

# SpartanRPC: Secure WSN Middleware for Cooperating Domains

Peter Chapin  
The University of Vermont  
Burlington, VT  
pchapin@cs.uvm.edu

Christian Skalka  
The University of Vermont  
Burlington, VT  
skalka@cs.uvm.edu

## ABSTRACT

In this paper we describe SpartanRPC, a secure middleware technology for wireless sensor network (WSN) applications supporting cooperation between distinct protection domains. The SpartanRPC system extends the nesC programming language to provide a link-layer remote procedure call (RPC) mechanism, along with an extension of nesC configuration wirings that allow specification of remote, dynamic endpoints. SpartanRPC also incorporates a capability-based security architecture for protection of RPC resources in a heterogeneous trust environment, via language-level policy specification and enforcement. We discuss an implementation of SpartanRPC based on program transformation and AES cryptography, and present empirical performance results.

## Categories and Subject Descriptors

D Software [D.3 Programming Languages]: D.3.3 Language Constructs and Features

## General Terms

Design, Languages, Security.

## Keywords

Remote procedure call, capability-based security, wireless sensor networks

## 1. INTRODUCTION

WSN applications are orchestrations of behavior among multiple nodes to produce a holistic effect. As WSN technology becomes more popular, applications are becoming larger and in some cases support cooperation between distinct social entities. For example, applications for emergency first response support coordinated actions between multiple medical and governmental organizations [5, 11]. In systems incorporating WSNs, technology at the WSN level reflecting such interactions is a natural evolution that has been anticipated by other researchers, studying diverse topics such as secure routing protocols in heterogeneous trust environments [10],

transport and network layer protocols [16], even mote-based web servers supporting secure channels [6].

In this paper we describe novel middleware technology to support WSN applications in this type of setting. We call our system SpartanRPC to evoke both its light weight and strong defense mechanisms. SpartanRPC comprises a new form of link-layer *remote procedure call (RPC)* as a natural extension of the nesC programming model, and *language based authorization* based on symmetric-key cryptography. As other authors have observed [13], RPC is an appropriate abstraction for node services on the network and supports whole-network (vs. node-specific) programming. We further observe that RPC allows nodes to provide flexible, modular services without the need for reprogramming, a kind of “micro web services”. *Secure RPC* is also clearly desirable in a heterogeneous trust environment, to disallow access to services by unauthorized parties. SpartanRPC provides new syntactic forms for specifying RPC security policies and for authorizing RPC invocations in networks comprising multiple protection domains.

## 1.1 Overview and Contributions

Previous related work illustrates interest in and useful applications of RPC in a WSN context. For example, the Marionette system uses network layer RPC for remote (PC-based) analysis and debugging of WSNs [20]. The Fleck operating system provides a small pre-defined set of RPC services for WSN applications, while the secFleck system extends this with a form a secure RPC [7]. SpartanRPC differs from these systems in that it extends the nesC programming language to allow user definition (unlike secFleck) of secure RPC services that can be accessed by nodes within the network itself (unlike Marionette). Our system is similar to and inspired by TinyRPC [13], except the latter does not provide security and has a different semantics that is not as expressive and flexible as our approach. Major contributions of our work include an RPC design that is consistent with existing nesC semantics, including an asynchronous “task-like” conception of RPC and *dynamic wires* as a natural extension of configuration wirings to allow flexibility in remote communication. We also provide a mechanism for fine-grained RPC authorization. Empirical results discussed in section 7 show that these features are not onerous in terms of additional space and energy consumption.

A primary goal of the SpartanRPC design is to provide RPC capabilities as transparently as possible. This means at least that the low-level details of RPC communication should be hidden from the user. Beyond that, it means that RPC features should be provided in a manner that fits in with existing nesC semantics. For example, RPC in WSNs must be asynchronous, since remote communication is time consuming and failure-prone. Thus RPC invocation should be expressed in a similar manner as existing asynchronous nesC control flow mechanisms (i.e. tasks). And RPC authorization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

policies should be expressed and enforced as control flow security policies, with details of underlying symmetric key cryptography remaining transparent.

### *Asynchronous Execution Model.*

In nesC a `command` is a synchronous unit of control; when one is `called`, it is pushed onto the call stack and the current continuation waits for it to complete and yield a return value. In contrast, when a `task` is `posted` it is placed in a queue for execution at some future time, and the calling context continues execution. We therefore propose a `task`-like mechanism for RPC invocation, that allows *remote postings*; in this our approach differs from TinyRPC that envisioned RPC as a form of `command` [13]. However, our mechanism differs from `tasks` in two important ways: first, we allow arguments to be passed in RPC calls, and second, the module that `posts` a `task` must define it, whereas we want to allow modules to execute functionality defined non-locally. We therefore introduce the `duty` mechanism. Duties may be `posted` like a `task`, but passed arguments and `provided` by modules for `posting` by other (possibly non-local) modules. Duties are discussed in section 2.

### *Dynamic Wires.*

The SpartanRPC design is inspired by TinyRPC [13] in its minimalist extension of configuration wiring syntax with abstractions for remote communication. In our system components can provide remotable interfaces, and in configuration definitions these interfaces can be wired to locally or non-locally. However there is a difference in our system, where wirings are inherently dynamic. Wirings to `remote` interfaces must specify the host of the wired-to component, and our wiring syntax requires the user to provide an inherently dynamic definition of endpoint hosts to allow fan-out wirings that can change at run-time. Since SpartanRPC works at the link layer, we believe this flexibility is fundamental in a general WSN setting where neighborhoods can be expected to change frequently due to node repositioning, failure, or changes in radio signal strengths due to environmental conditions. Dynamic wires are discussed more in section 3.

### *Fine-Grained Authorization Policies.*

Research on WSN security has addressed secure routing [10], link layer security [9], cryptography [1] and key distribution [2], and hardware issues [17]. This previous work has established a strong low-level foundation for security in wireless sensor networks. We believe that secure link-layer RPC is an appealing middleware solution in WSNs because of bang-for-buck— it allows programmatic specification of previously ad-hoc network-level behavior, while imposing relatively small syntactic and efficiency overhead. SpartanRPC also allows multiple symmetric keys to be used to protect multiple security domains within a network in a simple and usable manner. This in turn allows different protection domains in a single WSN to specify and enforce their own security policies within the network. In our SpartanRPC implementation we leverage AES encryption and MAC protocols to provide authentication. These underly a simple capability-based authorization mechanism presented to the user. Our security mechanism is discussed in more detail in section 4.

## 2. DUTIES AND REMOTABILITY

In previous related work, the TinyRPC designers proposed `commands` as the basic unit of abstraction for network RPC services [13]. However, nesC `commands` are fundamentally synchronous;

when a calling context invokes a `command`, the current continuation waits for the `command` to complete execution and provide a return value. The so-called “split phase” semantics of nesC provides a means for callbacks from lower levels on protocol stacks via events; for example a `command` might be used to start an analog to digital conversion operation and an event would be used by the ADC component to later report the converted value. However, any asynchronicity in such operations is obtained via *task postings*, not `commands`.

Thus, since it is unrealistic for RPC services in WSNs to be synchronous as discussed in section 1, we believe that the semantics of `tasks` are closer to being a correct fundamental abstraction. They are not quite right however, as RPC services will typically require arguments to be passed (whereas `tasks` take no arguments), and the poster of a `task` defines it, whereas remotely defined functionality is invoked via RPC by convention; a remote `task` posting would have more in common with mobile code than RPC. We therefore define a new RPC abstraction called a *duty*.

A `duty` is similar to a nesC `task` in that it is executed asynchronously with respect to its invoker and does not return any results. In essence, RPC server nodes expose functionality to clients in the form of `duties` and clients invoke that functionality by `posting` `duties` on remote servers. However, unlike a nesC `task`, a `duty` can take parameters and `duties` are defined on the remote device, not the invoking device. Also because of the unreliable nature of radio communications an invoked `duty` might not execute at all.

### 2.1 Syntax and Semantics

`Duties` are declared in interfaces and syntactically resemble `command` declarations. Consequently the syntactic extensions required for `duties` are minor. Instead of using the reserved word `command` as a storage class specifier in declarations, the new reserved word `duty` is used. `Duties` are allowed to take parameters (with restrictions as discussed below) but must return the type `void`. It is a compile time error if a `duty` is declared with any other return type. For example the following interface describes an RPC service for remotely flashing a mote LED:

```
interface LEDControl {
    duty void setLeds(uint8_t value);
};
```

`Duties` are defined in modules in a manner similar to the way `tasks`, `commands`, or `events` are defined. The reserved word `duty` is used as a storage class specifier on the definition. Like `commands` and `events` the name of the `duty` is qualified by the name of the interface in which it is declared. Including a `duty` in an interface definition automatically implies that the interface can be remotely invoked, or is *remotable* in the sense formalized in subsection 2.2. Any remotable interface provided by a component must be specified as `remote` in its provides specification, for example:

```
module LEDControllerC {
    provides remote interface LEDControl;
}
implementation {
    duty void LEDControl.setLeds(uint8_t value)
    {
        // ...
    }
}
```

The use of the `remote` reserved word is redundant; Sprocket is able to infer which interfaces are remotable from their definitions;

see subsection 2.2. However, declaring interfaces `remote` explicitly in the module specification promotes program readability. It is a compile time error if a remotable interface is not declared `remote` or if an interface is declared `remote` that is not remotable.

A module on the client node that wishes to use a remotable interface simply posts the duty in the same manner as tasks are posted. The use of `post` emphasizes the asynchronous nature of the invocation.

```
module LoggerC {
  uses interface LEDControl;
}
implementation {
  void f()
  {
    post LEDControl.setLeds(5);
  }
}
```

Note that the standard component semantics of nesC provide here a natural abstraction of “where” the RPC call goes, just as e.g. a normal command invocation will go through a component interface that is disconnected from its implementation. And like normal command invocation, it is configuration wirings that determine to where duty control flows in SpartanRPC, except upon duty invocation control may flow to a component residing on a different network node. This module must be connected to the remote modules by way of a dynamic wire as described in section 3.

When a duty is posted by a client node it may run at some time in the future on the server node. The client node continues at once without waiting for the duty to start, i.e. duty postings are asynchronous in the same manner that tasks are. Once posted the client has no direct way to determine the status of the duty.

It is possible for a duty to be posted multiple times by a client or by multiple clients. Because duties are implemented as nesC tasks as discussed in section 6, any posts of a particular duty received by a node while a previous post of that duty is pending are lost. However, this does not introduce any new problems because duty execution is not guaranteed in any case.

## 2.2 Remotable Interfaces

We impose certain requirements on RPC service definitions for ease of implementation. First of all, since WSN nodes do not share state we want to disallow passing references to duties—such a reference would be meaningless on the receiving node. Thus we define remotable types:

**DEFINITION 2.1.** *A type is remotable iff it satisfies the following inductive definition: The nesC built-in arithmetic types, including enumeration types, are remotable, and arrays of remotable types and structures containing remotable types are remotable.*

Since a remotable interface describes RPC services, we require that they specify that duties only take arguments of remotable type; also, remotable interfaces can only contain duties, to ensure meaningful remote usage.

**DEFINITION 2.2.** *An interface is remotable iff it only provides duties whose argument types are remotable.*

## 2.3 Implementation Limitations

The SpartanRPC implementation uses statically allocated arrays to store duty parameters, and so have fixed size. The current implementation restricts the number of duties in a remotable interface to 16. This limitation is arbitrary and could be changed by suitably

modifying the format of the SpartanRPC data packet as described in section 6.

The current implementation also limits the total size of all the duty parameters taken together to no more than 12 bytes. This assumes that the minimize sized packet header is used; in some applications the maximum allowed size of all duty parameters will be even less. While this restriction is significant we anticipate that the current limits will be acceptable for many applications of SpartanRPC. These limits are not fundamental to the design of our system; they could be raised at the expense of complicating the implementation.

In the current implementation duties can only be declared in interfaces. Furthermore if an interface declares even one duty, it must declare only duties. It is a compile time error if a command or event is declared in the same interface as a duty.

## 3. DYNAMIC WIRES

In an ordinary nesC program the “wiring” between components as defined by configurations is entirely static. The nesC compiler arranges for all connections and at run time the code invoked by each called command or signaled event is predetermined.

In a remote procedure call system for wireless networks, this static arrangement is insufficient. A node can not, in general, know its neighbors at compilation time but rather must discover this information after deployment. In addition, the volatility of wireless links, and of the nodes themselves, means that a given node’s set of neighbors will change over time. Representing the connections between nodes as static wires in general is impossible. In this Section we discuss a facility in SpartanRPC to allow *dynamic wirings* for remote control flow from duty invocation via remotable interfaces to RPC service (duty) implementation, wherein the programmer has control over wiring endpoints and how they may change during program execution.

### 3.1 Component IDs, Component Managers

We begin by discussing how remote components are identified for wiring. In order to uniquely identify components on the network, remotable components are specified via a two-element structure called a `component_id` defined in Figure 1. The `node_id` component is the same node ID used by TinyOS and is set when the node is programmed during deployment. The local ID is an arbitrary value defined by the programmer of the server node. Only components that are visible remotely need to have ID values assigned, however, the ID values must be unique *on the node*. Note that they need not be globally unique since `component_ids` are additionally distinguished by `node_ids`, simplifying applications.

A *component manager* is a component that dynamically specifies a set of component IDs that ultimately serve as dynamic wiring points in configurations. Formally, a component manager is a component that provides the `ComponentManager` interface defined in Figure 1. Thus, a component manager implements a command `elements` which when invoked will return a set of `component_ids`.

As a simple example, consider the following component manager `RemoteSelectorC`:

```
module RemoteSelectorC {
  provides interface ComponentManager;
}
implementation {
  component_id broadcast = { 0xFFFF, 1 };

  component_set
    ComponentManager.elements()
}
```

```

interface ComponentManager
{
    command component_set elements();
}

typedef struct {
    int count;
    component_id *ids;
} component_set;

typedef struct {
    uint16_t node_id;
    uint8_t local_id;
} component_id;

```

**Figure 1: Component Manager Interface and Type Definitions**

```

{
    component_set result = { 1, &broadcast };
    return result;
}
}

```

This component manager always returns a component set containing a single component. The special SpartanRPC broadcast component ID is used (0xFFFF) indicating that all neighbors should be the target of the dynamic wire. The component ID on the neighbors is specified as 1 in this case. In a more complex example the component manager could compute the component set each time the dynamic wire is used, filling in an array of component IDs based on information gathered earlier in the node’s lifetime. We consider such an example in section 5.

### 3.2 Syntax and Semantics

In SpartanRPC we extend the syntax and semantics of nesC to allow the target of a connection to be dynamically specified by a component manager. The syntax of wirings, or connections, is extended as in Figure 2, to allow a `dynamic_endpoint` to be wired to, versus an ordinary nesT `endpoint` which statically specifies the interface of a component.

```

connection ::=
    endpoint '->' dynamic_endpoint

dynamic_endpoint ::=
    '[' IDENTIFIER ']' ('.' IDENTIFIER)*

```

**Figure 2: Syntax for Dynamic Wires**

Given a dynamic wiring of the form `C.I -> [RC].I`, we informally summarize its semantics as follows. First, we statically require that `RC` is a component manager, and that `I` is remotable. At run time, if control flows across this wire via posting of some duty `I.d` within `C`, the method `elements` in `RC` is invoked to obtain a set of component IDs. The duties `I.d` provided by those remote components will then be posted on the host machines via an underlying remote communication, the details of which are hidden from the SpartanRPC programmer. Note that since this call to `elements` may return more than one component ID, this is a sort of fan-out wiring.

Combining and extending the examples begun in section 2 where a remotable interface `LEDControl` was defined, and subsection 3.1 where a component manager `RemoteSelectorC` was defined, consider a simple service that allows client nodes to turn on or off three LEDs on the server node. A client that wishes to use such a service could indicate its connection with one or more server nodes using a configuration such as:

```

components ClientC, RemoteSelectorC;

ClientC.LEDControl ->
    [RemoteSelectorC].LEDControl;

```

On the server the component that provides the LED controlling service must indicate that it is to be provided remotely.

```

module LEDControllerC {
    provides remote interface LEDControl;
}
implementation { ... }

```

The server’s configuration does not need to connect anything to the remote interface explicitly.

### 3.3 Callbacks and First-Class IDs

We assume that the local component IDs for “well known” services will be agreed upon ahead of time by a social process outside of our system. This is also how TCP port addresses for well known services on the Internet are handled. By broadcasting to a well known local component ID, a node can use services on neighboring nodes without necessarily knowing their node IDs.

If a node expects a reply from a service that it invokes, the calling node must set up a separate component with a suitable remote interface to receive the service’s result. In SpartanRPC remote invocations can only transmit information in one direction. Bidirectional data flow requires separate dynamic wires.

In this case the service would normally require the client to provide its component ID as arguments to the service invocation. The server could store these values for use by a server-side component manager. When the server transmits the reply back to the client, the server’s component manager would use the stored addressing information as appropriate.

For example assume that the LED controller on the server returns the old state of the LEDs whenever the LED value is changed. The server configuration would include an appropriate dynamic wire as follows

```

LEDControllerC.LEDResult ->
    [LEDControllerC].LEDResult;

```

The client must provide the `LEDResult` interface remotely to receive this result. In this example the `LEDControllerC` component is its own component manager. This makes it easy for the `elements` command to access global data that was recorded inside `LEDControllerC` when the service it provides was previously invoked. Creating a component that is its own component manager is a common idiom in SpartanRPC programs.

### 3.4 Implementation Limitations

Certain run time checks are needed to ensure the semantic integrity of the application. Since the invoking node can not know for sure that the component with a given ID actually provides the necessary interface, a check is made by the receiving node to verify that an appropriate interface is being used. If this check fails, the call is ignored. In the interest of minimizing radio traffic, no error indication is returned.

Also, there are restrictions on the fan-out size of a dynamic wiring. In our current implementation this limit depends on the number and size of the duty's parameters. The SpartanRPC header containing the list of target components must coexist with the duty arguments in a single 16 byte packet. For the case of a parameterless duty, the packet can hold at most seven component IDs. The format of SpartanRPC packets is discussed further in subsection 6.2.

## 4. SECURING RPC

Our primary goal is to provide convenient security primitives that application programmers can use when developing heterogeneous wireless sensor networks. The RPC services presented in the preceding sections provide an abstraction for building APIs presented by the network itself. Now, we focus on a means to protect these services via a language-based authorization mechanism, that will allow subnetworks in different protection domains to interact in a secure manner. This mechanism is a simple *capability-based* authorization system, that is sufficiently high-level to provide an abstraction for an underlying message authentication scheme, while being sufficiently low-level to serve as a basis for more complex protocols in applications.

### 4.1 Capability-Based Security

A capability is an unforgeable reference to a resource, the possession of which is necessary to gain resource access [18]. Although various systems implement capabilities in various manners, this principal is consistent. In SpartanRPC, resources are taken to be RPC services, and programmers may specify security policies associated with these services by requiring the activation of a capability for their usage. We envision that individual capabilities will usually be associated with statically assigned roles. When using a secure service, the user must activate the required capability before usage. As discussed later in this Section, AES MAC encryption underlies our capability-based mechanism, to ensure unforgeability of capabilities and efficiency in practice. But first, we discuss our language level capability-based mechanism as it is presented to the programmer.

### 4.2 Syntax and Semantics

To declare security policy, an RPC service provider can modify a remote interface provides declaration in components, with the syntax `requires C` where `C` is a string literal denoting a capability. For example:

```
module LEDControllerC {
    provides remote interface LEDControl
    requires "K";
}
implementation { ... }
```

This means that posting of any duty in `LEDControl` provided by this component requires activation of capability "K". All capability requirements are declared statically in this manner, and all capabilities are given statically, i.e. SpartanRPC does not support dynamic capability generation.

Notice that security policies are not declared for interfaces but rather for implementations of them. This is because a given interface might be provided by multiple components with different security needs, and the implementation of a remote interface truly constitutes a service.

In order to use a secured service, clients may activate a capability `C` when wiring to a protected component interface via the syntax `auth C`. For example, assuming that `RemoteSelectorC` is a

component manager for provider(s) of the protected `LEDControl` service described above:

```
configuration ClientAppC { }
implementation {
    ...
    auth "K" ClientC.LEDControl ->
    [RemoteSelectorC].LEDControl;
}
```

Any postings of duties from `LEDControl` in the `ClientC` component need not mention security at all—capabilities are activated at configuration wiring connections since that is where interface uses are reified with implementations (services).

### 4.3 Security Properties

The implementation of our security mechanism is described in more detail in section 6, but a summary is as follows. Capabilities are in one-to-one correspondance with AES symmetric keys. Activating a capability `C` at a wiring connection entails signing all messages associated with duty postings over that wire with a MAC under the key denoted by `C`. Any service provider will verify these MACs under the key denoted by the capability `C` required for the service. Since capabilities are known statically, we assume that keys are stored in ROM, so that a mote's capabilities are exactly the keys it is deployed with.

In this basic scheme, capabilities are supported by MAC authentication, with the same guarantees offered by AES authentication. Our system does not provide any form of replay protection out of the box, but this can be added at the application level. For example it is a simple matter for an application to pass a counter as an additional duty parameter. The server could verify that the count increases monotonically as a simple form of replay protection. Accordingly the `LEDControl` interface might be defined instead as

```
interface LEDControl {
    duty void setLeds
    (uint8_t value,
     int message_number,
     int sending_node);
}
```

The server records the highest message number received from each node and ignores at the application level any invocations involving messages numbers less than or equal to the highest seen so far. We feel that delegating replay protection to the application layer is appropriate since SpartanRPC is intended to be a low level infrastructure on which more complex systems can be built.

In addition our system does not currently offer any confidentiality service; duty arguments are transmitted in plaintext. However, extending our system to encrypt duty arguments is an area we intend to explore as future work.

### 4.4 Examples

Here we illustrate usage and features of our security mechanism by example. We observe that it is possible for a component to provide the same interface with different keys to support different classes of clients. For example, imagine that within the network certain nodes offer RPC services for setting a sensor sampling rate. Other nodes in that network are able to set a faster sampling rate than "public" nodes outside the network. This policy can be declared as follows:

```
module SensorControllerC {
    provides remote interface SensorControl
```

```

    as PublicSensor requires "public";
    provides remote interface SensorControl
    as PrivateSensor requires "private";
}
implementation {
    duty PrivateSensor.set_rate(int value)
    { ... }
    duty PublicSensor.set_rate(int value)
    { ... }
}

```

This scheme takes advantage of nesC's ability to define interface aliases in component specifications. Each alias must be given a unique interface ID.

Note that duties only transmit information in one direction. Results returned by a server to a client must be transmitted using a duty provided by the client. This organization makes it simple to apply a different security policy on returned results. For example, consider the case in which a `SensorControl` client sets up their own service `SensorResult` to receive sensor data reports. In order to eliminate false data from being reported by malicious nodes, the `SensorResult` service can be protected by a capability "sensor\_auth" that we assume is possessed only by nodes authorized to report sensor data:

```

module ClientC {
    uses interface SensorControl
    provides remote interface SensorResult
    requires "sensor_auth";
}
implementation {
    duty void SensorResult.report
    (uint8_t value)
    {
        // Posted by authentic server.
    }
    ...
    post SensorControl.read(5, NETWORK_ID);
}

```

This module posts a duty on the server but also provides a duty that the server can use to return sensor results. The network ID passed to the server's duty is intended to identify the network making the request so that the server can use the appropriate capability in its reply. The server module could contain:

```

module ServerC {
    provides remote interface SensorControl;
    uses interface SensorResult as Result1;
    uses interface SensorResult as Result2;
}
implementation {
    duty void SensorControl.read
    (uint8_t value, int network_id)
    {
        ...
        switch (network_id)
        case NETWORK_1:
            post Result1.report(reading);
            break;
        case NETWORK_2:
            post Result2.report(reading);
            break;
    }
}

```

Although no capability is needed to post the server duty, the client's network ID is used by the server to post an appropriately authenticated reply containing sensor data. The server's configuration might contain:

```

configuration ServerApp { }
implementation {
    auth "sense_auth1"
    ServerC.Result1 -> [Net1SelectorC]
    auth "sens_auth2"
    ServerC.Result2 -> [Net2SelectorC]
}

```

## 5. EXTENDED EXAMPLE

To illustrate the usefulness and practicality of our design we implemented a skeleton program that uses directed diffusion to gather temperature events in a heterogenous network. The directed diffusion algorithm requires that nodes communicate with a dynamically changing subset of neighbors. In addition, the algorithm requires the use of two distinct communication pathways. Nodes interested in receiving data communicate their interest forward over the network toward sensors. Nodes that observe the data communicate results backward toward the interested nodes. In our example we choose to protect these two pathways with different roles.

The dynamic wire mechanism of our system is a good match to the requirements of directed diffusion. It is straightforward for a component manager to compute the subset of neighbors currently needed in each communication. *Who* receives a communication is computed independently from *what* is communicated. In addition, our security primitives make it easy to authorize communication independently in two directions. In contrast other systems such as TinyRPC do not provide a convenient multicast facility, requiring the application programmer to attend to such details manually [13]. TinyRPC also provides no support for security at all.

### 5.1 Directed Diffusion

The directed diffusion algorithm [8] allows a node to express an interest in data of a certain kind. In our example interests are expressed as temperature thresholds. Any node that observes a temperature greater than the threshold is requested to report that data back to the interested node. Interests are associated with a certain data rate. Initially a node seeking temperature data floods the entire network using an interest with a low data rate. As data events find their way back to the interested node, that node selectively *reinforces* certain immediate neighbors by retransmitting the interest with a higher associated data rate to just those neighbors.

Each node maintains a cache of active interests. When a node observes or receives a data event it sends the data to all immediate neighbors that have expressed interest in it. Since not every neighbor is interested in all data, only a subset of neighbors are involved in each data transmission.

Each node also maintains a cache of data events that have been recently seen. This cache is used, in part, to measure the actual rate at which data is received from various neighbors. This information is made available to the reinforcement algorithm so that an appropriate decision can be made as to which nodes might be suitable to reinforce.

### 5.2 Interfaces

Interest propagation is handled by a remotable interface with a single duty:

```

interface InterestManagement {
    duty void set_interest(

```

```

uint16_t sender_node_id,
int      temp_threshold,
int      interval,
int      duration);
}

```

A node expresses interest in temperature data above a certain threshold by posting this duty on its neighboring nodes. The node transmits its ID to the neighbor so that data can later be returned to it. Nodes have no prior knowledge of the network topology.

The `interval` parameter is the desired interval in timer ticks between data transmissions. The `duration` parameter is the lifespan of the interest in timer ticks. Nodes should agree on a common timer period but they otherwise do not need to have synchronized clocks.

Data propagation is handled by a separate remotable interface with a single duty:

```

interface DataManagement {
  duty void set_data(
    uint16_t sender_node_id,
    uint16_t originator_node_id,
    int      temp_value)
}

```

Each data event is defined by a temperature value and the node ID where that temperature was observed. The neighbor node that sent the data event is typically different than the node where the event was originally generated. The sender node ID is necessary, however, so that the rate of data transmission coming from a particular neighbor can be measured.

### 5.3 Configuration

The interest and data caches are the two central components of our application: `InterestManagerC` and `DataManagerC`. The interest manager provides the `InterestManagement` interface remotely and uses the same interface on other components to propagate interests forward through the network. The data manager component provides and uses the `DataManagement` interface in a similar way to propagate data events backward toward the interest sources. Both components serve as their own component managers, using internal information to specify the destination nodes of each outgoing post operation.

The main configuration contains, in part, the following wiring for the interest manager:

```

auth "interest"
InterestManagerC.NeighborSensor ->
  [InterestManagerC].InterestManagement;

```

The `NeighborSensor` interface is an alias for the `InterestManagement` interface. When the local interest manager posts the `set_interest` duty, that duty is invoked in all neighbors currently selected by its own, internal component manager. These post operations are authorized using the `interest` role.

The wiring for the data manager is similar:

```

auth "data"
DataManagerC.NeighborSensor ->
  [DataManagerC].DataManagement;

```

In this case the `NeighborSensor` interface is an alias for the `DataManagement` interface. A different role is used to authorize data propagation to allow for more flexible deployment scenarios. For example, specialized temperature sensors that are authorized

to produce data might be forbidden from making interest declarations of their own. Such nodes would not be able to activate the interest role but could activate the data role when sending outgoing observations.

### 5.4 Interest Management

Interests are handled by an interest manager component with the partial specification as follows:

```

module InterestManagerC {
  provides interface ComponentManager;
  provides remote interface InterestManagement
    requires "interest";
  uses interface InterestManagement
    as NeighborSensor;
}

```

There are two significant functions in the interest manager component. A timer event handler simply scans the interest cache and removes interests that have expired. The `set_interest` duty, however, is more complicated. It is responsible for propagating incoming interest to other neighbors. If the data rate of an incoming interest is increased, it may be necessary to reinforce one or more neighbors in response.

Because the interest manager is its own component manager, setting up target node addresses entails updating an internal component set variable as appropriate. In the case when a new interest is received the interest manager wishes to propagate that interest to all neighbors. This is done with code as shown in Figure 3.

```

remote_set.ids = &remote_components;
remote_set.count = 1;
remote_components[0].node_id = 0xFFFF;
remote_components[0].local_id =
  INTERESTMANAGER_LOCALID;
post NeighborSensor.set_interest(NODE_ID,
  temp_threshold, interval, duration);

```

Figure 3: Propagating interests

The `remote_set` is a component set containing a pointer to an array of component IDs. In this case only a single component ID is needed to encode a SpartanRPC broadcast. The previously agreed upon local ID of the interest manager is used to specify which component on the neighbor nodes is to process the duty.

The interest manager is its own component manager and thus provides the `ComponentManager` interface. The implementation of the `elements` command in that interface is trivial in this case:

```

command
component_set ComponentManager.elements( )
{
  return remote_set;
}

```

Before the `post` operation in Figure 3 returns, this method is used to obtain a list of target components. The list is used at once to prepare the outgoing packet so that after the `post` operation is complete `remote_set` and `remote_components` can be reused without affecting any pending radio transmissions.

In the more complicated case where an interest is being reinforced, the interest manager must use information in the data cache

to compute which neighbors need reinforcing. Although SpartanRPC allows a component manager to dynamically select neighbor nodes, the component used as a component manager is statically bound. Thus in this example the interest manager can not switch its component manager to, for example, the data manager. To work around this, the interest manager calls a command in the data manager asking the data manager to populate its `remote_set` variable as shown in Figure 4.

```
remote_set.ids = &remote_components;

// Compute which neighbors need reinforcing
// and populate the remote_set variable.
call DCache.compute_reinforcement(
    &remote_set);

// Post the interest to those neighbors.
post NeighborSensor.set_interest(NODE_ID,
    temp_threshold, interval, duration);
```

Figure 4: Computing and posting interest reinforcements

The data manager has a dual structure where the implementation of the `set_data` duty simply adds the data event to the data cache, and the implementation of the timer fired event performs the complicated task of propagating data to interested nodes. The data manager handles remote nodes in the same general manner as described above.

## 6. IMPLEMENTATION

In this section we describe our current implementation of SpartanRPC. We have created a program we call *Sprocket* that acts as a SpartanRPC precompiler. It accepts a SpartanRPC enabled nesC program and outputs an ordinary nesC program that can then be processed by the standard nesC compiler. Sprocket targets nesC version 1.2 and the language it accepts is intended to be an extension of nesC version 1.2.

We focus here on describing the highlights of the implementation. In Sprocket, a duty posting is converted into a remote message send, containing an *identifier* associated with the posted duty so the receiver may dispatch the intended functionality. The RPC service provider runs a *skeleton* of any remotable interface, that receives these messages, interprets identifiers, and dispatches functionality appropriately. Dynamic wirings in RPC client programs are converted to statically wired *stubs* that implement dynamic wiring endpoints as message recipients: when a duty posting is converted into a message send by Sprocket, the component IDs in the dynamic wiring endpoints are integrated into the identifier in the message. To support security features, duty messages may also contain a MAC signed with the AES key associated with a particular capability; authentication of this MAC underlies SpartanRPC authorization.

### 6.1 Identifiers

A SpartanRPC identifier is a 4-tuple  $(N, L, I, D)$ . Here,  $N$  is the TinyOS ID of the node on which the duty is implemented; we assume that these are network-level unique.  $L$  is a local component ID, assigned to each component that provides a remotable interface; component IDs are node-level unique.  $I$  is an interface ID, required since a component may provide more than one remotable interface, even multiple instances of the same interface. Interface IDs must be shared across networks and therefore must be set by a

social process rather than a technical one. However, multiple interfaces can be given the same ID provided they are always provided by different components. Finally,  $D$  is a duty ID, which must be unique per (component, interface) pair.

In the current version of Sprocket, IDs are assigned statically by an arbitrary (automated and/or social) process, and we assume that Sprocket configuration files that define the association between IDs and the entities to which they refer are known to all interacting actors. More sophisticated techniques for defining and communicating RPC interface definitions between actors is an interesting topic for future work.

### 6.2 Data Packet Format

SpartanRPC packets contain a header with addressing information and marshalled duty arguments. In the current version of Sprocket the size of a Spartan RPC data packet is limited to 16 bytes (or 20 bytes when authentication is used). Figure 5 shows the packet format.

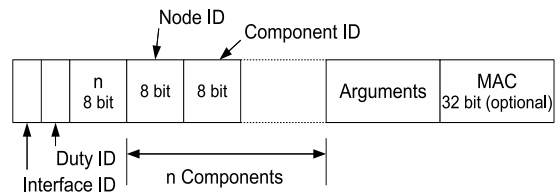


Figure 5: Spartan RPC data packet format.

The Spartan RPC packet header can introduce significant overhead in some cases. In the current version of Sprocket the interface ID and duty ID are packed as two four bit fields in a single byte. Also each intended destination is identified by a byte for the TinyOS node ID and a byte for the component ID. Finally an additional byte is used to encode the header's size. This yields a total overhead of  $2 + 2n$  bytes where  $n$  is the number of components intended to receive the packet. A special node ID of  $0xFF$  is used to represent a SpartanRPC level broadcast. Thus in the special (and common) case where all neighbor nodes are to process the remote call, the overhead is exactly four bytes.

The limited field sizes used in the header put static restrictions on the system. Only 16 remote interfaces per component can be used with at most 16 duties per interface. In addition, the current version of Sprocket limits the network to at most 255 nodes with 256 remotely accessible components per node. The limit on network size might be a problem in some cases. However, the limit on the number of remotely accessible components per node is probably more than is necessary. By simply adjusting the number of bits used for these fields (for example, 12 bits for the node ID and 4 bits for the local component ID) a different balance can be obtained without increasing the overall packet overhead.

### 6.3 Skeleton Generation

For each remote interface provided, Sprocket converts the duties in the component providing that interface into nesC commands. Sprocket also generates a skeleton component for every remote interface that handles incoming SpartanRPC duty data packets. These skeletons are connected to the active message components in the TinyOS library. Each time a duty message is received the skeleton checks the packet for applicability as follows.

1. If the packet is not intended for the interface supported by the skeleton, it is ignored.

2. If the duty ID in the packet is out of bounds for the interface, it is ignored.
3. If the packet does not contain a component ID that matches that supported by the skeleton, it is ignored.

If the packet is applicable, the skeleton unmarshalls the message, stores the duty arguments in skeleton-local variables, and posts a task that implements the duty. For each duty in the provided interface, Sprocket generates a trival task in the skeleton that simply calls the converted duty. For example:

```
// 'value' written when packet unmarshalled.
uint8_t value;
task void setLeds()
{
    call Blink.setLeds(value);
}
```

Thus the task-like semantics of duties are ultimately implemented in terms of ordinary nesC tasks.

## 6.4 Stub Generation

On the RPC client side, Sprocket converts each duty posting into a command in a special stub component generated by Sprocket. That command first calls the `elements` command in the component manager to obtain the list of target components. It then prepares a SpartanRPC data packet by marshalling the duty arguments. Finally it broadcasts the packet to all neighboring nodes using the TinyOS active message library. Recipients discern packets intended for them via packet identifiers as described above. For example, stub command supporting the `setLeds` duty is, in part, as follows:

```
command
void Blink.setLeds(uint8_t value) {
    struct component_set remote_components =
        call ComponentManager.elements();

    // Prepare SpartanRPC data packet using
    // the component set obtained above.

    // Send the packet using runtime function.
    send_buffer();
}
```

Sprocket converts dynamic wires into static wiring that connects the posting component to the automatically generated stubs. These stubs are connected to the component manager associated with the dynamic wiring. For example, a dynamic wire such as:

```
ClientC.LEDControl ->
    [RemoteSelectorC].LEDControl;
```

is converted converted into:

```
components Spkt__1;

ClientC.LEDControl -> Spkt__1;
Spkt__1.ComponentManager ->
    RemoteSelectorC;
Spkt__1.Packet -> AMSenderC;
Spkt__1.AMPacket -> AMSenderC;
Spkt__1.AMControl -> ActiveMessageC;
Spkt__1.AMSend -> AMSenderC;
```

Here the `Spkt__1` component is the Sprocket generated stub that handles the remote procedure call.

## 6.5 Security

When capability-based security is used for secure RPC invocation, Sprocket consults a global configuration file that maps capabilities to keys. To activate a capability over a dynamic wire, the Sprocket generated stub computes a message authentication code (MAC) that covers the SpartanRPC header and marshalled duty arguments. In the current implementation this MAC is computed using the AES encryption algorithm in CBC mode with an initialization vector of zero. Because SpartanRPC packets are currently limited to 16 bytes, only a single AES encryption is necessary to compute the MAC. The first four bytes of the resulting cipher text is used as the MAC value. While a MAC of only 32 bits would not normally be considered secure, wireless sensor networks generate and use data so slowly that attacking even such a short MAC is not considered feasible [9, 12].

For components providing a secure remote interface, the generated skeleton incorporates a MAC authentication procedure under the required key as declared in the component specification. The usual checks of interface ID and component ID are done first so as to avoid a costly MAC computation in the case where the received packet is not actually intended for the skeleton. Only when the other applicability checks succeed is the MAC checked. The duty invocation is ignored if the MAC check fails.

## 7. EMPIRICAL RESULTS

The sensor nodes that are the target of our system are highly constrained devices with limited memory, CPU resources, and electrical power. Conserving the resources of the platform is a matter of high importance. Our system consumes additional resources because of the overhead needed to manage remote procedure calls and cryptographic operations.

### 7.1 Test Programs

To explore the effect of these overheads on real system performance we conducted several experiments. Our test devices were two Tmote Sky wireless sensor nodes [15]. These devices are based on the Texas Instruments MSP430F1611 microcontroller [19] with 48 KiB of flash ROM, 10 KiB of static RAM, and running at a clock frequency of 8 MHz. For wireless communication each node uses a 2.4 GHz IEEE 802.15.4 Chipcon CC2420 [3] low power transceiver. The system software we used was TinyOS version 2.1.0.

The client node executed a program that periodically invoked a service on the server node, passing that service an eight bit value. The server used the least significant three bits of that value to control the red, green, and blue LEDs on the server mote. Several pairs of test programs were written that all performed the same essential function but in progressively more abstract ways.

**Baseline** These programs do not use any of our extensions. Instead all radio handling is done explicitly by the application and no cryptography is used. These programs form a point of comparison; all overheads are measured relative to this baseline.

**Duties** These programs use dynamic wires and duties for RPC support. They are used to measure the overheads due to our RPC mechanism. However, these programs provide no security services.

**Security** These programs use security authorizations in addition to dynamic wires and duties. They represent the fully enabled features of our system and have the maximum overhead.

**Table 1: Memory consumption of test programs.**

	ROM		RAM	
	Bytes	%	Bytes	%
Baseline Client	13096	—	378	—
Baseline Server	12576	—	306	—
Duties Client	13568	3.6	398	5.3
Duties Server	12624	0.4	308	0.6
Security Client	22662	73	608	61
Security Server	21978	75	534	74

We felt it was desirable to execute these tests in a realistic context. In many wireless sensor applications the most energy is consumed by the radio. To reduce this energy it is often desirable to use *low power listening*. In this mode of operation the receiver runs with the radio off most of the time, waking the radio up periodically to see if any remote devices are transmitting on the channel [14]. TinyOS supports this mode of operation directly for the CC2420 radio.

Since our system hides the radio handling from the application, the use of low power listening becomes a matter for Sprocket to consider. Our implementation assumes that low power listening will be used in all cases and writes the stubs and skeletons accordingly. Ultimately it would be desirable for Sprocket to be configurable to allow different radio handling policies to be specified easily. This would give users the option to control radio management while still keeping the details of radio communication out of the application code.

## 7.2 Memory Consumption

Table 1 shows the overall memory consumption, as reported by the nesC compiler, of each test program. All values shown in the table are in bytes.

The dramatic increase in memory consumption of the security enabled programs is largely a consequence of the pure software AES implementation used for the cryptographic operations [4]. Although we only support AES encryption (decryption is not necessary for MAC computation and verification) about half of that ROM increase is due to code with the other half due to fixed lookup tables used by that code. The RAM increase is largely due to the buffer space required for encryption and may be subject to further optimization. We note that most of the extra code and data space can be reused for each security enabled remote invocation. Support for security on additional dynamic wires or remote interfaces requires only about the same amount of overhead as is required for the plain duty case.

We also note that the use of low power listening increased the memory requirements modestly over the case when low power listening was not used. ROM requirements increased by about 1750 bytes, and RAM increased by about 55 bytes. The numbers in Table 1 are for the low power listening case.

## 7.3 Energy Consumption

We also evaluated the energy consumption of the three programs. This was done by placing a 14.9  $\Omega$  current sensing resistor in series with the 3.0 V power supply and observing the power supply current waveform on an oscilloscope. In our experiments the maximum current observed was 18.5 mA which caused the mote power supply voltage to drop to a minimum of 2.7 V; a value that was still well within its operating range. However, this drop was considered in the energy values reported in Table 2.

Sprocket currently uses a radio sleep interval of 10 ms with a

**Table 2: Energy consumption of client programs.**

	TX Energy ( $\mu$ J)	Compute Energy ( $\mu$ J)
Baseline Client	780	17
Duties Client	780	17
Security Client	780	22

50% duty cycle. That is, the receiver alternates between watching the channel for 10 ms and sleeping for 10 ms. The sender must transmit on the channel long enough to be sure that the receiver will hear it regardless of the amount of data being sent. When not actively transmitting Sprocket turns the transmitter radio off.

In our experiments the transmitter pulse width varied between 15 and 16 ms, and was the same for all three programs. Just before the transmitter pulse started, a small increase in power supply current was observed lasting for 3 ms (in the baseline and duties only case) to 4 ms (in the security enabled case). We assume this increase is due to activities on the node that are done just prior to enabling the radio for transmission. The fact that the security enabled case was active for more time prior to transmission is most likely due to the extra time required for MAC computation.

The “compute burst” power supply current waveform had a complex shape and was close to the noise floor but from that waveform we estimated the compute energy consumed by the mote for all three of our programs. The results are summarized in Table 2.

Note that despite our term “compute burst,” it is likely that some of this energy is actually being used to power up various supporting components related to the upcoming transmitter pulse. For example the CC2420 radio requires that its voltage regulator and oscillator be turned on and allowed to stabilize for a time before signals can actually be transmitted [3]. These details are handled by TinyOS, but the power supply current needed for them is part of our observations. The MSP430 microcontroller has exceptionally low power requirements. Even when fully active the microcontroller consumes a maximum current of only about 500  $\mu$ A [19]. In our environment a 500  $\mu$ A current burst of 4 ms corresponds to just 6  $\mu$ J of energy which is clearly a minority of the observed energy.

## 8. CONCLUSION

We have extended nesC with a light weight, link-layer, secure RPC facility, yielding a language called SpartanRPC. SpartanRPC is a middleware technology supporting secure WSN applications comprising multiple protection domains. It is ideal for settings in which multiple subnetworks administered by distinct social entities cooperate to obtain a holistic behavior. A language-level capability-based authorization mechanism provides applications programmers with an easy and effective means for specifying and enforcing security policies.

Because of the long delays and unreliability inherent in radio communication, SpartanRPC treats remote execution of RPC services as fundamentally asynchronous. Inspired by existing nesC practice SpartanRPC provide task-like units of remote execution called *duties*. In addition SpartanRPC extends nesC configurations to allow components on different nodes to be wired together in a dynamic manner, i.e. remote wirings to RPC services can change during program execution. This accommodates typical routing and programming patterns in WSN applications.

We have implemented SpartanRPC in the Sprocket framework, wherein RPC features are transformed at compile into standard nesC code, and symmetric key cryptography and MACs underlies the authorization mechanism. Empirical results suggest that SpartanRPC as implemented in Sprocket is efficient and realistic for

programming practice. We have illustrated the facility of the language itself in multiple code examples, including an implementation of secure directed diffusion in a heterogeneous trust environment.

## 9. REFERENCES

- [1] G. Bertoni, L. Breveglieri, and M. Venturi. ECC hardware coprocessors for 8-bit systems and power consumption considerations. *itng*, 00:573–574, 2006.
- [2] S. A. Çamtepe and B. Yener. Key distribution mechanisms for wireless sensor networks: a survey. Technical Report TR-05-07, Rensselaer Polytechnic Institute, 2005.
- [3] Chipcon. CC2420 2.4 GHz IEEE 802.15.4 / zigbee-ready RF transceiver. Preliminary datasheet rev 1.2, June 2004.
- [4] P. J. Erdelsky. Rijndael encryption algorithm. <http://www.efgh.com/software/rijndael.htm>, September 2002. Accessed September 2009.
- [5] T. Gao, C. Pesto, L. Selavo, Y. Chen, J. G. Ko, J. H. Lim, A. Terzis, A. Watt, J. Jeng, B.-R. Chen, K. Lorincz, and M. Welsh. Wireless medical sensor networks in emergency response: Implementation and pilot results. In *Technologies for Homeland Security, 2008 IEEE Conference on*, pages 187–192, 2008.
- [6] V. Gupta, M. Millard, S. Fung, Y. Zhu, N. Gura, H. Eberle, and S. C. Shantz. Sizzle: A standards-based end-to-end security architecture for the embedded internet (best paper). In *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 247–256, Washington, DC, USA, 2005. IEEE Computer Society.
- [7] W. Hu, P. Corke, W. C. Shih, and L. Overs. secFleck: A public key technology platform for wireless sensor networks. In *EWSN '09: Proceedings of the 6th European Conference on Wireless Sensor Networks*, pages 296–311, Berlin, Heidelberg, 2009. Springer-Verlag.
- [8] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1), February 2003.
- [9] C. Karlof, N. Sastry, and D. Wagner. TinySec: a link layer security architecture for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 162–175, New York, NY, USA, 2004. ACM.
- [10] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. *Elsevier's AdHoc Networks Journal, Special Issue on Sensor Network Applications and Protocols*, 1(2–3):293–315, September 2003.
- [11] K. Lorincz, D. J. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing*, 3(4):16–23, 2004.
- [12] M. Luk, G. Mezzour, A. Perrig, and V. Gligor. MiniSec: a secure sensor network communication architecture. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 479–488, New York, NY, USA, 2007. ACM.
- [13] T. D. May, S. H. Dunning, G. A. Dowding, and J. O. Hallstrom. An RPC design for wireless sensor networks. *Journal of Pervasive Computing and Communications*, 1(1), March 2006.
- [14] D. Moss, J. Hui, and K. Klues. TEP-105: Low power listening. <http://www.tinyos.net/tinyos-2.x/doc/html/tep105.html>. Accessed September 2009.
- [15] moteiv. Tmote sky low power wireless sensor module. Datasheet, November 2006.
- [16] M. Perillo and W. Heinzelman. *Fundamental Algorithms and Protocols for Wireless and Mobile Networks*, chapter Wireless Sensor Network Protocols. CRC Hall, 2005. To appear.
- [17] A. Perrig, J. Stankovic, and D. Wagner. Security in wireless sensor networks. *Commun. ACM*, 47(6):53–57, 2004.
- [18] J. S. Shapiro and S. Weber. Verifying the eras confinement mechanism. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 166, Washington, DC, USA, 2000. IEEE Computer Society.
- [19] Texas Instruments. MSP430F1611. Revised Datasheet, May 2009.
- [20] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM.