SpartanRPC: Secure WSN Middleware for Cooperating Domains

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

Christian Skalka
The University of Vermont
Burlington, Vermont
Email: 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.**

***Keywords*-remote procedure call, capability-based security, wireless sensor networks;**

I. Introduction

As WSN technology becomes more ubiquitous applications of overlapping yet independently controlled sensor networks will begin to appear. Networks from cooperating but distinct security domains may wish to use each other's nodes to increase the resolution, spacial coverage, or lifespan of certain sensing or control functions. For example, assume that two distinct networks are deployed to the same space such that nodes from both networks can communicate with each other. While the primary purpose of the two networks might be independent, their administrators might nevertheless agree to collaborate on certain supporting functionality such as data collection and analysis.

Although in some cases data collected by each security domain could be shared off-network, perhaps via Internet connected gateways, certain advantages could be gained by enabling in-network interactions. For example if one network is partitioned due to a failed node, the separated partitions might be able to continue communicating by routing their messages over a cooperating network's nodes.

In situations where low latency is important, such as with time synchronization protocols [1], in-network interactions between independent, cooperating WSNs would be more effective than attempting to synchronize two networks via long distance Internet links. Tracking applications [2] could also benefit from in-network interactions between cooperating networks. Handing off tracking information from one network to another via an Internet link would require activating a potentially large number of nodes in each network in order to send information to their respective gateways. This creates power consumption concerns. Other potential applications where in-network interactions could play a role include secure routing protocols in heterogeneous trust environments [3], transport and network layer protocols [4], and even mote-based web servers supporting secure channels [5].

High level applications can also benefit from in-network interactions between cooperating WSNs. For example, the use of WSNs to facilitate emergency care in disaster situations has been described [6], [7]. While this previous work has focused on WSNs in a single security domain, it is likely that multiple domains would be in use during many emergencies as first responders from different organizations or different political jurisdictions work together. In those cases it is not realistic to assume that each domain would have Internet connectivity; the disaster site might be too remote or the supporting network infrastructure might be non-functional. Direct interaction between the WSNs of cooperating domains becomes an effective way for them to share information.

In this paper we describe novel middleware technology to support WSN applications in this setting. Our system, which we call SpartanRPC, provides a new form of link-layer RPC as a natural extension of the nesC programming model, and *language based authorization* based on symmetric-key cryptography. As other authors have observed [8], 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.

A. 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 [9]. 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 [10]. SpartanRPC differs from these systems in that it extends the nesC programming language to allow programmer 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 [8], 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. Finally we created an implementation of SpartanRPC [11] from which we obtained empirical results showing that SpartanRPC 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 communication details 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.

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 the command to complete and yield a result. In contrast, when a *task* is posted it is placed in a queue for execution at some future time, and the calling context continues immediately. We 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 [8]. 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 an ordinary 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 Sect. II.

Dynamic Wires: The SpartanRPC design provides a 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, wirings to remote interfaces must specify the host of the wired-to component, and our 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. Dynamic wires are discussed more in Sect. III.

Fine-Grained Authorization Policies: Research on WSN security has addressed secure routing [3], link layer security [12], cryptography [13] and key distribution [14], and hardware issues [15]. This previous work has established a strong low-level foundation for security in WSNs. We believe that secure link-layer RPC is an appealing middleware solution in WSNs because 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. Our security mechanism is discussed in more detail in Sect. IV.

II. Duties and Remotability

Because of the slow, unreliable nature of wireless communications we believe it is unrealistic for RPC services in WSNs to be synchronous. Instead we believe that the semantics of tasks are closer to being a correct abstraction. They are not quite right however, as RPC services will typically require arguments to be passed, and while the poster of a task defines it, an RPC service invokes remotely defined functionality. We therefore define a new RPC abstraction called a *duty*.

A. Syntax and Semantics

Duties are declared in interfaces and syntactically resemble command declarations. Instead of using the reserved word `command` the new reserved word `duty` is used. Duties are allowed to take parameters (with restrictions as discussed below) but must return the type `void`. For example the following interface describes an RPC service for remotely flashing a mote LED:

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

Duties are defined in modules in a manner similar to the way tasks, commands, or events are defined. The reserved word `duty` is again used 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 Sect. II-B. 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 ctl)
    { ... }
}
```

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 {
    ...
    post LEDControl.setLeds(42);
}
```

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. Like a normal command invocation, configuration wirings determine where duty control flows. However, in SpartanRPC duty invocation control may flow to a component residing on a different network node. The invoking module must be connected to the remote modules by way of a dynamic wire as described in Sect. III.

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. Also, due to the unreliability of the network a posted duty may not run at all.

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 Sect. VI, 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.

B. *Remotable Interfaces*

We impose certain requirements on RPC service definitions for ease of implementation. First, since WSN nodes do not share state we 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 duties taking only 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.

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

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

interface ComponentManager {
    command component_set elements();
}
```

Figure 1. Component Manager Interface and Type Definitions

III. 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. In this section we discuss the facility in SpartanRPC to allow *dynamic wirings* for control flow from duty invocation via remotable interfaces to duty implementation, wherein the programmer has control over wiring endpoints and how they may change during program execution.

A. *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 Fig. 1. The `node_id` member is the same node ID used by TinyOS and is set when the node is programmed during deployment. The local ID member 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*.

A *component manager* is a component that provides the `ComponentManager` interface defined in Fig. 1. It dynamically specifies a set of component IDs that ultimately serve as dynamic wiring endpoints.

As a simple example, consider the component manager `RemoteSelectorC` as shown below:

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

  command component_set
    ComponentManager.elements() {
        return result;
    }
}
```

This component manager always returns a component set containing a single component. The special SpartanRPC broadcast node 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 example. In a more complex example the component manager would 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.

B. 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 follows:

```
  connection ::=
      endpoint ’->’ dynamic_endpoint

  dynamic_endpoint ::=
      ’[’ IDENTIFIER ’]’ (’.’ IDENTIFIER)*
```

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.

For example, 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:

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

On the server the component that provides the LED controlling service must indicate that it is to be provided remotely as shown in Sect. II-A. The server’s configuration does not need to connect anything to the remote interface explicitly.

C. 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. 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 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 an argument to the service invocation. The server could store that value for use by a server-side component manager.

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. This is a common SpartanRPC idiom.

IV. 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 network services. Now, we focus on a means to protect these services via a language-based authorization mechanism. This mechanism is a simple *capability-based* authorization system, that is sufficiently high-level to hide the underlying message authentication scheme, while being sufficiently low-level to serve as a basis for more complex protocols in applications.

A. Capability-Based Security

A capability is an unforgeable reference to a resource, the possession of which is necessary to gain resource access [16]. 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 be associated with statically assigned roles.

B. Syntax and Semantics

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

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

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. This is reasonable because capabilities are associated with network services. Thus the number of needed capabilities scales with the number of interacting security domains and not with the total number of nodes.

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:

```
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.

C. Security Properties

The implementation of our security mechanism is described in more detail in Sect. VI, but a summary is as follows. Capabilities are in one-to-one correspondence 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 using 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 node's capabilities are exactly the keys it is deployed with.

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 an application could pass a counter as an additional duty parameter. The server could verify that the count increases monotonically as a simple form of replay protection.

We feel that delegating replay protection to the application is appropriate since SpartanRPC is intended to be a low level infrastructure on which more complex systems can be built. Not all applications will need replay protection and it is our desire to keep the core overhead minimal.

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

V. Example

To illustrate the usefulness of our design we implemented a skeleton program that uses directed diffusion to gather temperature events in a heterogeneous network. The directed diffusion algorithm requires that nodes communicate with a dynamically changing subset of neighbors. The dynamic wire mechanism of our system makes it 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, 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 pathways with multiple capabilities.

A. Directed Diffusion in Multiple Domains

The directed diffusion algorithm [17] is an approach for diffusing data across a wireless sensor network. The algorithm 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. A certain data rate, expressed as a time interval between transmissions, is associated with each interest. Initially a node seeking temperature data floods the 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 direct or indirect 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.

B. Interfaces

Interest and data event propagation are handled by separate interfaces, as shown in Fig. 2, each containing a single duty.

```
interface InterestManagement {
    duty void set_interest(
        uint16_t sender_node_id,
        int      temp_threshold,
        int      interval,
        int      duration);
}

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

Figure 2. Directed Diffusion Interfaces

A node expresses interest in temperature data above a certain threshold and at a certain data rate by posting the `set_interest` duty on its neighboring nodes. Similarly a node passes data to its interested neighbors by posting the `set_data` duty.

C. *Configuration*

The interest and data caches, which we call "managers," are the two central components of our application. The interest manager provides the `InterestManagement` interface remotely and uses the same interface on other components. The data manager provides and uses the `DataManagement` interface in a similar way. Both components serve as their own component managers, using internal information to specify the destination nodes of each outgoing post operation.

In our example interest propagation is be controlled by two capabilities. The shared `ext_interest` capability allows a node from any protection domain to request a low data rate from nodes in any other domain. The `int_interest` capability is defined internally and independently by each protection domain, and allows a node in the same domain to request a high data rate.

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

```
auth "ext_interest"
InterestManagerC.NeighborSensors ->
    [InterestManagerC].InterestManagement;
auth "int_interest"
InterestManagerC.NeighborSensors ->
    [InterestManagerC].InteresttManagement;
```

Because the interest manager provides and uses the same interface, it defines `NeighborSensors` as an alias for the `InterestManagement` interface that it uses remotely. When the 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 both interest capabilities; neighbors can be in multiple protection domains. In this example no attempt is made to track which neighbors are in which domains. As a result two messages are sent to each neighbor selected by the interest manager, but this could be improved by using separate component managers for the internal and external domains.

D. *Interest Management*

The interest manager has a partial specification as follows:

```
module InterestManagerC {
    provides interface ComponentManager;
    provides remote interface
        InterestManagement as ExtManagement
            requires "ext_interest";
    provides remote interface
        InterestManagement as IntManagement
            requires "int_interest";
    uses interface InterestManagement
        as NeighborSensors;
}
```

The `set_interest` duty is provided for both internal and external post operations. The implementation is essentially the same. However, the duty used by other protection domains ignores requests for data rates that are too high.

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 propagates that interest to all neighbors. This is done inside the interest manager's `set_interest` duty with the following code:

```
remote_set.ids   = &remote_components;
remote_set.count = 1;
remote_components[0].node_id  = 0xFFFF;
remote_components[0].local_id = INTEREST_ID;
post NeighborSensors.set_interest( ... );
```

The "well known" local ID of the interest manager is used to specify which component on the neighbor nodes is to process the duty. The implementation of the `elements` command in the `ComponentManager` interface merely returns `remote_set` computed above. Before the posting of `set_interest` returns, `remote_set` is used to prepare the outgoing packet. After the post 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 communicates with the data manager using connections not shown here. With the data manager's help the interest manager computes appropriate neighbors dynamically before posting `set_interest` on those neighbors.

E. Data Management

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 a timer fired event performs the task of propagating data to interested nodes. The data manager manipulates the timer frequency to match the highest required data rate. However since not all data needs to be sent to all neighbors at such a high rate, only a dynamically changing subset of neighbors is selected for each timer event. This is done by adjusting an internal `component_set` before posting the `set_data` duty.

We further assume that nodes will only want to accept data events from authorized producers. All legitimate posts of `set_data` must be done using the `data` capability. The main configuration thus also contains dynamic wires such as:

```
auth "data"
DataManagerC.NeighborSensors ->
    [DataManagerC].DataManagement;
```

The specification of the data manager is, in part:

```
module DataManagerC {
    provides interface ComponentManager;
    provides interface DataManagement
        requires "data";
    uses interface DataManagement
        as NeighborSensors;
}
```

Notice that there are no security related artifacts in the body of the data manager's implementation.

VI. Implementation

In this section we describe our implementation of SpartanRPC. We have created a program we call *Sprocket* [11] that accepts a SpartanRPC enabled nesC program and outputs an ordinary nesC program.

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*. When a duty posting is converted into a message send by Sprocket, the component IDs in the dynamic wiring endpoints are integrated into the message. To support security features, duty messages may also contain a MAC computed with an AES key associated with a particular capability; authentication of this MAC underlies SpartanRPC authorization.

A. Identifiers

A SpartanRPC identifier is a 4-tuple (N, C, 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. C is a 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. Finally, D is a duty ID, which must be interface-level unique.

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.

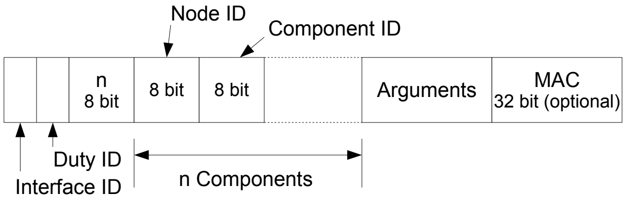

Figure 3. SpartanRPC data packet format.

B. Data Packet Format

SpartanRPC packets contain a header with addressing information and marshalled duty arguments. The total size of a SpartanRPC data packet is limited to 16 bytes in the current version of Sprocket. (or 20 bytes when authentication is used). Fig. 3 shows the packet format.

The SpartanRPC packet header can introduce significant overhead in some cases. In the current version of Sprocket, I and D are packed as two four bit fields in a single byte. Each intended destination is identified by a byte for N and a byte for C. 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, leaving 12 bytes for duty parameters. If a parameterless duty is called, the maximum fan out supported by our implementation is seven.

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.

C. 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 implementation. 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. If the packet is not intended for the (N, C, I) triple supported by the skeleton or if D is out of bounds for the interface, the packet is ignored. In the interest of minimizing radio traffic, no error indication is returned.

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 trivial 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.

D. Stub Generation

On the RPC client side, Sprocket converts each duty posting into a command in a 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.

Sprocket converts dynamic wires into static wiring that connects the posting component to the generated stub. The stub is 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 the configuration as follows:

```
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;
```

The `Spkt__1` component is the Sprocket generated stub.

E. Security

When capability-based security is used, Sprocket consults a configuration file that maps capabilities to keys. To activate a capability over a dynamic wire, the Sprocket generated stub computes a 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 data so slowly that attacking even such a short MAC is not considered feasible [12], [18]. Our MAC computation is simplistic, but we feel it is adequate to demonstrate a proof of concept.

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.

VII. 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.

A. Test Programs

To explore the performance of our system we conducted several experiments. Our test devices were two Tmote Sky wireless sensor nodes [19]. These devices are based on the Texas Instruments MSP430F1611 microcontroller [20] 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 [21] low power transceiver. The system software we used was TinyOS version 2.1.0 [22].

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 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. The "baseline" programs did not use any of our extensions. All radio handling was done explicitly and no cryptography was used. The "duties" programs used dynamic wires and duties for RPC support. The "security"

Table I
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

programs added the authorization support and exercised the full functionality of our system.

In a realistic context the average energy consumption of the radio can often be reduced by using *low power listening* [23]. In this mode the receiver runs with the radio off for significant periods of time, waking the radio up periodically to see if any remote devices are transmitting on the channel. In addition the transmitter broadcasts for a certain amount of extra time to ensure that its transmission will properly overlap with at least one receiver cycle. We used this mode in our test programs. As a result the extra transmission and reception energy required for each data packet completely overshadowed overhead due to the SpartanRPC extensions.

Since our system hides the radio handling from the application, the use of low power listening becomes an issue for Sprocket to handle. Our implementation assumes that low power listening will be used in all cases and writes the stubs and skeletons accordingly. Sprocket currently uses a radio sleep interval of 10 ms with a 50% duty cycle. When not actively transmitting Sprocket turns the transmitter radio off.

B. Memory Consumption

Table I shows the overall memory consumption, as reported by the nesC compiler, of each test program.

The dramatic increase in memory consumption of the security enabled programs is a consequence of the pure software AES implementation used for the cryptographic operations [24]. The RAM increase is largely due to the buffer space required for encryption and may be subject to further optimization. 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.

C. Energy Consumption

We also evaluated the energy consumption of the three client programs. This was done by placing a 14.9 Ω 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 transmitter pulse width varied between 15 and 16 ms, and was the same for all three programs. The energy consumed during transmission was 780 μJ. 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. This *compute burst* consumed 17 μJ of energy in the first two programs and 22 μJ of energy in the security enabled case. Presumably the additional energy used in the security case is at least partially due to the cryptographic computations.

Despite our term “compute burst,” it is likely that some of this energy was 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 [21]. For example, Even when fully active the microcontroller draws a maximum current of only about 500 μA [20]. In our environment a 500 μA current burst of 4 ms corresponds to just 6 μJ of energy which is clearly a minority of the observed energy.

VIII. 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 application 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 [11], 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 with an implementation of secure directed diffusion in a heterogeneous trust environment.

References

[1] S. Ganeriwal, R. Kumar, and M. B. Srivastava, “Timing-sync protocol for sensor networks,” in *SenSys ’03: Proceedings of the 1st international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2003, pp. 138–149.

[2] R. R. Brooks, P. Ramanathan, and A. M. Sayeed, “Distributed target classification and tracking in sensor networks,” in *Proceedings of the IEEE*, vol. 91, no. 8. IEEE, August 2003, pp. 1163–1171.

[3] 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*, vol. 1, no. 2–3, pp. 293–315, September 2003.

[4] M. Perillo and W. Heinzelman, *Fundamental Algorithms and Protocols for Wireless and Mobile Networks*. CRC Hall, 2005, ch. Wireless Sensor Network Protocols, to appear.

[5] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 247–256.

[6] 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*, 2008, pp. 187–192.

[7] 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*, vol. 3, no. 4, pp. 16–23, 2004.

[8] 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*, vol. 1, no. 1, March 2006.

[9] 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*. New York, NY, USA: ACM, 2006, pp. 416–423.

[10] 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*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 296–311.

[11] P. Chapin, “Sprocket home page,” accessed April 2010. [Online]. Available: http://www.cs.uvm.edu/%7Epchapin/Sprocket/

[12] 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*. New York, NY, USA: ACM, 2004, pp. 162–175.

[13] G. Bertoni, L. Breveglieri, and M. Venturi, “ECC hardware coprocessors for 8-bit systems and power consumption considerations,” *itng*, vol. 00, pp. 573–574, 2006.

[14] S. A. Çamtepe and B. Yener, “Key distribution mechanisms for wireless sensor networks: a survey,” Rensselaer Polytechnic Institute, Tech. Rep. TR-05-07, 2005.

[15] A. Perrig, J. Stankovic, and D. Wagner, “Security in wireless sensor networks,” *Commun. ACM*, vol. 47, no. 6, pp. 53–57, 2004.

[16] J. S. Shapiro and S. Weber, “Verifying the eros confinement mechanism,” in *SP ’00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2000, p. 166.

[17] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, “Directed diffusion for wireless sensor networking,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, February 2003.

[18] 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*. New York, NY, USA: ACM, 2007, pp. 479–488.

[19] moteiv, “Tmote sky low power wireless sensor module,” Datasheet, November 2006.

[20] Texas Instruments, “MSP430F1611,” Revised Datasheet, May 2009.

[21] Chipcon, “CC2420 2.4 GHz IEEE 802.15.4 / zigbee-ready RF transceiver,” Preliminary datasheet rev 1.2, June 2004.

[22] “TinyOS community forum,” accessed April 2010. [Online]. Available: http://www.tinyos.net/

[23] D. Moss, J. Hui, and K. Klues, “TEP-105: Low power listening,” accessed April 2010. [Online]. Available: http://www.tinyos.net/tinyos-2.x/doc/html/tep105.html

[24] P. J. Erdelsky, “Rijndael encryption algorithm,” September 2002, accessed April 2010. [Online]. Available: http://www.efgh.com/software/rijndael.htm