Using Coroutines for RPC in Sensor Networks

Marcelo Cohen, Thiago Ponte, Silvana Rossetto and Noemi Rodriguez

Departamento de Informatica, PUC-Rio
Rua Marques de Sao Vicente, 225, Gavea, Rio de Janeiro, RJ, 22453-900, Brazil
{mca,tponte,silvana,noemi}@inf.puc-rio.br

Abstract

This paper proposes a concurrency model which integrates the asynchronous and event-driven nature of wireless sensor networks with higher-level abstractions that provide a more familiar programming style for the developer. As a basis for this proposal, we designed and implemented a cooperative multitasking scheduler, based on coroutines, for the TinyOS operating system. We then used this scheduler to implement RPC-like interfaces that capture different communication patterns common in wireless sensor networks. This allows the programmer to work, when appropriate, with a synchronous style, while maintaining an asynchronous model at the message exchange level.

1 Introduction

Sensor networks are highly unstructured and dynamic environments, in which there are no well-behaved communication patterns. Similarly to other embedded systems, sensors must respond to different stimuli, including physical events and messages from other devices. As a consequence, computing models for such systems are typically event-driven and asynchronous [2]. A representative example is the TinyOS operating system [12] and its associated nesC programming language [11], which have become practically standard tools for developing WSN applications. The nesC programming language supports the definition of *interfaces* that define *commands* that must be provided by modules that implement an interface, and *events* that may be signaled by such modules.

However, programming distributed event-based systems is typically a hard programming chore. When an operation invoked by an event cannot complete immediately, because it depends on further data that is yet to be signaled, it must be split across two or more invocations of event handlers, forcing the programmer to code "continuations" within his handlers. Events may come from many sources, both local and remote. This means that the program must typically deal simultaneously with several lines of activity. Besides, to maintain the interactivity, the system must make sure that no single event handler takes very long to execute. So, some tasks must be broken into many small pieces, between which the system saves the needed state information in global variables and returns to the main loop. This process, referred by Adya et al. as *stack ripping* [1], is one of the main difficulties for developing applications using the event-driven programming style [5].

We believe that it is possible to provide higher-level programming interfaces in a way that is consistent with an asynchronous event-based core. Cooperative multitasking seems to be a promising alternative in this scenario, allowing different activity threads to use their own stacks, while avoiding the need (and overhead) of dealing with preemption. Instead of splitting a conceptual activity across different event handlers, the programmer can have a handler yield control to the main passive loop when an operation cannot complete immediately. In [17], we described our first experiments with this model, in which we modified the TinyOS operating system to have it support coroutines and introduced a programming pattern for creating synchronous views of asynchronous interfaces.

We have now reimplemented the support for cooperative multitasking as a separate scheduling module for the TinyOS operating system. This provides better support for testing and distributing the coroutine module. In this paper, we present this implementation. We then explore how we can use our model to provide an RPC abstraction for inter-node communication. Because the number of participating devices in a wireless sensor network application can be large, we believe this is an area where it is specially important to provide the programmer with an alternative to the basic event-handler model. We also discuss how, with appropriate communication primitives, we can build RPC-like

1-4244-0910-1/07/$20.00

abstractions to fit different interaction patterns.

The RPC paradigm was probably the most popular one for client-server applications running in local-area networks. From the beginning, however, critiques to the RPC paradigm were made [19, 6], mostly discussing the imposition of a synchronous structure on the client application and the difficulty of matching RPC with fault tolerance, partly due to its one-to-one architecture. As the scope of distributed systems grew, from local-area to geographical networks, both of these aspects gained importance. New critiques to the one-to-one and synchronous RPC model have also appeared from novel application areas, such as ubiquitous computing [18]. All of these considerations and critiques led many researchers and developers to believe that the RPC model should be abandoned. However, the persistence of the model in many areas of distributed programming, even in new areas of applications such as web services, testifies its popularity among programmers.

In the context of sensor networks, an RPC-like programming model can be particularly useful to support *cluster-based architectures* [23]. Clusters are designed as a set of spatially-adjacent sensor nodes deployed around a target phenomena to sense, process and communicate data of interest. One node is elected as the cluster head, which is responsible for the control and coordination of sensor nodes within the cluster and for the interaction with other cluster heads and with the base station. Clustering techniques help the nodes minimize energy dissipation by reducing the overall message exchange in the network. We believe that RPC abstractions can be specially useful to simplify one-to-many and many-to-one interactions among cluster heads and nodes within clusters.

This paper is organized as follows. In Section 2 we present a brief introduction to wireless sensor networks and to the TinyOS operating system. In Section 3 we describe how we implemented support for coroutines in TinyOS. Next, in Section 4, we discuss the implementation of remote procedure calls using coroutines. Section 5 contains some discussion and related work, and, to conclude, Section 6 presents some final remarks.

2 Wireless sensor networks

The use of wireless sensor networks has been growing, and they have become an important basis in a number of distinct applications, such as monitoring ecological or security conditions. A sensor network may consist of potentially thousands of tiny, low-power nodes, each of which executes 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 of such an event is message arrival. Typically, communication is radio-based and the radio is an asynchronous input/output device that contains no buffer, so each packet must be serviced by the node as soon as it becomes available. Systems for sensor networks 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 [12] is the current state of the art in operating systems for sensor network research. In the next section we describe some important design aspects of TinyOS.

2.1 The TinyOS design

The design of TinyOS 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. Typically, a command handler deposits request parameters and conditionally posts a task for later execution. *Events* are signaled to indicate service completion or hardware events. An event handler can deposit information in the component's environment, post tasks, signal higher-level events or invoke commands.

Tasks allow for 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.

There are two different programming constructors in TinyOS: *modules*, that are used to provide code; and *configurations*, that are used to wire components together. The behavior of a TinyOS component 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* [11] was specially designed to provide the event-driven concurrency model used by TinyOS.

3 A coroutine-based concurrency model

Although the TinyOS programming model is well suited to the constraints of sensor networks, it is not always easy to use. Its multitasking engine maintains a two-level scheduling structure that forces the program into the structure of

a finite state machine, which, when many states and events are involved, can be difficult to understand and maintain. To develop even a simple sensing application in nesC, the programmer must typically partition basic requests into two-phase operations. To share data between these operations, he must resort to global variables. He cannot follow the classical programming discipline of maintaining data pertaining only to a certain activity in local variables.

In Figure 1 we illustrate these difficulties with a simple request-sense-forward application: basically, whenever a request sensing message is received, the node reads and forwards its sensor value. In our example, we use standard nesC interfaces to receive a sensing request (`Receive`), collect a sensor value (`AcquireData`) and forward the collected data (`Send`). When the `Receive.receive` event is signaled (on message arrival), a sensor reading is started by calling the `AcquireData.getData` command. When the `AcquireData.dataReady` event is signaled, a message is constructed with the collected data and forwarded using the `Send.send` command.

In this simple example, the main application task (receiving a request, taking the sensor value and forwarding it) must be partitioned into three distinct pieces of code and implemented as a state machine. This is because operations used to acquire sensor values and to forward messages across the network cannot complete immediately, i.e., they are typical split-phase operations with a command to request the operation and an event to signal operation completion. Besides, since a new request can be received before the last one was completely handled, we (the programmer) need to coordinate the state transitions of the application. In this example, if a new sensing data message was constructed before the last `Send.send` command was completed, it could erroneously overwrite the data to be sent. So, we define a global variable named `g_locked` that is used to ensure correct execution of the application.

Our goal is to provide a more intuitive programming abstraction to the developer. A classical and more convenient way to receive a request, get data readings and forward them would be to implement a single language procedure with a loop to receive a new request, to get the sensor value and to send the collected data, like this:

```
void BasicApp() {
  while(TRUE) {
    // wait a new request
    receiveRequest(message_t* msg,...);
    // read the sensor data
    err = getData(&data);
    // send data
    if (!err) sendData(data,...);
  }
}
```

In other words, we would like to 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.

```
module BasicAppC {...}
implementation {
  message_t g_packet;
  bool g_locked = FALSE;
  ... // initialization code

  event message_t Receive.receive(message_t* msg,..)
  {
    call AcquireData.getData();
    return msg;
  }

  event void AcquireData.dataReady(uint16_t data)
  {
    if (g_locked)  return;
    else {
      Msg* rsm;
      rsm = (Msg*) call
                   Packet.getPayload(&g_packet,..);
      rsm->data = data;
      if (call Send.send(&g_packet,...))==SUCCESS)
         g_locked = TRUE;
    }
  }

  event void Send.sendDone(message_t* bufPtr,...)
  {
    if (&g_packet == bufPtr)
      g_locked = FALSE;
  }
}
```

Figure 1. A simple TinyOS application.

The most popular way of providing local variables for different threads of activity is to use multithreading, which is exactly what TinyOS avoids with its event-based structure. Multithreading is typically preemptive, meaning that control switches can occur at any moment, imposing an overhead and introducing arbitrary race conditions. Coroutines, on the other hand, introduce multitasking in a cooperative fashion: each coroutine has its own execution stack, as a thread does, but control is transferred only through the use of explicit control transfer primitives. The switch between any two threads of execution is explicit in the program. This means that the point where possible interleaves may occur, with associated accesses to global memory, are clearly marked in the program, avoiding several of the problems related to race conditions. Besides, the overhead associated to a context switch will be incurred only when explicitly requested by the program: this is specially relevant in the constrained context of sensor networks.

Coroutines also fit in particularly well with our desire to maintain the original event-based programming model available to the programmer alongside the new synchronous view. In the original TinyOS model, a new event is handled only when the previous handler has completed execution. Using coroutines, a new event will be handled when either the previous handler has completed execution, as before, or has yielded control, if it was running as a coroutine and has explicitly requested a blocking operation.

3.1 Coroutine implementation for TinyOS

In our previous work [17], we extended TinyOS 1.x task scheduler to include: a *coroutine construct* implementation, a *coroutine queue* (with pre-allocated space) and a *coroutine scheduler* to resume coroutines that are ready. In this work, we explore features offered by TinyOS 2.x — particularly the design of the task scheduler as a component — in order to provide coroutine facilities for nesC applications without the need to change the operating system core.

A coroutine is represented by a code address and has its own stack. As discussed in [8], there are different syntactic and semantic ways to support coroutines. Symmetric coroutine facilities provide a single control-transfer primitive (such as Modula-2's *transfer* operation [22]), that allows coroutines to explicitly pass control between themselves. Asymmetric coroutine mechanisms provide two control-transfer operations: one for invoking a coroutine and one for suspending it. Suspending a coroutine implicitly returns control to its invoker. If the coroutine ends its execution normally, control is also returned to the invoking coroutine. We have adopted asymmetrical coroutines, allowing arbitrary functions independently written to be invoked as coroutines. The interface we designed for the provision of asymmetric coroutines is called `CoRoutine` and defines four commands:

- `void postCoRoutine(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 coro_id)`: informs the scheduler that the given coroutine is ready to be resumed.

Coroutine construct The next step was to implement these commands. We designed a component called `CoRoutineC` which implements the `CoRoutine` interface and an internal set of operations needed to support a coroutine construct. These operations involve: allocating memory for a coroutine stack, associating a function to a coroutine, transferring control to a coroutine, and yielding control from a coroutine.

One simple way to implement coroutines is by using the `setjmp` and `longjmp` functions [10]. 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`.

Our implementation uses `setjmp/longjmp` functions and is based on the PCL (*Portable Coroutine Library*) library designed by Libenzi [14]. We implemented coroutine operations for the ATmega128L [4] microcontroller which is adopted in sensor platforms such as Mica2, MicaZ and Mica2Dot [7]. The ATmega128L microcontroller has 128KB for program memory and 4KB for data memory. Coroutines stacks are allocated on the heap with default size of 256 bytes. At compile-time, it is possible to know the number of coroutines defined by the application, so we allocate the exact number of coroutine stacks required by each application.

Coroutine scheduler In TinyOS 2.x, the task scheduler is a component, so we can replace the standard TinyOS scheduler by simply changing the default scheduler component. The standard TinyOS scheduler is described by a configuration component called `TinySchedulerC`. To replace this scheduler, one must define a new `TinySchedulerC` configuration component in the application directory. All scheduler implementations must provide both a parameterized `TaskBasic` interface and a `Scheduler` interface. The former supports nesC *post* statement (to schedule a task) and *task* declarations (to declare a task) and enables TinyOS core systems to operate properly; the latter is used to initialize and run tasks. As long as these interfaces are provided, the TinyOS core code can run, unchanged, with new scheduler implementations.

In the new configuration, we provide in addition to TaskBasic and Scheduler interfaces, the CoRoutineSched interface in order to support coroutine facilities. The CoRoutineSched interface is similar to the TaskBasic interface. It includes an event (`runCoRoutine`) that is signaled by the scheduler and allows transferring control to a coroutine; and a command (`postCoRoutine`) that is called when a component wants to schedule a coroutine. The main component in our `TinySchedulerC` configuration is called `SchedulerCoroP`. It is a module that implements CoRoutineSched, Scheduler and TaskBasic interfaces. The McuSleep interface is used for microcontroller power management and is not relevant for this work. With this scheduler implementation, the `CoRoutine` interface becomes available to applications.

3.2 Evaluation

In order to evaluate the overhead added by coroutines, we simulated the behavior of a simple TinyOS sensing application in which a node takes a sensor reading by using the `AcquireData` interface (a command is called to get a new sensor value and an event is signaled when the value is available). In our simple application, each time the `Timer.fired` event is signaled,

the `AcquireData.getData` command is called and then, when the `AcquireData.dataReady()` event is signaled, the operation is finished.

We have implemented two versions of this application: the first one using original TinyOS interfaces and the second one including coroutines. By using coroutines we can encapsulate both the invocation of `AcquireData.getData` and the handling of its result into a unique procedure, eliminating the split-phase behavior.

We used the *ATmega128L Emulator* provided by ATEMU [15]. First, we simulated the original version (without coroutines and our new scheduler). After that, we simulated the new version of the application, which explores coroutines. We then compared the number of clock cycles needed to take a new sensor reading after the last reading is completed.

The number of clock cycles used with the original TinyOS scheduler was 104,213, and with our new scheduler 122,982. The overhead added by using coroutines for taking a sensor reading was thus about 18.01%. Although not despicable, we believe this is a reasonable price to pay for the possibility of simplifying the task of the programmer. Furthermore, these are the results we obtained with our initial implementation, which we believe we can still improve.

4 RPC in sensors networks

Vinoski [20] argues that the complexity added with asynchronous interactions is a consequence of mixing the communication style at message exchange level with the communication semantics at the application level. Programming abstractions must hide low-level issues such as network and message exchange details behind a more familiar programming idiom. For the application developer, when it is necessary to obtain certain information before taking the next step in the process execution, the synchronous view is fundamental. Lea et al. [13] emphasize that all computer systems are essentially asynchronous and that the notion of synchronous operation is merely a convenient programming style which must be built over an asynchronous communication basis.

Remote Procedure Calls (RPC) have been popular in distributed setting since the eighties. The idea of extending the main programming abstraction we use in sequential programs to distributed settings seems to appeal to many programmers. However, as the focus of distributed programming shifted from local area networks to wide-area, many critiques have been placed against RPC, often related to the blocking nature of the original model and to its one-to-one nature. We believe it is possible to explore the advantages of programming with RPC in a way that is compatible with the requirements of asynchronous and many-to-many applications. Messages can be exchanged asynchronously at a basic level, but the programmer can see communication as a remote procedure call if it suits him. At the same time, it is important to provide the programmer with all the different programming styles that he may need: so remote procedure calls must not be the only alternative for communication.

We explored the coroutine facilities supported by our new TinyOS scheduler and CoRoutine interface to design and implement a synchronous view of basic Send/Receive primitives (interfaces) provided by TinyOS. This view can be freely combined with the original TinyOS model of commands and events and can be used to simplify programming. In the next sections we describe how we built this synchronous view and how it can be used in a RPC programming style for sensor networks applications.

4.1 Synchronous view of Send/Receive

TinyOS provides distinct interfaces to abstract the underlying communications services and a number of components that provide (implement) these interfaces. The `Send` interface provides the basic address-free message sending interface designed for broadcasting. `AMSend` is similar to `Send`, but takes a destination AM address in its send command. These interfaces provide a command for sending a message and an event to indicate whether a message was sent successfully or not. The `Receive` interface provides the basic message reception interface through an event for receiving messages.

To build a synchronous view of basic Send/Receive interfaces offered by TinyOS, we first designed synchronous interfaces named `AMSendS` and `ReceiveS`. The main difference between `AMSend` and `AMSendS` interfaces is the absence of the `sendDone` event in the latter. The idea is that the `send` command should return only after completion (this is the synchronous view of the command). However, instead of maintaining the entire application blocked waiting for message to be sent, we implement this command in a coroutine, so it is suspended and resumed appropriately and the system remains asynchronous. A similar idea is explored for receiving. Instead of defining the `receive` operation as an event, we define it as a command in `ReceiveS` interface. In this case, the current context (or coroutine) is suspended until a new message is received. By using the new interface, the programmer can call the `ReceiveS.receive` command at the point in the program where a message is expected.

Implementing synchronous view of Send/Receive interfaces We implemented two components to provide interfaces `AMSendS` and `ReceiveS`. These components use operations provided by the CoRoutine interface to suspend and restore the current context of execution.

```
implementation {
  uint8_t g_coro_id; // coroutine identification
  error_t g_err;     // error control
  command error_t AMSendS.send
                (am_addr_t addr,
                 message_t* msg, uint8_t len) {
    g_err = call AMSend.send (addr, msg, len);
    if (g_err != SUCCESS)
       return g_err;
    g_coro_id = call CoRoutine.getId();
    call CoRoutine.suspend();
    return g_err;
  }

  event void AMSend.sendDone
                (message_t* msg, error_t error) {
    g_err = error;
    call CoRoutine.restore(g_coro_id);
  }
}
```

Figure 2. AMSendS implementation.

Figure 2 presents the implementation of the `AMSendS` interface. When the `AMSendS.send` command is called, the standard `AMSend` interface is used by calling the `AMSend.send` command. Current coroutine identification is stored and the context of execution is suspended. When the `AMSend.sendDone` event is signaled, the coroutine that was previously suspended can be restored.

The implementation of the `ReceiveS` interface follows similar ideas and uses the standard `Receive` interface. When the `ReceiveS.receive` command is called, current coroutine identification is stored and the context of execution is suspended. When the `Receive.receive` event is signaled, the coroutine that was previously suspended can be restored. In order to deal with the possibility of failures, we have introduced a timer that defines an upper bound for the time a coroutine remains suspended waiting for a message. If this upper bound is reached before a message arrives, `ReceiveS.receive` will return a `TIMEOUT` code.

4.2 Synchronous RPC in sensor networks

By using the synchronous view of send/receive interfaces, we can easily design stubs and proxies that support synchronous remote calls. We discuss this possibility by means of a simple example. We design an interface, named `RemoteData`, with a command to get sensing values from neighboring nodes. This command has four parameters: the destination address (one can use `AM_BROADCAST_ADDRESS` for broadcasting, or a specific address for a particular node), a reference value that will receive the remote value, a time limit to wait for a reply and the sensor type (e.g., TEMP, LIGHT, etc.). Figure 3 shows the component that implements the `RemoteData` interface. It constructs a message with the appropriate request and uses the `AMSendS.send` command to send this message. After message sending

```
implementation {
  command error_t RemoteData.get
      (am_addr_t addr, uint16_t* value,
           uint8_t timer, uint8_t type) {
    message_t packet;
    void payload;
    request_t req_msg;
    result_t res_msg;
    // compose a packet with the request ...
    // send the request
    if (call AMSendS.send
           (addr, &packet,...) == SUCCESS) {
       // wait for result
       error_t err = call ReceiveS.receive
                              (..., &payload, timer);
       if (err == SUCCESS) {
          result_t *res_msg = (result_t*) payload;
          *value = res_msg->data;
       }
       return err;
    }
    return FAIL;
  }
}
```

Figure 3. Proxy to take remote sensing.

```
void BaseStation() {
  uint16_t g_value;
  task ProcessTemp() {
    // do something
    if (g_data > NORMAL_TEMP) ...
  }
  event void Timer.fired() {
    uint16_t value;
    error_t err = call RemoteData.get
      (AM_BROADCAST_ADDRESS, &value, TIMER, TEMP);
    if (err != TIMEOUT)
       if (post ProcessTemp()) g_value = value;
  }
}
```

Figure 4. Example using remote invocation.

is successfully completed, the `ReceiveS.receive` command is called to wait for the request result. When this command is completed, the received value is extracted from the message and returned as a parameter of reference.

The command `RemoteData.get` can be periodically called by the application in a base station (or in a cluster leader) to get a sensor reading from any neighboring node. The code in Figure 4 shows an example.

The `RemoteData` interface explores features of wireless communication, such as message broadcast. Since the number of neighbors of a node is normally dynamic (nodes can move, turn themselves off to save power, or become unreachable for reasons like message collisions), broadcasting messages is a good alternative to reach neighbors without the cost of maintaining network infrastructure. In the example shown in Figure 4, only one reply is sufficient for each remote request, so the first received message is used. It is also possible to direct the request to a specific node by means of the `addr` parameter.

```
implementation {
  command uint8_t RemoteBroadData.get
          (uint16_t *values,
           uint8_t timer, uint8_t type) {
    message_t packet;
    uint8_t count=0;
    void payload;
    request_t req_msg;
    result_t res_msg;
    // compose a packet with the request...
    // send the request
    if (call AMSendS.send
             (AM_BROADCAST_ADDR,
              &packet,...) == SUCCESS) {
       // wait for results
       while(TRUE) {
         error_t err = call ReceiveS.receive
                        (..., &payload, timer);
         if (err == TIMEOUT)  break;
            result_t *res_msg = (result_t*) payload;
            *(values+count) = res_msg->data;
            count++;
         }
       }
    }
    return count;
  }
}
```

Figure 5. Proxy for multiple remote readings.

However, there are other situations in which a number of replies for a broadcasted message are desired and must be handled. This approach can be used, for example, to aggregate values from distinct sensor nodes in a cluster-based architecture. In Figure 5, we show a distinct implementation for a remote sensing request, which we called `RemoteBroadData.get`. In this case, a node (for example, a cluster head) broadcasts a request and waits for a number of replies. The command will return the number of received values and a vector with these values, so the application can define how to process the collected values. In our example, the timer parameter is kept the same whenever the `ReceiveS.receive` command is called. This is reasonable considering that a cluster head normally defines some kind of TDMA protocol to allocate time slots for each node to send messages to it (in order to avoid message collisions). Thus, the timer parameter can be defined by considering the length of time slots. Another approach could be to decrease the timer value at each received message.

5 Discussion and related work

The work presented in [9] 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.

Welsh and Mainland [21] 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.

May et al. [3] describe the design and implementation of an RPC mechanism for wireless sensor networks. Their proposal includes extensions to the nesC programming language that allow the programmer to specify, in a configuration, that an interface is to be bound to a remote provider. This allows the configuration to document remote interactions, but on the other hand makes the implementation dependent on compiler extensions. Syntactic extensions are also present in interface descriptions, to establish that a command may be invoked remotely. The remote calls themselves are always asynchronous in the sense of the one-way CORBA operations. The rationale behind this is to avoid busy waiting and to maintain the original split-phase model of TinyOS.

One point of consensus that seems to be emerging from the diverse needs of current applications is that there is no single set of programming abstractions that will be adequate for all applications. The same seems to be true even inside a single application. In the setting of wireless sensor networks, it may be the case that a remote procedure call is the most appropriate way of requesting a given value from a neighbouring node, while the direct use of events may be more appropriate to transmit "out-of-band" data such as alarms. When we insert the cooperative multitasking module in the TinyOS system, the original programming model remains available. The programmer can choose whether to use a synchronous or asynchronous view in each local or remote interaction.

6 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 sensor networks and showed how it can be used to couple higher-level programming abstractions with the basic event-driven I/O model. Coroutines are a lightweight construct and seem to match well the constraints of sensor networks. The Remote Procedure Call abstraction is a familiar concept which allows the programmer to capture many common interaction patterns.

The design of the RPC system we describe also illustrates advantages of the support for modularity offered by the TinyOS operating system. The possibility of redefining the scheduler as a component allows us to redefine system behavior without tampering with its internals, allowing system and scheduler versions to be independently updated. The design of a low-level interface which offers synchronous *send* and *receive* operations also seems adequate, for it allows different communication patterns, such as one-to-one and one-to-many, to be programmed as new components or in the final application.

This work is part of a project in which we study applications of cooperative multitasking. In a previous paper [16], 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. In the sensor network domain, we are particularly interested in exploring cooperative multitasking to simplify the programming task.

Acknowledgments This work was partly supported by CNPq, the Brazilian Research Council, grants number 152101/2005-5 and 305320/2004-1.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

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

[3] T. D. M. andS. H. Dunning and J. O. Hallstrom. An RPC design for wireless sensor networks. In *Proceedings of the 2nd IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, Washington DC, USA, Nov 2005.

[4] ATmega128(L) summary. http://www.atmel.com/.

[5] R. Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Hawaii, USA, May 2003.

[6] K. Birman and R. van Renessee, editors. *Reliable Distributed Computing with the Isis Toolkit*, chapter RPC considered inadequate, pages 68–78. IEEE Computer Society Press, 1994.

[7] Crossbow Technology, Inc. http://www.xbow.com/.

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

[9] 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.

[10] 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.

[11] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[12] 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*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[13] D. Lea, S. Vinoski, and W. Vogels. Asynchronous middleware and services. *IEEE Internet Computing*, 10(1):14–17, 2006.

[14] D. Libenzi. Portable coroutine library (PCL), 2005. http://xmailserver.org/libpcl.html.

[15] 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.

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

[17] S. Rossetto and N. Rodriguez. A cooperative multitasking model for networked sensors. In *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*, Lisboa, Portugal, 2006.

[18] U. Saif and D. Greaves. Communication primitives for ubiquitous systems or RPC considered harmful. In *Workshop on Smart Appliances and Wearable Computing (in conj. with ICDCS'01)*, Mesa, AZ, 2001.

[19] A. Tanenbaum and R. van Renesse. A critique of the remote procedure call paradigm. In *EUTECO'88 Conf.*, pages 775–783, Vienna, 1988. Participants Edition.

[20] S. Vinoski. RPC under fire. *IEEE Internet Computing*, 9(5):93–95, 2005.

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

[22] N. Wirth. *Programming in Modula-2*. Springer-Verlag, third edition, 1985.

[23] Y. Yu, B. Krishnamachari, and V. K. Prasanna. Issues in designing middleware for wireless sensor networks. *IEEE Network*, 18:15–21, 2004.