applicable to events with a high variance without stressing the hardware of the nodes (e.g. high sampling rates) or without complex algorithms.

Detecting and removing the occurrence of false-positives is a key factor for the deployment of an efficient sensor network: less data would be transmitted to the base station and the transmitted data would be affected by lower noise. The virtual sensor can be used to link together values of neighbor nodes, thus actually summarizing the knowledge of a group of sensors dislocated in the same area.

5.3. Fine-tuning event detection

A key issue when writing an application for sensor nodes is the understanding of the physical phenomenon that the sensors are going to observe. However, it is sometimes difficult to setup the right threshold for event detection, since there can be little knowledge of the phenomenon, or since the environment conditions change as time goes by. For example, after the network has been deployed, it is possible that motes need to be re-calibrated, because of deterioration of sensing hardware or other causes (e.g. dust).

A trivial solution consists in re-programming all nodes by sending the whole code of the application through the network. The drawback is that network programming wastes a lot of energy. More efficiently, VirtuS can be used to change the value returned to the application, e.g. by adding/subtracting a quantity to the acquired value in order to correct the effects of deterioration. In this case, there is no need to transfer the whole binary code of the application, but it is only required to send a small instruction sequence (few bytes) to those nodes that are not working properly.

5.4 Emulation

The first phase of the development of an application for sensor network is usually carried out on a PC through the

use of a simulator, which simplifies the burden of compiling, executing and correcting software modules. For example, the TOSSim simulator provides an environment for the execution of TinyOS application and allows the programmer to specify the values that must be gathered by nodes, in order to explore all the functionalities of the software under test. Then, the development continues on real hardware, where the application is exposed to operating conditions that are similar to those of its final deployment.

However, this second phase is much more troublesome and time-consuming: for example, let us suppose that a node should react in some way when the sensed value of light becomes greater than a given threshold. To test if the application is correct, the developer has to physically access the specific node and expose it to a source of light.

VirtuS enables developers to test an application on real hardware without actually deploying the nodes. In fact, the application under test can be exposed to programmer-defined conditions by generating an emulated environment.

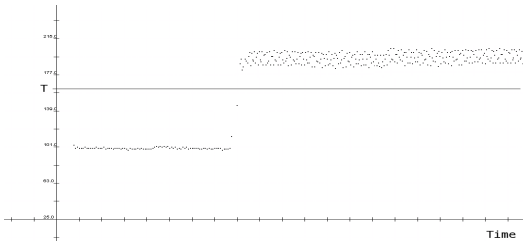
6. Experimental validation

We experimentally validated the virtual sensor using VirtuS in a simple, but realistic, application. In our scenario a sensor network is used to monitor the level of light inside a building. The acquired information is used to determine strategies of energy management (e.g., if some lights are turned on even after the closing time, then they must be switched off, etc.). A light is considered as turned on if the level of light is greater than a given threshold, otherwise the light is considered as turned off.

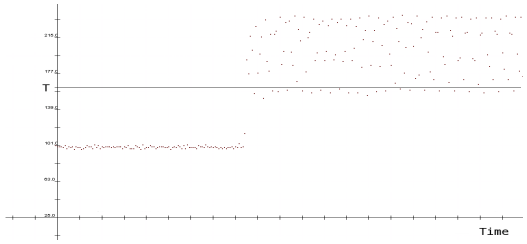
We used Tmote [5] nodes, based on the Telos-B architecture, to determine the level of light. Figure 2(a) shows the value of the samples when a light is turned off (on the left-hand side of the graph) and when it is turned on (on the right hand side). Based on this knowledge, we developed the application and set the value of the threshold to T, as shown in Figure 2(a) (T is a little bit higher than the mean value, to compensate for eventual light coming from the outside). Then the sensor network is installed inside the building.

After deployment, the application is not working properly: sometimes, even if the light is turned on, the value acquired by the nodes is lower than T, thus the nodes are reporting wrong information to the base station.

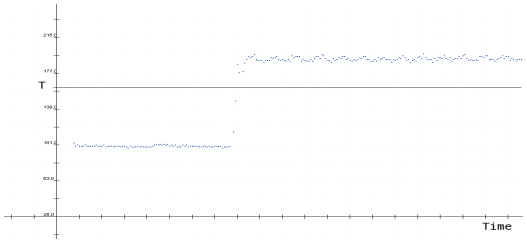
After some investigation, it is noticed that the reason of the malfunction is due to the fact that some incandescence lamps were replaced with fluorescent lamps. Figure 2(b) shows the sampled values when the light is turned on/off if a fluorescent lamp is used. Because of the higher flickering, the sensor readings are characterized by larger variability with respect to the values of Figure 2(a). Therefore in some cases, even if the light is turned on, the acquired value is lower than T.



(a) Incandescence lamps.



(b) Fluorescent lamps (before using VirtuS).



(c) Florescent lamps (when Virtus is used).

Figure 2. Measures of the level of light.

Instead of modifying the application and reprogramming all nodes, we used VirtuS to adapt the application to the new environment. Figure 2(c) shows the sensor readings registered by the application when the virtual sensor is used. In detail, the virtual sensor is programmed to compute the mean value of four data readings before returning the value to the main application (the instruction sequence of the program is shown at the end of Section 4)

The experiment was carried out by using the OscilloscopeRF application, available with the TinyOS distribution. The OscilloscopeRF application consists of two files: OscilloscopeM.nc and Oscilloscope.nc. OscilloscopeM.nc, more than 100 lines of code, contains the implementation of the application logic. Oscilloscope.nc, about 20 lines of code, is the main configuration file and it wires the application logic to the other components (i.e. system, sensor-driver and networking components). To add VirtuS to the application we had to modify only Oscilloscope.nc. In particular, we had to modify just few lines, as shown in Figure 3. This shows that the virtual sensor architecture can be

```
configuration Oscilloscope{}
implementation{
    components Main, OscilloscopeM,
               ... , LightSensorC;

    Main.StdControl -> OscilloscopeP;
    ...
    OscilloscopeM.SensorControl -> LightSensorC;
    OscilloscopeM.SensorADC     -> LightSensorC;
}
```

(a) Original

```
configuration Oscilloscope{}
implementation{
    components Main, OscilloscopeM,
               ... , LightSensorC, VirtuS;

    Main.StdControl -> OscilloscopeP;
    ...
    OscilloscopeM.SensorControl -> VirtuS;
    OscilloscopeM.SensorADC     -> VirtuS;
    VirtuS.SensorControl -> LightSensorC;
    VirtuS.SensorADC     -> LightSensorC;
}
```

(b) Modified

Figure 3. OscilloscopeRF.nc (some details are not shown).

used with minimal changes also with existing applications. Even if the prototype implementation of VirtuS is not yet optimized, the increase of code size of the OscilloscopeRF application was small (less than 5 KBytes) and the RAM required to run the application was almost unchanged (the compiled code required less than 256 more bytes of RAM).

7. Related work

XNP [4] and Deluge [3] are two systems that enable network reprogramming of sensor nodes based on TinyOS. Their mechanisms ensure maximum flexibility, but they are also quite expensive in terms of energy requirements because of the large size of program images. Also, they require to stop and restart the application (it is not possible to tune an application while it is running).

Maté [6] [7] is a tiny virtual machine designed for sensor networks. Using virtual machines to execute the whole sensor application ensures high flexibility. However, adopting this solution with existing applications would be impractical since it would require re-writing the whole application from scratch using the language of the specific virtual machine.

SOS [2] is a new operating system for sensor nodes. SOS consists of a common kernel and dynamic application modules. Application modules can be substituted during runtime by injecting the new version of the modules into the network. This solution presents high flexibility, but it requires to write the entire application according to the interfaces offered by the kernel of the operating system. As a consequence, tuning existing applications for sensor nodes, mostly written for the nesC/TinyOS platform, is not obvious.

8 Conclusions and future work

We presented a software component called virtual sensor. The component configures and tunes existing applications by placing an abstraction layer between the application logic and the sensor-driver. The virtual sensor acts like a tiny interpreter and it can be remotely programmed. By pre-processing the sensed data, the behavior of the whole application is silently modified without actually modifying the source code of the application logic. A TinyOS-compatible prototype implementation of the virtual sensor was implemented and an experimental test proved its effectiveness in a real application.

References

[1] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: a holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, 2003.

[2] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *Proceedings of MobiSYS 2005*, 2005.

[3] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[4] C. T. Inc. http://www.xbow.com.

[5] M. Inc. http://www.moteiv.com.

[6] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM Press.

[7] P. Levis, D. Gay, and D. Culler. Active Sensor Networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.

[8] S. R. Madden, M. J. Franklin, and J. M. Hellerstein. TinyDB: An Acquisitional Query Processing System for Sensor Netowrks. *ACM Transactions on Database Systems*, 30(1):122–173, March 2005.

[9] TinyOS. http://webs.cs.berkeley.edu/tos/.