Experiences with the TinyOS Communication Library ⋆

Paolo Corsini, Paolo Masci and Alessio Vecchio

Dipartimento di Ingegneria della Informazione
Università di Pisa
56122 Pisa, Italy

Abstract. TinyOS is a useful resource for developers of sensor networks. The operating system includes ready-made software components that enable rapid generation of complex software architectures. In this paper we describe the lessons gained from programming with the TinyOS communication library. In particular, we try to rationalize existing functionalities, and we present our solutions in the form of a communication library, called TComm-Lib.

1 Introduction

A sensor network is a wireless network of communicating nodes. Each node consists of an embedded micro-controller 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 data collected by nodes to a monitoring base station.

Hardware resources of the nodes are extremely limited because of a set of constraints: *i)* the cost of the nodes must be kept as low as possible since the number of nodes can be in the order of hundreds or even thousands elements, *ii)* they must be energy efficient since the replacement of batteries is often unfeasible or expensive, *iii)* their size must be kept small in order to be ubiquitous. For instance, nodes based on the Telos-B platform are equipped with 48KBytes of instruction memory, 10KBytes of RAM and the current draw in active and sleep mode is respectively $1.8mA$ and $5.1\mu A$.

Programming in such environment is not an easy task, not only because of the hardware limitations, but also because developers of applications for sensor networks belong to different technical areas, from telecommunications to electronics and computer science. Fortunately, they do not have to write their own applications with assembly-like languages, but can leverage on high-level languages and libraries providing basic services. The standard platform for sensor networks is TinyOS [1], available for a number of different hardware architectures, and nesC [2] is the used programming language.

In this paper, we report our experience on developing applications for sensor networks, focusing on the communication services provided by TinyOS. Besides highlighting the positive and negative features of the library, we describe the issues encountered, and provide the adopted solutions in the form of a communication library, *TComm-Lib*.

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

Corsini P., Masci P. and Vecchio A. (2006).
Experiences with the TinyOS Communication Library.
In *Proceedings of the 5th International Workshop on Wireless Information Systems*, pages 47-55

TComm-Lib is not aimed at introducing new network services, instead it tries to rationalize existing functionalities in order to give a simple user experience for effective programming with sensor networks. TComm-Lib has been built, almost completely, by simplifying, re-organizing, and extending the original TinyOS library.

2 TinyOS and nesC

TinyOS is an open-source operating system designed for wireless sensor networks. Architecture and implementation of TinyOS applications are component-based, enabling rapid innovation and modularity. The libraries shipped with the operating system include a number of ready-made components that can be connected together with user-defined components.

Applications that run on the TinyOS platform, as well as TinyOS itself, are written with nesC [2], a component-oriented extension of the C programming language. With nesC, programmers can define new components using a C-like syntax, and connect them together in order to create new components or applications (the act of connecting 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.

A component can offer multiple instances of the same interface. Each interface can be connected to different components, and a specific interface is selected through the use of an index. In this case the interface is said to be *parametric*.

2.1 The TinyOS Communication Library

The TinyOS communication library supports single and multi hop communication with other nodes (via the wireless transceiver), and serial communication between a sensor node and the base station (via USB or serial port).

Since our paper focuses on the communication library of TinyOS, in the following we describe `GenericComm`, the component used for communication.

`GenericComm` implements single hop and serial line communication (the last feature is used only by the node connected to the base station). Moreover, it provides *i)* a control interface (`StdControl`) to initialize, start and stop the component, *ii)* a parametric interface for sending packets (`SendMsg[uint8_t id]`[1]), *iii)* a parametric interface for receiving packets (`ReceiveMsg[uint8_t id]`). The implementation of `GenericComm` is specified in the following file (some details are not shown):

```
configuration GenericComm{
    provides{
        interface StdControl;
        interface SendMsg[uint8_t id];
        interface ReceiveMsg[uint8_t id];
    }
```

[1] This notation means that 256 instances of the SendMsg interface are available (*uint8_t* is an unsigned int type coded on 8 bits).

```
}
implementation{...}
```

Let us now examine the interfaces for sending and receiving packets of `GenericComm`. The `SendMsg` interface declares one command and one event:

```
interface SendMsg{
    command result_t send(uint16_t address,
                          uint8_t  length,
                          TOS_Msg* msg_ptr);
    event   result_t sendDone(TOS_Msg* msg_ptr,
                              result_t success);
}
```

The `send()` command has three parameters: the destination address, the length (in bytes) of the payload, and a pointer to the buffer containing the message to be sent. The command is non-blocking. The `sendDone()` event is signaled when the message is actually transmitted. The event has two parameters: a pointer to the buffer containing the last message sent, and a flag reporting the success or failure of the sending attempt.

Communication over the serial connection is achieved by using a special and reserved address (codified as `UART_ADDR`).

The `ReceiveMsg` interface declares only one event:

```
interface ReceiveMsg{
    event TOS_Msg* receive(TOS_Msg* msg_ptr);
}
```

The `receive()` event is signaled every time a message is successfully received over the radio or over the serial port. The event has one parameter, that is a pointer to the buffer containing the received message. The event must return to the caller the buffer for the next receive operation (a buffer-swap mechanism is used to avoid the overhead of copying data).

3 Developing Network Applications

3.1 Features and Limits

In this section, we report our experience on using the TinyOS communication library and explore how unexpected behavior may arise for a number of factors.

Selective Powering of Communication Hardware. Hardware devices are represented by software components. By calling the `start()` command of a software component, the corresponding hardware device is powered on. Similarly, by calling the `stop()` command a hardware device can be powered off. In some cases, a single software component represents several hardware devices. For example, `GenericComm` is associated with both the radio chip and the controller of the serial interface. Nevertheless, if no dedicated control interface is exported, then selective powering on/off is unfeasible. In

fact, the `start()` command turns on all the hardware associated with a software component, even if only one of the devices is actually needed. Thus, when the `start()` command of `GenericComm` is called, for instance because the application is going to communicate over the wireless channel, the radio is powered-up, but at the same time the serial controller is automatically powered-up too.

Management of the Duty Cycle. In many cases, the application running on the nodes of a sensor network is cyclical: the environment is sensed, acquired information is transmitted, then the application sleeps for a given amount of time before restarting the cycle. Usually, the sleep time is longer than the active time by several orders of magnitude, therefore it is of primary importance to reduce the amount of energy consumed during the sleep time. Power management can be done at the application level by using the *start()* and *stop()* commands. Nevertheless, hardware devices may present a slow power-up phase with respect to the clock of the microcontroller (e.g. because the voltage regulator of the device needs a certain time to stabilize), and trying to use a component during the transition phase may cause unpredictable behavior. In other words, applications need a view on the power status of hardware devices, but the *StdControl* interface does not suit properly to this purpose (the `start()` and `stop()` commands return immediately and do not wait for a complete power-on, or power-off, of the device). In the context of the communication library, this problem arises because the radio chip device has a slow power-up phase, but `GenericComm` provides no interface to expose the transition process.

Controlling the Transmission Power. The radio installed on the sensor nodes has a transmission range that is approximately one hundred meters, if used with maximum power. However, in some scenarios, it can be useful to manually reduce the transmission power: for example, if the network is particularly dense, by reducing the transmission range it is possible both to save energy and decrease the number of packet collisions. Also, during the debugging phase of a network protocol or an application, the reduction of the transmission range can help to set up a multi hop testbed. The TinyOS library includes software components for radio range setup, but they are usually hidden inside complex configurations that do not expose an interface to access such functionality.

Power Saving Modes. TinyOS automatically switches the microcontroller to a low-power mode whenever possible, thus extending the life-time of nodes even when the application does not explicitly take care of it. For example, the TMote sky nodes [3] automatically go into sleep mode when all the following conditions are satisfied: *i)* the radio is turned off, *ii)* all high speed clock outputs are disabled, *iii)* the serial peripheral bus is idle, *iv)* the task queue is empty. Nevertheless, automatic transition to the sleep mode is disabled by default on almost all platforms. Hence, the programmer has to explicitly include the activation of the power saving subsystem into the application code.

3.2 Related Work and Motivations

The TinyOS community is making a big effort to improve and expand the APIs of the whole TinyOS library. In particular, recent works are proposing solutions to promote a standard framework across different platforms. In [4] the authors discuss design patterns useful to solve common problems, and aimed to the development of efficient and robust program structures. In [5] an abstraction layer is presented in order to standardize the interfaces of nesC components across different platforms. Both these works reflect the philosophy of the TinyOS Enhancement Proposals (TEPs), basis of TinyOS 2.0. Notice that sometimes radical changes are required in order to actuate such proposals. In fact, applications written for TinyOS 2.0 are not backward-compatible with the previous platform.

Further research work related to the programming paradigms provided by TinyOS is described in [6], where the authors summarize and analyze the experience of the TinyOS community in creating software abstractions for the communication layer.

The motivations of our work rely on finding solutions that balance between new programming trends of TinyOS 2.0 and the semantic structure of old-style applications for sensor nodes. In particular, the revisions we propose are backwards compatible, and enable developers to improve existing programs without actually re-engineering the entire application.

4 TComm-Lib Solutions

Most of the ideas behind TComm-Lib are the result of our previous research experience in the context of sensor network. In particular, during the implementation of a monitoring application, we encountered all the issues previously described.

In the following subsections we show how simple revisions of the TinyOS communication library may help application developers to gain control over specific resources and produce modular architecture for effective code.

4.1 Selective Powering of Communication Hardware

As previously introduced, `GenericComm` implements communication over the serial line and over wireless channels. We split `GenericComm` into two components, on the basis of the offered function: `RfmComm` dedicated to wireless transmissions, and `UsbComm` used for communication over the serial line.

`RfmComm` preserves the core interfaces of `GenericComm`, i.e. `SendMsg` and `ReceiveMsg`. Thus, the component of the TinyOS library can be replaced with the new component with minor changes to the application code. Besides the interfaces for sending and receiving packets, `RfmComm` includes also `SplitControl`, which replaces `StdControl` and is used to safely turn on and off the component, and `RFPower`, which adjusts the transmission power. These interfaces are described in Section 4.2 and 4.3.

```
configuration RfmComm{
    provides{
```

```
        interface SplitControl;
        interface SendMsg[uint8_t id];
        interface ReceiveMsg[uint8_t id];
        interface RFPower;
    }
}
implementation{....}
```

UsbComm provides the same interfaces of RfmComm, except for RFPower:

```
configuration UsbComm{
    provides{
        interface SplitControl;
        interface SendMsg[uint8_t id];
        interface ReceiveMsg[uint8_t id];
    }
}
implementation{....}
```

RfmComm and UsbComm offer a clear vision of the hardware devices actually involved and they can be used by developers depending on the needed functionality. Moreover, as a nice side-effect, there is no need to reserve the UART_ADDR address for the serial line.

4.2 Management of the Duty Cycle

Events signaling the progress of the power-up phase can be found within a specific TinyOS control interface, SplitControl. The SplitControl interface is provided with the start() and stop() commands of the StdControl interface, but it also offers the events startDone() and stopDone(), used to signal the completion of the corresponding command:

```
interface SplitControl{
    command result_t init();
    command result_t start();
    command result_t stop();
    event result_t initDone();
    event result_t startDone();
    event result_t stopDone();
}
```

Within TComm-Lib, both RfmComm and UsbComm support the SplitControl interface. As a consequence, the components can be appropriately turned on and off, even if they are associated with hardware devices with a slow power-up phase.

4.3 Controlling the Transmission Power

The power used for wireless transmission can be controlled through interfaces that change on the basis of the used radio chip. For example, the *mica* and *mica2* platforms

use the *CC1000* radio, thus the `CC1000Control` interface must be used, while the *micaZ* and *Telos-B* platforms are equipped with the ZigBee-compliant *CC2420* radio chip, and the `CC2420Control` interface must be used.

Different control interfaces show common commands but generally they are not compatible, since different radio chips have different available functions. Moreover, these interfaces include a number of commands that are actually rarely used by other components.

We decided to simplify the interface and to expose only the command to control the transmission range. The new interface is called `RFPower` and it has only one command, `SetRFPower(uint8_t power)`, that is platform-independent. The `power` parameter specifies the radio range (0 is the shortest, 31 is the longest).

```
interface RFPower{
    command result_t SetRFPower(uint8_t power);
}
```

Within TComm-Lib, the `RFPower` interface is provided by the `RfmComm` component.

4.4 Power Saving Modes

The automatic sleep mode of the microcontroller of the sensor nodes can be activated by calling the `enable()` command of a low-level software component (`HPLPowerManagementM`).

We transparently changed the default setting of the automatic power-saving subsystem by extending the `Main` component of the TinyOS library: when the `init()` command is called to initialize the application, the power management module is activated. The new component, called `LpMain`, can be transparently incorporated within existing applications since the interfaces are unchanged.

5 The Importance of Power Management

Energy is the most valuable resource of sensor nodes. Several research studies propose architectures to optimize energy consumption, where effective solutions are strictly connected to specific application scenarios. However, besides the benefits achievable through architectural solutions, experience highlights the importance of low-level implementation and precise access to hardware resources.

In fact, controlling the power status of hardware devices is a simple, yet effective, energy saving solution that can be adopted by any software layer. Besides duty cycle optimizations included inside low-level software components, the whole application for sensor nodes may also present high level components that can be toggled on and off.

Let us imagine a scenario where a sensor network is used to monitor the level of light inside a building. The application periodically samples the light and sends a message to the sink node. Each message contains the ID of the sender node and the corresponding sampled value. Let us also assume that single hop communication is used for wireless transmission. This application is characterized by an active period where the application

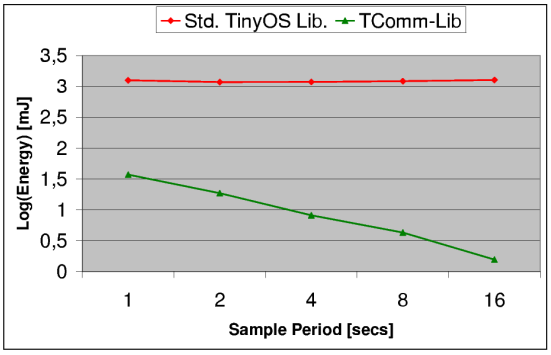

Fig. 1. Energy consumption varying the period of operation.

performs data acquisition and communication and an inactive period where the node is idle. This periodical model of operation is representative of a wide class of applications for sensor networks.

During the inactive period the node must enter the low power state, in order to save as much energy as possible. The amount of energy that can be saved by entering the low power state can be relevant, especially when the inactive period becomes longer.

Figure 1 shows the difference, in terms of energy consumption, of two versions of the application[2]: the first one uses TComm-Lib to enter the low power state as soon as possible (i.e., immediately after the transmission of a packet), the second one does not take care of driving the components used by the application into the low power state. As expected, when the period of operation of the application becomes longer, the difference becomes even more relevant.

From the programmer's perspective, power management can be easily done by using the TComm-Lib revised components and interfaces: the `RfmComm` component can be safely switched on/off as required, since it provides a feedback of its state to the application, and the `UsbComm` component is left turned off and will never be activated since not needed. Power management is more troublesome if using the standard TinyOS communication library. For example, the USB subsystem cannot be switched off, as it is hidden within `GenericComm`. Also, safely switching the radio on and off requires an additional programming effort to avoid using the radio equipment when it is still in an inconsistent state.

For testing purposes, we included these mechanisms also within the application implemented with the standard TinyOS communication library. We observed that within the simulation environment everything worked fine and obtained the same power consumption of TComm-Lib. Nevertheless, after installing this modified version on real

[2] We used PowerTOSSIM [7] to estimate the energy consumption.

nodes, we experienced that nodes were not able to transmit any message at all, and sometimes they crashed. We attributed these malfunctions to the incomplete power-up phase of the radio (these problems are not visible in the simulated environment).

This problem can be avoided, for example, by introducing a delay between the activation of the radio subsystem and its usage. However, this requires the programmer to know the length of the transition phase of the radio component from the low power state to the operational state, that can be different on the basis of the hardware platform.

6 Conclusions

Creating software abstractions suitable for sensor networks is challenging since the sensor nodes require software architectures radically different from traditional networking systems.

As known, the application logic is central for defining an effective strategy for energy saving. To make these strategies real, applications need the cooperation of the underlying layers, which must expose mechanisms for power management. As described in [8], the strategies adopted by applications are in many cases based on few and recurring principles, for this reason the underlying layers can offer abstractions that appear to be general and reusable.

In this paper we have described our experience on developing applications for sensor networks, focusing on the abstractions provided by the TinyOS communication library. On the basis of the lessons learned we rationalized and re-organized the original communication library, not only to provide cleaner programming abstractions, but also to generate more energy efficient code.

References

1. TinyOS: (http://webs.cs.berkeley.edu/tos/)
2. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: a holistic approach to networked embedded systems. SIGPLAN Not. **38** (2003) 1–11
3. Moteiv Inc.: (http://www.moteiv.com)
4. Gay, D., Levis, P., Culler, D.: Software Design Patterns for TinyOS. Proceedings of the ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05), Chicago (2005)
5. Handziski, V., Polastre, J., Hauer, J., Sharp, C., Wolisz, A., Culler, D.: Flexible Hardware Abstraction for Wireless Sensor Networks. Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN '05), (2005)
6. Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, R., Woo, A., Brewer, E., Culler, D.: The Emergence of Networking Abstractions and Techniques in TinyOS. Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI2004) (2004)
7. Shnayder, V., Hempstead, M., Chen, B., Welsh, M.: PowerTOSSIM: efficient power simulation for tinyos applications. Proceedings of ACM SenSys 2003 (2003)
8. Levis, P., Hill, J., Buonadonna, P., Szewczyk, R., Woo, A.: A Network-Centric Approach to Embedded Software for Tiny Devices. Lecture Notes in Computer Science (2001)