A cooperative multitasking model for networked sensors

Silvana Rossetto and Noemi Rodriguez
Departamento de Informática, PUC-Rio, Brazil
silvana,noemi@inf.puc-rio.br

Abstract

This paper proposes a concurrency model which integrates the asynchronous and event-driven nature of networked sensors with a more familiar programming style for the developer. We argue that coroutines can provide a basis for this integration and describe some details of its implementation, which was developed as an extension to the TinyOS operating system.

Keywords *Sensor networks, TinyOS, programming models, event-driven model, cooperative multitasking, coroutines*

Technical Areas *Programming abstractions for sensor networks, operating system and middleware support*

1. Introduction

Recent advances in low-power microcontrollers and wireless communication have enabled multifunctional tiny devices which can be densely deployed throughout a physical space to sense, process and communicate required data. Similarly to other embedded systems, networked sensors must respond to different stimuli, including physical events and messages from other devices. Besides, networked sensors are highly unstructured and dynamic environments, in which there are no well-behaved communication patterns. As a consequence, computing models for such systems are typically event-driven and asynchronous [4, 6].

In their simplest form, event-driven models encompass a set of independent activities or services, each of them with a single execution line that handles each incoming event at a time. This leads to a state-machine programming model, in which the arrival of a new event can trigger a transition that depends on the current state. Although conceptually simple, this is not the classical way in which programmers are used to thinking. When an operation cannot complete immediately, it must be split across one or more invocations of the event handler, forcing the programmer to code "continuations" within his handlers. Between event handler invocations, new events may occur, triggering new logical execution lines. The developer must, thus, construct his program as a state machine and face issues such as maintaining consistent state across multiple logical tasks. This can impose high programming costs.

In this paper, we propose a programming model which combines the asynchronous basis of event-driven systems with a more classical programming interface for the developer. The central idea is to investigate a computing model that takes into account the limited resources of sensor devices while supporting higher-level programming paradigms able to reduce the cost of developing sensing applications. We use cooperative multitasking to support lightweight concurrency without forcing the programmer to manually build continuations within an application.

The traditional approach to dealing with concurrent execution lines is to use multithreading. However, multithreading is typically preemptive, that is, the programmer has no control over the moment at which control switches occur. This imposes context switching overhead and the need to deal with race conditions and deadlocks. One alternative is to use *coroutines* [13, 7], a programming language construct that enables a process to maintain distinct execution lines and explicitly alternate the control among them. Thus, distinct parts of an application can suspend only it is impossible for them to continue immediately and be resumed later, when their processing requirements become available. That is, coroutines are a collaborative and non-preemptive type of multitasking. This aspect is fundamental in making coroutines suitable for networked sensors: context switching occurs only it is necessary (reducing the computing cost associated to it) and most concurrency issues (such as race conditions) can be implicitly avoided.

To experiment with the coroutine abstraction for networked sensors, we have integrated a coroutine-based concurrency model into TinyOS [11], a well-known operating system for networked sensors. Basi-

cally, we have implemented a simple API and a coroutine scheduler that allow us to transform two-phase operations (typically specified by means of the pair command/event in a TinyOS interface), into one operation that is appropriately suspended and resumed, allowing the programmer to maintain a sequential view of his application. In this paper, we discuss the implementation of this model and the gain of using it in developing a sensing application, and evaluate the associated computing cost.

The rest of this paper is organized as follows. Section 2 describes TinyOS. Section 3 presents the proposed concurrency model and discusses how we have integrated this model into TinyOS. Section 4 evaluates the cost of using coroutines in a sensing application. Finally, in Section 5, we include some final remarks.

2. Systems for networked sensors

Networked sensors consist of potentially thousands of tiny, low-power nodes, each of which execute concurrent, reactive programs that must operate with severe memory and power constraints. Information must be simultaneously captured from sensors, manipulated, and streamed onto a network. Moreover, nodes must deal with events that require real-time responses. An example is message arrival. Typically communication is radio-based. The radio is an asynchronous input/output device that contains no buffer, so each bit must be serviced by the node as soon as it becomes available. Systems for networked sensors thus present some special requirements. First, software solutions must make efficient use of processor and memory while enabling low power consumption. Second, it is necessary to maintain a number of concurrent flows and juggle numerous outstanding events.

TinyOS [11] is the current state of the art in operating systems for sensor network research. It implements a component-based model based on split-phase interfaces, asynchronous events and deferred computation. Hardware interruptions trigger immediate execution of event handlers. Tasks are the main unit of execution, and each task executes to completion, except that it may be interrupted by a hardware event (there is no concurrency among tasks). In the next section we describe some important design aspects of TinyOS.

2.1. The TinyOS design

TinyOS design is based on three programming constructs: *commands*, *events*, and *tasks*. Both commands and events are intended to perform small amounts of work. *Commands* are used to request services and complete immediately. Typically, a command handler deposits request parameters and conditionally posts a task for later execution. *Events* are signaled to indicate service (command) completion or hardware events. The lowest-level components have handlers connected directly to hardware interrupts, which can be external interrupts, timer events or counter events. An event handler can deposit information in the component's environment, post tasks, signal higher level events or call commands. *Tasks* are a form of deferred procedure call that allows postponing processing. They are atomic with respect to each other and run to completion, but can be preempted by interrupts (hardware events). Tasks allow concurrency within each component since they execute asynchronously with respect to events. To ensure low task execution latency, individual tasks are expected to be short, i.e, lengthy operations should be spread across multiple tasks. Posted tasks are executed by the TinyOS scheduler when the processor is idle.

TinyOS highlights separation of construction and composition. There are two different programming constructors: *module*, that is used to provide code; and *configuration* that is used to wire components together. The TinyOS component behavior is specified in terms of interfaces that can be provided or used by the component. Interfaces specify a multi-function interaction channel between two components, the *provider* and the *user*. The interface provider must implement a set of named functions called *commands*, and the interface user must implement the set of named functions, called *events*, that can be signaled upon completion of the commands it uses.

A C-like language called *nesC* [10] was specially designed to provide the event-driven concurrency model used by TinyOS. In order to avoid data races – which can occur due to concurrent updates to shared state – nesC offers two options: either to implement all of the critical code inside tasks or to use *atomic sections* to update the shared state. Atomic sections are small code sequences that nesC ensures will run atomically by disabling and enabling interrupts.

3. A new concurrency model for networked sensors

Although it deals well with the main constraints of networked sensors, the programming model designed by TinyOS is not easy to use. Its multitasking engine maintains a two-level scheduling structure that is similar to a finite state machine model, which can be difficult to program. To develop a simple sensing application in nesC, the programmer must typically control shared memory access and partition basic requests into

```
module SurgeM {
  uses {
    interface ADC;
    interface SendMsg;
    interface Timer;
    ... }}
implementation {
  TOS_Msg gMsgBuffer;
  norace uint16_t gSensorData;
  bool gSendBusy;

 task void SendData() {
   SurgeMsg *pReading;
   pReading = (SurgeMsg *)((&gMsgBuffer)->data);
   pReading->reading = gSensorData;
   if ((call SendMsg.send(..., &gMsgBuffer)) != SUCCESS)
     atomic gfSendBusy = FALSE;
 }

 event result_t Timer.fired() {
   call ADC.getData();
 }

 async event result_t ADC.dataReady(uint16_t data) {
   atomic if (!gSendBusy) {
     gSendBusy = TRUE;
     gSensorData = data;
     post SendData();
   }
 }

 event result_t SendMsg.sendDone(...) {
   atomic gSendBusy = FALSE;
 }
}
```

Figure 1. Core logic of the *Surge* application.

two-phase operations.

As an example, Figure 1 shows the main code of *Surge*, a nesC application in which nodes in the network take light readings and forward them to a base station. Each time the `Timer.fired` event is signaled, the `ADC.getData` command is called to get a new sensor value. When the sensor value is available, the hardware signals the `ADC.dataReady` event. As `gSendBusy` is accessed in a hardware event handler, its use must be protected by one *atomic* statement. The handler for this event finishes by posting the `SendData` task, which sends the value to the base station.

In this code, the main task (to take data readings and forward them) was broken into four distinct pieces of code, since sensor value reading is a typical split-phase operation. The ADC interface defines the command `getData` to request a reading; and the event `dataReady` to signal the sensor value is available.

Our goal is to provide a more intuitive programming abstraction to the developer. A classical and more convenient way to get data readings would be to call just one simple command and get the sensor value as the return value to this request. In our proposal, we encapsulate the two phases of a typical request/answer in a single request, enabling the programmer to structure his application as sequential code instead of as a state machine. Figure 2 illustrates how the *Surge* code is simplified with this approach. In the original version, `Timer.fired` issues the `ADC.getData` command, and `ADC.dataReady` handles the event signaled by this command. In the new code, in Figure 2, operation `newSendData` contains both the invocation of `ADC.getData` and the handling of its result, eliminating the split-phase behavior.

We achieve this goal by employing *coroutines*. The possibilities brought by coroutines are in many ways similar to those offered by *multithreading*, but there are important differences. Multithreading [5] is a notion widely explored in order to enable processes to have distinct execution lines called *threads*. The developer defines the distinct threads of execution and the operating system, or an appropriate library, is responsible for relocating the processor control among threads. Coroutines allow the programmer to suspend an execution line to wait for some system event and restore this execution later, but context changes are explicit, avoiding race conditions. On the other hand, the programmer is responsible for control transfers.

In this work, we use coroutines to allow a networked sensor application to respond to system events continuously, modeling each split-phase operation as a new coroutine, and transferring control back to the main loop when it needs to wait for hardware events to continue. Control transference is encapsulated in blocking operations, so they will only occurs when it is necessary and the programmer need not explicitly deal with *transfers*.

We propose changing interfaces like ADC into new interfaces in which only *commands* are offered: blocking operations are transformed into split-phase calls; the context of execution is preserved; and yet the application is able to respond to other system events or deferred tasks while the operation is not able to continue. In order to support this scenario, the following three steps were necessary: (1) building a basic API with coroutine operations for sensors; (2) implementing a coroutine scheduler to resume coroutines automatically; (3) defining a step-by-step procedure to construct proxy interfaces, redefining TinyOS two-phase operations as one-phase operations.

3.1. Coroutine operations for sensors

Our first step was to implement support for the coroutine construct in TinyOS. A coroutine is represented by a code address and has its own stack. There are different syntactic and semantic ways to support coroutines [7]. We support asymmetrical coroutines, al-

```
module newSurgeM {
  uses {
    interface newADC;
    interface newSendMsg;
    interface Timer;
    ... }}
implementation {
 TOS_Msg gMsgBuffer;

 void newSendData() {
   SurgeMsg *pReading;
   result_t result;

   pReading = (SurgeMsg *)((&gMsgBuffer)->data);
   call newADC.getData((&pReading)->reading);
   call newSendMsg.send(..., &gMsgBuffer, &result);
  }

 event result_t Timer.fired() {
   POST(newSendData);
 }
}
```

Figure 2. New core logic of the *Surge* application.

lowing arbitrary functions, independently written, to be invoked as coroutines. For this, we implemented a set of operations that allocate storage for a coroutine stack, associate a function to a coroutine storage area, transfer control to a coroutine, and yield control from a coroutine.

A key issue related to coroutine implementation is how to pass parameters between two coroutines. It is difficult because we typically use the stack to pass parameters and each coroutine uses a different stack. Moreover, a typical coroutine can have several entry points (immediately after each transfer operation). To maintain our implementation simple, we use (scope restricted) global variables to transfer data.

Implementation We implemented coroutines for the microcontroller ATmega128L [1] (from Atmel-AVR family [2]). It has 128 KB of flash for program memory and 4 KB of SRAM for data memory. The coroutines stacks are allocated on the heap.

One simple way to implement coroutines is by using the `setjmp` and `longjmp` functions [9]. The `setjmp` function saves the current execution context (including the stack pointer) into a pre-defined data structure to be passed later as argument to the `longjmp` function. The `longjmp` function, in its turn, resumes the execution context previously saved by `setjmp`.

The AVR Libc [3] library (a subset of ANSI-C library for AVR microcontrollers) implements `setjmp` and `longjmp` functions and defines a specific data structure, called `jmp_buf`, to be used by these functions. To implement the coroutine construct, we do a `setjmp` to save the initial state in the `jmp_buf` structure and then manually adjust the program counter and the stack pointer fields.

3.2. Coroutine scheduler

TinyOS provides a queue for all tasks and a scheduler to execute these tasks using a FIFO policy. We have extended the TinyOS task scheduler to include a coroutine queue (with pre-allocated space) and a coroutine scheduler that resumes coroutines that are ready. The TinyOS scheduler is implemented as a set of C functions in the file "sched.c". Since we are using TinyOS 1.x, we have changed this file directly (a future step is to update this implementation to TinyOS 2.0 by designing an appropriate scheduler component).

A `TOS_post_coro` function (similar to `TOS_post`) was implemented. Basically, this function receives a function as argument and associates it to an available coroutine. The coroutine is then marked as "ready" and can be resumed by the scheduler. We have changed the TinyOS main loop to call a function (similar to `TOSH_run_next_task`) that checks and resumes all ready coroutines.

In order to offer access to coroutine services from the component level, a set of functions are exported by file "sched.c", which includes:

- `bool POST(procedure_t proc)`: schedules a procedure to be executed as a coroutine;
- `uint8_t GETID()`: returns the current coroutine;
- `void SUSPEND()`: transfers control execution back to the main coroutine;
- `void RESTORE(uint8_t id)`: informs the scheduler that the given coroutine is ready to be resumed.

3.3. Proxy for current TinyOS interfaces

Using the new TinyOS scheduler and the set of functions exported by it, we can rewrite typical TinyOS interfaces, encapsulating in their implementation the transfer of control between coroutines. As a result, the programmer that uses such an interface in his application will be provided with the sequential view we wanted. Figure 3 presents, as an example of this, the interface proxy we have implemented to replace the ADC interface.

In the new version of the ADC interface, the `newADC.getData` command calls the original command `ADC.getData` and suspends the current execution line by calling `SUSPEND`. At this point the execution control goes back to the main loop, so the application is

```
interface newADC {
 async command result_t getData(uint16_t *data);
}

module newADCM {
 provides { interface newADC; }
 uses { interface ADC; }
}
implementation {
 uint8_t gBusy, gCoroID;
 uint16_t gData;

 async event result_t ADC.dataReady(uint16_t data) {
  atomic gData = data;
  atomic RESTORE(gCoroID);
  return success;
 }

 async command result_t newADC.getData(uint16_t *data){
  atomic{
   if (gBusy) return fail;
   gBusy = 1;
   gCoroID = GETID();
  }
  call ADC.getData();
  SUSPEND();
  atomic {
   *data = gData;
   gBusy = 0;
  }
  return success;
 }
}
```

Figure 3. Proxy for ADC interface.

able to handle other events or tasks. The coroutine identifier is stored in the `gCoroID` variable to be used later, when this coroutine becomes ready to continue.

When the requested value is available, the `ADC.dataReady` event is signaled. This means the previous suspended coroutine can continue its execution. The coroutine scheduler is notified by calling the `RESTORE` function. The next time this coroutine is resumed, the execution will continue after the `SUSPEND()` statement.

Besides appropriately suspending and resuming the `getData` task, the proxy for the ADC interface encapsulates all shared memory access. So, the programmer need not worry with atomic accesses to the sensed data. We have used the same step-by-step procedure to construct a proxy for the `SendMsg` interface. Using these proxies the programmer can write the simplified version of the *Surge* application, presented in Figure 2.

4. Evaluation

In order to evaluate the overhead added by coroutines, we compare the simulated behavior of the *Surge* code presented in Figure 1 and Figure 2, at the machine code level. Because we are using `setjmp/longjmp` functions to implement coroutines, it is necessary to use a program that allows for emulating, in an instruction by instruction manner, the operation of individual sensor nodes. For that, we adopted ATEMU [14], a fine grained sensor network simulator. Like TOSSIM [12], (a simulator specially designed for TinyOS programs), ATEMU simulates sensor networks in which the nodes are AVR microcontrollers [2] and runs the microcontroller program, rather than models of the software. However, TOSSIM uses a few system libraries – instead of AVR Libc library – when compiling nesC code into a binary for the development workstation. Among these libraries is the one that supports `setjmp/longjmp` functions.

Figure 4 shows the results of simulations we have conducted. We used the *ATmega128L Emulator* (version 0.4) provided by ATEMU. In the first case, we simulated the core *Surge* code (presented in Figure 1) compiled with TinyOS 1.x. In the second case, we replaced the original "sched.c" file of TinyOS 1.x by our new implementation, which adds an initialization step to allocate memory space for coroutines and a coroutine scheduler (our goal in this case was to measure the overhead imposed by the new scheduler even if we don't use coroutines). Finally, in the third case, we simulated the new version of the *Surge* code (presented in Figure 2). We are interested in comparing the number of clock cycles needed to start the application and to take the first sensor reading in each case.

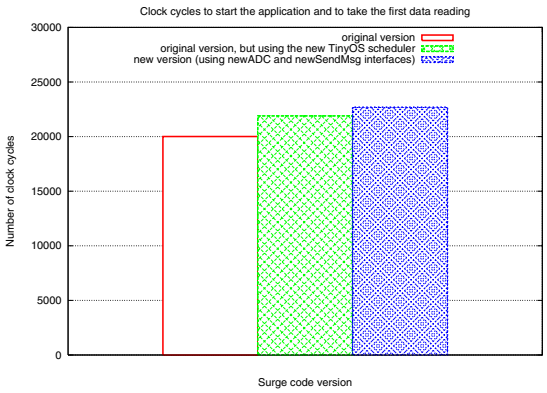

Figure 4. Clock cycle overhead added by coroutines.

The first results are encouraging. The overhead added by the new scheduler is approximately 9.4% and the further overhead added by using coroutines is about 3.6% (the total overhead in the third case is 13.4%). With respect to code size, basically we have just increased the size of the "sched.c" file (which contains all coroutine operations and the coroutine scheduler) in approximately 60%. These are the results we obtained

with our initial implementation, which we believe we can still improve.

5. Final Remarks

Software architectures for networked sensors are typically concurrent and event driven. However, event-triggered programming models are not natural for programmers: applications have to be written as explicit state machines, which are hard to understand and maintain. In this work, we proposed a coroutine-based concurrency model for networked sensors. By using the coroutine abstraction, we can build a sequential view for the programmer, without losing the event-based and asynchronous characteristics required by the networked sensors. Coroutines are a lightweight construct and seem to match well the constraints of sensor networks.

The work presented in [8] on *Protothreads* also proposes a programming abstraction which intends to reduce the complexity of high-level programs in event-triggered sensor nodes systems. Unlike coroutines, protothreads implement a type of continuation (called *local continuation*) that does not require its own stack: all protothreads run on the same stack and context switching is done by stack rewinding. The main limitation of protothreads is that variables with function-local scope are not automatically saved across blocking operations because the stack is rewound at every blocking statement. In our system, each coroutine has its own stack, and thus maintains its local variables across control transfers. Besides, since protothreads are implemented using the *C switch statement*, programs cannot utilize switch statements together with protothreads.

Welsh and Mainland [16] propose *abstract regions* to abstract interaction details between nodes in a sensor network. To simplify the programming task using *abstract regions*, the authors implemented a synchronous programming interface for TinyOS based on "lightweight threads". The system maintains two execution flows: a main flow, which is event-driven and cannot block; and an application flow, which can invoke blocked operations. The same stack is shared by these two flows of execution. Using this structure, the application can block while the system remains event-driven. In our proposal, the main flow is event-driven and the application can be divided into more than one control flow, each one with its own stack of execution.

This work is part of a project in which we study applications of cooperative multitasking. In a previous paper [15], we discussed how coroutines can be used to couple the advantages of asynchronous communication with the use of the well-known remote procedure call abstraction in geographically distributed systems.

Acknowledgments Work partly supported by CNPq: a Brazilian government research agency.

References

[1] Atmega128(l) summary. http://www.atmel.com/.

[2] Atmel corporation. http://www.atmel.com/.

[3] Avr libc. http://www.nongnu.org/avr-libc/.

[4] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Comput. Networks*, 38(4):393–422, 2002.

[5] A. Birrell. *An Introduction to Programming with Threads*, pages 88–118. Prentice Hall, 1991.

[6] D. Culler, D. Estrin, and M. Srivastava. Overview of sensor networks. *Computer*, 37(8):41–49, Aug 2004.

[7] A. L. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.

[8] A. Dunkels, O. Schmidt, and T. Voigt. Using Protothreads for Sensor Node Programming. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[9] R. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *Usenix Annual Technical Conference*, pages 239–250, San Diego, CA, USA, 2000.

[10] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The NesC language: A holistic approach to networked embedded systems. In *Conference on Programming Language Design and Implementation*, pages 1–11, New York, USA, 2003. ACM Press.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[12] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *1st ACM Conference on Embedded Networked Sensor Systems*, pages 126–137. ACM Press, 2003.

[13] C. D. Marlin. Coroutines: A programming methodology, a language design and an implementation. *Lecture Notes in Computer Science*, 95, 1980.

[14] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. Baras. Atemu: A fine-grained sensor network simulator. In *Sensor and Ad Hoc Communications and Networks*, pages 145–152, 2004.

[15] N. Rodriguez and S. Rossetto. Integrating remote invocations with asynchronism and cooperative multitasking. In *Third International Workshop on High-level Parallel Programming and Applications (HLPP'05)*, Warwick, Inglaterra, 2005.

[16] M. Welsh and G. Mainland. Programming sensor networks with abstract regions. In *USENIX/ACM Symposium on Network Systems Design and Implementation*, 2004.