
PETER CHAPIN, University of Vermont
CHRISTIAN SKALKA, University of Vermont

We describe SpartanRPC, a secure middleware technology that supports cooperation between distinct protection domains in wireless sensor networks. SpartanRPC extends nesC to provide a link-layer remote procedure call (RPC) mechanism, along with an enhancement of configuration wirings that allow specification of remote, dynamic endpoints. RPC invocation is secured via an authorization logic that enables servers to specify access policies, and requires clients to prove authorization. This mechanism is implemented using a combination of symmetric and public key cryptography. We illustrate the usefulness of SpartanRPC through an extended example, and report on benchmark testing of a prototype implementation.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Compilers; Run-time environments; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems—Real-time and embedded systems; C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/server

General Terms: Security, Languages
Additional Key Words and Phrases: sensor networks, remote procedure call, trust management

ACM Reference Format:
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

SpartanRPC is a new programming language that supports application development in a decentralized, open world security model for wireless sensor networks (WSNs). Traditional networks have long enjoyed support for an open world security model via public key based security architectures such as the secure sockets layer (SSL) and the simple distributed security infrastructure (SDSI). The goal of our work is to introduce an open world security model to the TinyOS programming environment for embedded device programming. SpartanRPC is a dialect of nesC [Gay et al. 2003].

Currently, TinyOS security models are very simple and support only a closed world paradigm. TinySec [Karlov et al. 2004] and MiniSec [Luk et al. 2007] are based on shared secrets and generally assume that an entire network comprises a single security domain. Furthermore, these systems support confidentiality and integrity properties, but not access control, aka authorization. We have designed and implemented an extension to TinyOS programming model [Gay et al. 2003] called SpartanRPC, that in-
includes primitive features for specifying and enforcing authorization policies and allows multiple security domains to interact within a single network. SpartanRPC security mechanisms leverage public key cryptography and an authorization logic to support an open world model where shared secrets are not required a priori.

1.1. Overview and Applications

The SpartanRPC system provides an applications programming interface for managing resource access control in a WSN. It allows network administrators to define security policies that mediate access to specified resources on network nodes, and allows subnetworks with different security credentials to interact.

Consider a first-responder situation, in which multiple social entities must interact and cooperate on an ad-hoc basis. Recent work has shown the effectiveness of WSNs in such scenarios [Gao et al. 2008; Lorincz et al. 2004], in their ability to coordinate multiple data collection and communication devices in an ad-hoc, easily deployable manner. However, data collection and communication in this scenario (and other similar ones) must be a secured resource, due to e.g. HIPA requirements in the case of medical response. Furthermore, security must be coordinated on-site in a WSN comprising subnetworks administered separately (police, medical units from different hospitals, etc.), and no prior coordination between administrations can generally be assumed.

The SpartanRPC system is designed to address these types of scenarios.

For example, data communication can be treated as a secure resource by setting data access policies for individual nodes. WSN data communication protocols typically implement a publish/subscribe semantics, whereby users of data “subscribe” to data produced by “publishers”. Directed diffusion [Intanagonwiwat et al. 2003] is one such protocol, where network nodes express interest in certain types of data to neighboring nodes, which is subsequently reported to them when it is produced (published). In Sec. 5 we show how authorization policies can be assigned to interest update facilities on network nodes, which in effect imposes an access policy on data subscription in the network. Thus, in a first-responder scenario, if an EMT team emplaces a WSN to monitor patient locations and vital signs, a security policy can be imposed whereby responding police departments can emplace their own WSN, and through it access patient identity and location data but not medical data directly from the EMT network. This direct data access will often be necessary due to real-time constraints and lack of Internet connectivity in emergency situations.

Time synchronization is another important WSN function that is security sensitive, since many higher-level protocols rely on it. A number of previous authors have considered secure time synchronization in the presence of “insider” attacks [Manzo et al. 2005; Ganeriwal et al. 2008], whereby nodes within the network may be compromised and function as malicious actors capable of corrupting the protocol. In particular, the FTSP protocol can be attacked by a single compromised “root” node injecting false timing information into the network [Manzo et al. 2005], even when symmetric keys are used for secure information exchange. However, the threat model in this work treats all nodes in a network as equally compromisable. In cases where a connected sub-component of a network running an FTSP protocol is more resistant to compromise, due to e.g. differences in hardware, a SpartanRPC policy can be established whereby only nodes in the most tamper-resistant sub-component of the network may function as roots, in a manner similar to that described for secure directed diffusion: FTSP time sync updates on any given node can be defined to require a root authorization level. This implies that nodes requiring secure time synchronization must be at most a single radio hop from a root node, but nodes willing to accept possibly corrupted time sync data can extend the network indefinitely. Note that in this scenario, SpartanRPC poli-
cies adapt to heterogeneity in network device hardware, vs. network administration as in the previous example.

Other potential applications of our system include secure routing protocols in heterogeneous trust environments [Karlof and Wagner 2003], transport and network layer protocols [Perillo and Heinzelman 2005], tracking protocols [Brooks et al. 2003], and even mote-based web servers supporting secure channels [Gupta et al. 2005].

1.2. Technical Foundations

Language-Based Security. SpartanRPC provides language-level abstractions for defining remote services and associated security policies. Programmers are presented with an extension of nesC, with new features for defining remote access controlled services, and for invoking those services at specific authorization levels. This hides the implementation details of underlying security protocols and only requires mastery of a simple authorization logic. SpartanRPC programs are compiled in the same manner as nesC programs, in fact the SpartanRPC compiler rewrites SpartanRPC programs to nesC code and compiles the latter.

Asynchronous Remote Procedure Call. As other authors have observed [May et al. 2007], RPC is an appropriate abstraction for node services on the network and supports whole-network (vs. node-specific) programming. Secure RPC is well-studied in a traditional networking environment, and is a natural means of layering security over a distributed communication abstraction.

It is necessary for RPC invocation in a WSN to be asynchronous, since synchronous call-and-return to a remote node would significantly impede performance in the best case and cause deadlock in the worst. In order to minimally impact the nesC programming model, we define RPC invocation as a form of remote task. Local tasks are units of programmer-defined asynchronous computation in nesC, so treating remote computational services as remote tasks fits well in this paradigm. Remote tasks can be invoked on one-hop neighbors, providing a link layer service on which network layer services can be built. For example in Sec. 5 we illustrate how a secure multi-hop data collection protocol can be built using our link layer service.

PK-Based Authorization Policies. SpartanRPC provides language-level abstractions for specifying RPC authorization policies. The policy language we support is RT [Li et al. 2002], which allows network entities to communicate composable credentials for authorizing service invocations. Credentials are typically signed by certificate authorities and do not require shared secrets to validate. In SpartanRPC these credentials are implemented with ECC public keys [Bertoni et al. 2006], which are validated during the initial authorization phase. ECC is significantly more tractable than RSA in a WSN setting. Furthermore, following an initial authorization phase our protocol establishes a shared AES key for subsequent invocations of a given service by the same node. Since hardware AES is available on common WSN radio chipsets, we obtain highly efficient performance for secure invocations following authorization. This is demonstrated with empirical results reported in Sec. 7.

1.3. Outline of Paper

In Sec. 2 we describe the fundamental language abstraction we provide for defining remote services, called duties. In Sec. 3 we describe a modification of remote services that accommodates dynamically changing communication neighborhoods. In Sec. 4 we define our authorization logic, and show how to specify duty posting policies and how they are enforced in the implementation. In Sec. 5 we provide the extended example of secure directed diffusion. In Sec. 6 we summarize novel and important features of
module LEDControllerC { 
   provides remote interface LEDControl; 
}
implementation {
   duty void LEDControl.setLeds(uint8_t ctl) { ... }
}

module LoggerC { 
   uses interface LEDControl; 
}
implementation {
   void f() { ... post LEDControl.setLeds(42); }
}

Fig. 1. Duty Implementation and Invocation Examples

our implementation. In Sec. 7 we discuss empirical results of system performance. We conclude with a discussion of related work in Sec. 8.

2. 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 a more appropriate 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.

2.1. 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 controlling a collection of LEDs:

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 Sec. 2.2. Any remotable interface provided by a component must be specified as remote in its provides specification. The first code sample in Fig. 1 shows an LEDControllerC component that provides the LEDControl interface remotely.

A module on the client node that wishes to use a remote interface simply posts the duty in the same manner as tasks are posted. The use of post emphasizes the asynchronous nature of the invocation. The second code sample in Fig. 1 shows how the LEDController interface would be used. The standard component semantics of nesC provide 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 flows 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 Sec. 3.

When a duty is posted by a client 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. The success or failure of a duty posting
is not signaled to the client in the implementation (just as the receipt or non-receipt of
a message send is not signaled in e.g. the Amsend protocol in TinyOS). Thus any error
semantics for duty postings must be implemented by the application developer.

2.2. Remotable Interfaces

We impose certain requirements on RPC service definitions for ease of implementa-
tion. 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 if and only if it satisfies the following inductive
definition: The nesC built-in arithmetic types, including enumeration types, are
remotable, 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 if and only if it only provides duties whose
argument types are remotable.

3. DYNAMIC WIRES

In an ordinary nesC program the “wiring” between components as defined by config-
urations 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.

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 on the left side of Fig. 2.
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. The
component_set structure defined on the right side of Fig. 2 wraps an arbitrary array of
component_id values.

A component manager is a component that provides the ComponentManager interface
defined at the bottom of Fig. 2. 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
in Fig. 3. This component manager always returns a component set containing a sin-
gle 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, fill-
ing in an array of component IDs based on information gathered earlier in the node's
lifetime.

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 follows:

\[
\text{connection ::= endpoint '->' dynamic_endpoint}
\]

\[
\text{dynamic_endpoint ::= '[' IDENTIFIER ']'} ('.' IDENTIFIER)
\]

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 command \text{elements} in RC is called to obtain a set of component
IDs. The duties I.d provided by those remote components will then be posted on the
host nodes via an underlying link layer communication, the details of which are hidden
from the SpartanRPC programmer. Thus, duties can only be posted on neighbors. Note
that since this call to \text{elements} may return more than one component ID, this is a sort
of fan-out wiring.

For example, the programmer could wire the LoggerC component mentioned in Fig. 1
to LED controller components on a dynamically changing subset of neighbors using a
configuration such as:

\[
\text{LoggerC.LEDControl} \rightarrow [\text{RemoteSelectorC}];
\]

The server's configuration does not need to wire anything to the remote interface
explicitly.
3.3. Callbacks and First-Class IDs

We assume that the 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 component ID, a node can use services on neighboring nodes without knowing their node IDs.

If a node expects a reply from a service that it invokes, the invoking 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. This design provides a natural "split-phase" semantic wherein the invoker of a service can continue executing while waiting for the result of that service.

For example, a service might require the client to provide the node ID and component ID of the component that will receive the service result as arguments to the service invocation. The server could store those values for use by a server-side component manager. It is permitted for a component to be its own component manager making it easy for a service to return a result by posting the appropriate duty. This is a common SpartanRPC idiom.

4. SECURITY FEATURES AND UNDERLYING PROTOCOLS

In Sec. 1 we discuss several motivating examples of interacting sensor networks. In these and other settings it is clear that authorization policies must not depend on secrets shared a priori, since involved social entities may not interact prior to sensor network deployments. To achieve open-endedness and flexibility in authorization policy specification and enforcement, we address the problem at two levels. At a high level, our system integrates a distributed trust management system [Chapin et al. 2008] to specify, comply with, and enforce security policies. At a low level, our system leverages public key cryptography to implement authorization policies.

Distributed Authorization Logic. Authorization in trust management systems is more expressive than in traditional access control schemes such as access control lists or role based access control (RBAC) [Sandhu et al. 1996]. In these simpler models, access is based on identities of principals. But in the distributed scenarios we are considering here, creating a single local database of all potential requesters is untenable. Where there are multiple domains of administrative control, no single authorizer can have direct knowledge of all users of the system. Furthermore, in highly dynamic and volatile environments, no single entity in-network can be expected to keep pace with changes in an authoritative manner. Finally, basing authorization purely on identity is not a sufficiently expressive or flexible approach, since security in modern distributed systems utilize more sophisticated features (e.g. delegation). These problems are addressed by the use of trust management systems such as the RT framework [Li et al. 2002].

The RT framework is a collection of trust management systems with varying expressiveness and complexity [Li et al. 2002; Li et al. 2003; Li and Mitchell 2003b]. We chose to use the base system, RT0, in this foundational presentation because we feel it resides at a “sweet spot” between the competing goals of ease of use, simplicity of implementation, and expressivity. Whenever we refer to RT we mean RT0.

Like other trust management systems such as SPKI/SDSI [Ellison et al. 1999], RT represents principals as public keys and does not attempt to formalize the connection between a key and an individual. This makes creating bogus keys impossible since any new key injected into the system will simply be regarded as a new principal. The RT literature usually refers to principals as entities. RT allows each entity to define roles in a name space that is local to that entity. An authorizer associates permissions with
a particular role; to access a resource a requester must prove membership in the role. In this way RT provides a form of role based access control.

To define a role, an entity issues credentials that specify the role's membership. Some of these credentials may be a part of private policy, others may be signed by the issuer and made publicly available as certificates. The overall membership of a role is taken as the union of the memberships specified by all known defining credentials.

Let \( A, B, C, \ldots \) range over entities and let \( r, s, t, \ldots \) range over role names. A role \( r \) local to an entity \( A \) is denoted by \( A:r \). RT credentials are of the form \( A:r \leftarrow f \), where \( f \) can take on one of four forms to obtain one of four credential types:

1. \( A:r \leftarrow E \)
   This form asserts that entity \( E \) is a member of role \( A:r \).

2. \( A:r \leftarrow B:s \)
   This form asserts that all members of role \( B:s \) are members of role \( A:r \). Credentials of this form can be used to delegate authority over the membership of a role to another entity.

3. \( A:r \leftarrow B:s:t \)
   This form asserts that for each member \( E \) of \( B:s \), all members of role \( E:t \) are members of role \( A:r \). Credentials of this form can be used to delegate authority over the membership of a role to all entities that have the attribute represented by \( B:s \). The expression \( B:s:t \) is called a linked role.

4. \( A:r \leftarrow q_1 \cap \cdots \cap q_n \)
   Where the \( q_i \) are qualified role names such as \( B:s \). This form asserts that each entity that is a member of all roles \( q_1, \ldots, q_n \) is also a member of role \( A:r \). The expression \( q_1 \cap \cdots \cap q_n \) is called an intersection role. In our implementation only two constituent roles \( q_1 \) and \( q_2 \) are allowed in an intersection role. This does not limit expressivity since intermediate roles can be introduced as necessary to handle larger intersections.

For all credential forms \( A:r \leftarrow f \), the principal \( A \) is called the issuer of the credential.

The formal semantics of RT can be expressed in terms of Datalog [Li et al. 2002]. The translation of RT credentials to Datalog requires only a single relation \( \text{isMember} \) to assert when a particular entity is a member of a particular role. A type (1) credential, called a membership credential, is translated into Datalog simply as a fact. For example the credential \( A:r \leftarrow E \) becomes the fact \( \text{isMember}(E; A; r) \).

The meaning of an RT credential \( \llbracket C \rrbracket \) is the Datalog fact or rule to which it translates. Let \( C \) be a set of RT credentials split into two disjoint subsets \( C = C_f \uplus C_r \), where \( C_f \) is the set of all membership credentials. The meaning of \( C \), which we denote as \( \llbracket C \rrbracket \), is the minimum model of the Datalog program \( \llbracket C_f \rrbracket \) using \( \llbracket C_r \rrbracket \) as input [Abiteboul et al. 1995]. The authorizer associates an access permission with a particular role, say \( A:g \), that we call the governing role. Hence we formally define authorization in a given credential environment \( C \) as follows:

**Definition 4.1.** Given a credential set \( C \), entities \( A \) and \( E \), and role \( g \), \( E \) is authorized for \( A:g \) in \( C \), denoted \( C \vdash E \in A:g \) if and only if \( \text{isMember}(E; A, g) \) is in \( \llbracket C \rrbracket \).

One appealing characteristic of the RT trust management system is monotonicity. Negative credentials that explicitly remove entities from roles are not supported. Consequently if an authorizer has incomplete information she might deny access that

---

Logical variables are shown prefixed with '?' to distinguish them from constants.
would otherwise be granted, but she will never grant access that should have been
denied. This property is essential in a wireless sensor network context where the un-
reliability of wireless communication together with the limited memory resources of
sensor nodes make it impossible to guarantee complete information about all roles.

**ECC Public Key Cryptography.** Trust management systems such as RT depend on
public key cryptography. Distributed certificates are protected with digital signatures
made using the issuer’s private key. Furthermore public key cryptography is used to
authenticate message senders to ensure that the entity requesting a service is the
entity it claims to be. However, public key cryptography is computationally expensive
and this presents a problem for limited devices such as wireless sensor network nodes.

The on-node cryptographic computations required by our system are digital signa-
ture verification, session key negotiation using Diffie-Hellman key exchange, and mes-
sage authentication code (MAC) computation using a session key. Of these three op-
erations the first two involve complex public key cryptography. The MAC computation
occurs much more frequently but is much cheaper since it uses hardware accelerated
symmetric key cryptography. In fact, the motivation behind creating session keys is to
avoid public key operations for every message.

Although the RSA public key cryptosystem has been implemented on sensor nodes
[Gura et al. 2004; Wang and Li 2006], the resource consumption required to do so is
considerable. However, the feasibility of using elliptic curve public key cryptosystems
(ECC) on such platforms has been repeatedly demonstrated [Gupta et al. 2005; Malan
et al. 2008; Liu and Ning 2008; Szczecowiak et al. 2008]. Hardware implementa-
tions of ECC for resource limited devices have also been demonstrated [Kumar and Paar
2006; Lee et al. 2008].

ECC can achieve much higher security for a given number of key bits saving memory
and network bandwidth relative to other public key cryptosystems. In our implementa-
tion, described in detail in Sec. 6, we use 160 bit ECC keys, providing a security similar
to 1024 bit RSA keys [Lenstra and Verheul 2001]. We believe this is a reasonable level
of security for the anticipated applications.

4.1. Program Logic

Our authorization model can be viewed as a client-server interaction; respective sides
of the interaction protocol are summarized separately as follows.

**4.1.1. RPC Server Side Logic.** RPC service providers establish policy by assigning gov-
erning roles $A:g$ to remote interface implementations. Service providers also possess
a set of assumed credentials $C$, which establish an authorization environment. As we
will describe in detail, the set $C$ may grow as additional credentials are communicated
to servers. Finally, in the presence of security, client invocations of any RPC service
are not anonymous, but are performed on behalf of some entity $B$, which must be a
member of the governing role $A:g$ to use the protected service.

In summary, access to an RPC level is allowed if and only if the property $C \vdash B \in A:g$
holds, where:

— $B$ is the identity of the RPC client
— $A:g$ is the governing role of the RPC service
— $C$ are the credentials known to the RPC host server

RPC service programmers specify governing roles as part of module definitions—
specifically at remote interface `provides` clauses. Hence, governing roles are associ-
ated with interface `implementations`, not interfaces themselves. This allows applica-
tion flexibility, in that the same interface can be implemented with various authoriza-
tion levels within the same network. Syntax is as follows:
provides remote interface $I$ requires $A.g$

Note the minor modification to previously introduced syntax for remote module definitions, via the requires keyword.

4.1.2. RPC Client Side Logic. In order to use a secure remote module, RPC clients wire to it as for unsecured modules (see Sec. 3.2), but with two additional capabilities: (1) the client specifies under what RT entity the invocations will be performed, and (2) the client may also specify credentials in their possession which are to be activated for use in the invocation. Syntax is as follows:

\[
\text{enable } "C_1, \ldots, C_n" \text{ as } "B" \text{ for } C.I \rightarrow [M].J
\]

For any invocation made through this wiring the credentials $C_1, \ldots, C_n$ will be remotely added to the RPC server's database for the authorization decision, via a process detailed in Sec. 6.3.1. Note that these credentials need not establish authorization entirely by themselves, rather they will be added to the server's existing credentials, all of which will be used in the authorization decision. A special form of the enable clause using "*" for the list of credentials is also supported. This form indicates that all credentials known to the client should be communicated to the server.

Each node is deployed with a collection of ECC key pairs, one for each entity the node represents. When an invocation is made the entity $B$ mentioned in the as clause of the dynamic wire is used in the request. The as clause is optional; if it is omitted a distinguished default entity is used for the invocation.

4.1.3. Example. Suppose that an existing network deployment NetA is imaged with a component SamplingRateC which provides a means to control sampling rates through an interface SamplingRate. Further, since sampling rate modification is a sensitive operation, the network administrators require NetA.control authorization to use this component:

\[
\text{module SamplingRateC } \{ \\
\quad \text{provides remote interface SamplingRate requires "NetA.control"}; \\
\}
\]

Any node supporting this component will transparently receive RT credentials from neighboring nodes and attempt to use those credentials to establish that each client entity is a member of the NetA.control role in the formal sense described above.

Suppose also that nodes in NetA are deployed with the credential

\[
\text{NetA.control } \leftarrow \text{WSNAdmin.control}
\]

Here the role WSNAdmin.control is administered by some overarching network authority. However this authority need not be physically “present” in the network during operation. Instead the credential above represents NetA's access control policy: any entity blessed by WSNAdmin as a controller can control NetA.

Suppose further that another subnet, called NetB, wishes to modify the sampling rate of NetA. A node in NetB might be imaged with the following credentials, among possibly others:

\[
\begin{align*}
\text{WSNAdmin.control } &\leftarrow \text{NetB.control} \quad (1) \\
\text{NetB.control } &\leftarrow \text{NetB} \quad (2)
\end{align*}
\]

Note that credential (1) is issued by the WSNAdmin authority, while credential (2) is issued by NetB. Critically, direct communication with NetA authorities to obtain these credentials is unnecessary.
enable
"WSNAadmin.control <- NetB.control,
NetB.control <- NetB" as "NetB"
for
ClientC.SamplingRate -> [RemoteSelectorC];

Fig. 4. Security Enabled Dynamic Wire

In order to invoke this service the wiring as shown in Fig. 4 could be made on the client side. Note the activation of the necessary credentials, as well as the specification of client identity as NetB.

4.2. Underlying Security Protocols

The protocols that underlie the above described API rely on a combination of public and symmetric key cryptography. We use the TinyECC library [Liu and Ning 2008] for public key functionality, and AES encryption for symmetric key functionality. The latter approach has the added benefit of hardware support on many current embedded platforms, e.g. those employing the Chipcon CC2420 radio.

Our authorization protocol proceeds in the following manner:

(1) Clients communicate their RT credentials to servers as signed certificates, via a periodic broadcast beacon.
(2) Servers receive certificates, verify their authenticity, and maintain a local RT minimum model database.
(3) When an RPC service is first invoked the client initiates a Diffie-Hellman session key negotiation with the server; the server negociates if and only if the client is appropriately authorized for the relevant RPC service.
(4) Following the previous step, normal RPC communication is authorized via a MAC appended to messages sent by the client to the RPC server, using the negotiated session key.

No authentication of the server is needed since duty postings are unidirectional. Any results from the server must be communicated by way of a second duty posted from the server to the client. In that case, the client switches to a server role and the protocol proceeds as described above.

Fig. 5 illustrates the three elements of our security protocol. It is important to note that these operate asynchronously. Certificates are broadcast and checked by a background process. Session key negotiation, once initiated, is also handled by a background process. Duties posted before a session key is fully negotiated will fail, and session key negotiation attempted before the necessary certificates have been transferred will also fail. However, if authorization is permitted by policy then eventually it will be granted once the requisite background processing is complete.

This decoupling of the three stages of the protocol is well suited to the event driven architecture of wireless sensor network nodes. The cryptographic operations done in the background can be very time consuming; in principle a node can continue with its other activities during that processing.

If certificates fail to transfer or if session key negotiation fails to complete, those operations will simply be retried when necessary. A new node in the network will gather certificates as they are broadcast and inform its neighbors of its own certificates without any special handling. Replay attacks are not relevant in this context. An "attacker" that replays messages in our protocol would actually be assisting in the communication process by reinforcing communication.
A.12 P. Chapin and C. Skalka

Certificate
Sender

Session Key
Sender

Client

Certificates

Certificate
Receiver

Session Key
Receiver

Server

Node A

Node B

Fig. 5. SpartanRPC Security Protocol Elements

A. \( r \leftarrow B. s \cap C. t \)

<table>
<thead>
<tr>
<th>4</th>
<th>A (40)</th>
<th>r</th>
<th>B (40)</th>
<th>s</th>
<th>C (40)</th>
<th>t</th>
<th>sig (42)</th>
<th>chk (2)</th>
</tr>
</thead>
</table>

Fig. 6. Intersection Certificate Format (parenthesized numbers indicate byte counts)

4.2.1. RT Certificate Format and Validation. Our certificate representation of an RT credential contains the public keys denoting entities mentioned in the credential. Roles are identified by one byte numeric codes and are scoped by the entity defining the role. Credential forms are distinguished by numbers \{1, 2, 3, 4\}. Certificates are also signed by their issuing authority. Conveniently, the issuing authority is always mentioned in a credential, e.g. the issuing authority of A.\( r \leftarrow B \) is A, so the means to verify the certificate (i.e. the relevant public key) is always included with it for free. This does not introduce a security problem. Since entities are identified directly by their keys, an attacker who creates a new key is simply creating a new entity. The over-the-air format for the intersection certificate is illustrated in Fig. 6. The other certificate forms are organized in a similar way.

Verification of RT certificates is performed using the TinyECC library, and is the most computationally expensive component of our system. On the TelosB platform verification of certificates requires on the order of 90 seconds. Also, because we use a unreliable channel to communicate certificates (as we discuss below), verification failure can easily occur as the result of message corruption. In order to catch this situation early before expending resources on ECC verification, we append to each certificate a 16 bit Fletcher checksum, which are commonly used in WSNs since their error detection properties are almost as good as CRCs with significantly reduced computational cost [Fletcher 1982]. Thus, certificate verification is performed in two stages; first, message integrity is established via the Fletcher checksum, and if this succeeds full certificate verification is done.

Currently the certificates carry no lifetime information and are considered to be valid forever. This is not ideal since a certificate issuer may eventually change her policy but currently has no way to revoke old certificates. However, adding a feature for certificate revocation introduces non-monotonicity into the semantics of the authorization logic [Li and Feigenbaum 2001; Rivest 1998].
Adding an expiration time to the certificates is more logically appealing but would require nodes to support real time services and some degree of time synchronization. This is a non-trivial extension of our basic system that is beyond the scope of our work.

4.2.2. Certificate Distribution. Prior to deployment, individual nodes are imaged with certificates that will be available during live operation. Certificates are generated off-line using a custom tool created for that purpose [Willett 2011].

When a certificate is received by a node that supports RPC services, the receiver checks whether it has already received the certificate. This is accomplished by maintaining a database of Fletcher checksums of previously received certificates. If the certificate is both new and verifies correctly as described above, the receiver adds the credential represented by the certificate to its knowledge base and updates its RT minimum model. By keeping the minimum model updated with each new certificate received, authorization checks for future RPC service invocations are very fast. If Fletcher checksums collide and a new certificate is mistakenly believed to have already been received, it will be rejected, which is incomplete but sound from an authorization perspective.

4.2.3. Session Keys and Authorized RPC Invocations. Authorized RPC invocations are made using MACs on invocation messages, created with AES session keys established between each client/server pair for a particular service. Session keys are established via ECC-based Diffie-Hellman. The message format for this Diffie-Hellman exchange request is illustrated in Fig. 7. The client sends a session key negotiation message containing its public key and a triple composed of the server node address \( N \) and the component id \( C \) and interface id \( I \) of the desired service. The server will respond with a similar message containing the server’s public key and the client’s node address, but only if the client is authorized.

Once session keys between a client and all necessary servers are established, invocation messages communicating requests for the service include a MAC generated with each session key, verified on the receiver end. Fig. 8 shows the format of such messages.

Invocation messages include multiple MACs since separate session keys are negotiated with each of \( n \) servers. If the duty arguments consume \( d \) bytes of data, then invocation messages consume \( 2 + n + d + 4n \) bytes. In practice this puts significant restrictions on the amount of data that can be passed to duties. As we describe in Sec. 6.3.1 our implementation uses a 43 byte message payload for sending certificate fragments. We feel that using this same payload size for invocation messages allows for reasonable values of both \( d \) and \( n \). Alternatively an implementation could send multiple invocation messages with one for each server, reducing the number of MACs...
required on each message to one. However, that greatly increases radio traffic since the
duty arguments and active message overhead must be duplicated for each message.

To conserve space in the invocation messages we only use a 32 bit MAC. Such a
small MAC would not normally be considered secure. However, wireless sensor net-
works generate data so slowly that attacking even such a short MAC is not considered
feasible [Karlof et al. 2004; Luk et al. 2007].

A note on multicast security. As discussed in Sec. 3, fan-out wirings are a common
idiom, and provide a form of multicast communication. However, as is well-known use
of MACs for security in a multicast setting presents significant challenges. In particu-
lar, while $n$-way Diffie-Hellman can be used to negotiate secret keys between $n$ actors,
such a scheme cannot be used in light of the possibly heterogeneous authorization re-
quirements we anticipate. For example, suppose a node $A$ fan-out wires to service $s$
on distinct nodes $B$ and $C$, and suppose also that $A$ is authorized for $s$ on both nodes
but that $B$ is not authorized for $s$ on $C$ and vice-versa. If a single session key were
negotiated between $A$, $B$, and $C$ in this case, then $B$ could make unauthorized use of
$C$'s version of $s$ and vice-versa. While a variety of techniques have been proposed to
mitigate this problem [Canetti et al. 1999], most typically rely on very large multicast
groups and are not applicable in our setting. Thus, in our system, fan-out wirings to
authorized services are implemented by computing multiple MACs on outgoing mes-
sages, one for each distinct session key.

4.3. Security Properties

We stress that our scheme is intended to enforce authorization, which we achieve via
the protocols described above. Integrity is a side effect of this, since we use MACs
to enforce authorization, which are computed over complete message payloads and
are verified by the receiver. Although confidentiality is not directly supported by our
current protocols, it could be easily added. In particular payloads could be encrypted
using negotiated session keys (payloads are currently sent as plaintext).

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 argument. The server could verify that the count increases mono-
tonically as a simple form of replay protection.

We feel that delegating replay protection to the application is appropriate since Spar-
tanRPC is intended to be a low level infrastructure on which more complex systems
can be built. Furthermore the need for replay protection is likely to be application
specific.

Perhaps the most problematic vulnerability of our system is to denial of service at-
tacks. It is not clear how these could be mitigated without significant changes to the
underlying security protocols. For example, a constant flood of certificates over the cor-
rect channel would place receiving nodes in a constant state of ECC digital signature
verification, potentially consuming large amounts of CPU time and energy. Mitigation
of such attacks is outside the scope of this work but has been discussed in the literature
[Raymond and Midkiff 2008].

5. EXAMPLE: SECURE DIRECTED DIFFUSION

To illustrate the usefulness of our design we have implemented a secured version of the
well-known directed diffusion protocol [Intanagonwiwat et al. 2003] for ad-hoc routing
of data in a sensor network. It is one example of the publish/subscribe paradigm for
data gathering. In our secure version, facilities for subscribing to a data stream are
defined as secure RPC services by data stream providers. Directed diffusion supports
multi-hop data collection, so this example illustrates how our link layer RPC service supports network layer communication.

The directed diffusion algorithm [Intanagonwiwat et al. 2003] allows a node to subscribe to a data stream by expressing an interest in it. In our example, an interest is expressed as temperature data above a given threshold. A certain data rate, expressed as a time interval between samples, 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.

5.1. Directed Diffusion in Multiple Domains

To demonstrate our authorization logic, we imagine that subscription to a data stream requires a certain authorization level. Assume that the Federal Emergency Management Agency (FEMA) manages a role tempControllers that specifies entities with the authority to define who can use the temperature sensing service. FEMA does not own any sensor networks itself but is nevertheless trusted to make appropriate delegation decisions, and so can function as a kind of certificate authority.

FEMA can delegate control over the service to the state of Vermont \( V \) by issuing a certificate denoting the following credential:

\[
\text{FEMA}.\text{tempControllers} \leftarrow V
\]

Note this certificate will be signed by FEMA. The state of Vermont can then place fire departments in various jurisdictions into a tempUsers role by issuing, for example, its own signed certificates denoting the following credentials:

\[
\text{V}.\text{tempUsers} \leftarrow A \quad \text{V}.\text{tempUsers} \leftarrow B
\]

where \( A \) is the RT entity of the Addison town fire department and \( B \) is the RT entity of the Burlington city fire department.

The sensor network of the Addison town fire department protects access to the temperature sensing application using the governing role \( A.t \). The nodes are deployed with the following credential pre-loaded to serve as access control policy.

\[
A.t \leftarrow \text{FEMA}.\text{tempControllers}.\text{tempUsers}
\]

This policy allows access to an entity \( E \) if a FEMA tempController says \( E \) is a tempUser. The Addison town fire department nodes are also deployed with the following certificates gathered off-line from public sources such as FEMA's web site.

\[
\text{FEMA}.\text{tempControllers} \leftarrow V \quad \text{V}.\text{tempUsers} \leftarrow A
\]

These certificates prove that \( A \) is a tempUser according to at least one FEMA certified tempController. Similarly the Burlington city fire department protects its temperature sensing application using the governing role \( B.t \). Its nodes are deployed with a similar policy statement and with certificates showing that \( B \) is a member of \( V.\text{tempUsers} \).

If both fire departments are called upon to work together fighting a large fire, their sensor networks will now be able to interoperate, for purposes of temperature sensing, \textit{with no prior coordination}. Furthermore suppose the fire rages out of control and the fire department from neighboring Crown Point, New York, \( C \) is called in to assist. The new nodes introduced by \( C \) will also be able to interoperate with both sets of existing nodes provided they are deployed with certificates such as the following, where \( N \) is the RT entity of the state of New York:

\[
\text{FEMA}.\text{tempControllers} \leftarrow N \quad \text{N.\text{tempUsers}} \leftarrow C
\]
In our example we implemented the program for the Addison town fire department. However, the other programs are identical, from the point of view of temperature sensing, except for changes in the governing role on the remote services and changes in the deployed policy and certificates.

5.2. Interfaces
Interest and data propagation are handled by separate interfaces, as shown in Fig. 9, each containing a single duty.

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. The duration parameter of the `set_interest` duty specifies a lifetime of the interest. Once an interest expires it is removed from the node's interest cache. Temperature values are expressed as integers presumably corresponding to the output of an analog to digital converter. Similarly time intervals are expressed as integer multiples of some unit time, the exact value of which is arbitrary.

A node passes data to its interested neighbors by posting the `set_data` duty on those neighbors. The `originator_node` is the ID of the node where the data was originally observed; the node soliciting this data will typically want to know its provenance.

Both of these interfaces include the sender's node ID as an explicit parameter. The nodes use that information to track paths through the network. Although the node ids are also part of the low level radio packets sent between the nodes, we chose for demonstration purposes to manage the interest and data information strictly at a higher level in the protocol stack. As usual greater efficiency may be possible by mixing protocol layers.

5.3. 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.

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

```c
enable "*" for
  InterestManagerC.NeighborSensors ~> [InterestManagerC].InterestManagement;
```
In this example, we assume the nodes are deployed with a single default entity. As discussed in Sec. 4.1.2, because the as clause is missing this wiring makes the invocation on behalf of that entity. We anticipate that single entity nodes will be common.

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 all available certificates; neighbors can be in multiple security domains.

5.4. Interest Management

The interest manager has a partial specification as follows:

```plaintext
module InterestManagerC {
    provides interface ComponentManager;
    provides remote interface InterestManagement requires "A.t";
    uses interface InterestManagement as NeighborSensors;
}
```

Because the interest manager is its own component manager, setting up target node addresses entails updating an internal component_set. 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 as shown in Fig. 10.

The “well known” local component ID of the interest manager is used to specify which component on 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 cannot 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.

5.5. 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 same role as for interests. The specification of the data manager provides interface DataManagement requiring role A.t but there are no security related artifacts in the body of the data manager’s implementation.

6. IMPLEMENTATION

In this section we describe the implementation of the SpartanRPC system using RTo trust management for authorization. We call our implementation SprocketRT [Chapin 2012].

SprocketRT rewrites a SpartanRPC enabled program into a pure nesC program and provides a supporting runtime system. The runtime system contains certain components, described in Sec. 6.3, that are common to all SpartanRPC programs. In addition SprocketRT generates other components that are specialized for the particular program being rewritten.

6.1. Service Endpoint Identifiers

Service endpoints are identified by the 4-tuple \((N, C, I, D)\) where \(N\) is the TinyOS ID of the node on which a duty is implemented. \(C\) is the local component ID assigned to each component that provides a remotable interface. \(I\) is an interface ID, required since a component may provide more than one remotable interface. Interface IDs are component-level unique. Finally \(D\) is a duty ID, which must be interface-level unique.

In the current version of SprocketRT, \((C, I)\) values are assigned statically by an arbitrary (automated or social) process. SprocketRT accepts configuration files that define the association between \((C, I)\) values and the entities to which they refer. Duties are numbered in the order in which they appear in their enclosing interface definitions.

Some RPC systems, such as ONC RPC, allow each node to provide a registry of RPC services available on that node [Srinivasan 1995]. When a large number of RPC services are provided by a node it becomes unreasonable to expect clients to have hard coded knowledge of the endpoint identifiers for all those services. Instead clients communicate with the single well known registry to obtain endpoint identifiers that were dynamically assigned. In contrast we assume this configuration information is known a priori to all interacting actors. It is unclear if sensor networks could benefit from a more sophisticated technique for defining and communicating endpoint identifiers, but it would be an interesting topic for future work.

6.2. Program Rewriting

There are five major features requiring SpartanRPC-to-nesC rewriting by SprocketRT: interface definitions, call sites where remote services are invoked, duty definitions, dynamic wires, and server components providing remote interfaces. In addition SprocketRT generates a stub component for each dynamic wire, and a skeleton component for each remote interface. Finally SprocketRT generates configurations that wrap server components. Here we summarize rewriting strategies for these features.

6.2.1. Interfaces, Call Sites, and Duty Definitions. Duty declarations in interfaces are rewritten to command declarations by substituting command for duty. Since nesC commands are allowed to have arbitrary parameters, duties with parameters present no complications. SprocketRT verifies that if an interface contains a duty, then the only
declarations in that interface are duties. Sprocket\textsubscript{RT} further verifies that the parameters of each duty, if any, conform to the restrictions described in Sec. 2.2.

Call sites where duties are posted are rewritten to command invocations by substituting call for post. Only post operations applied to duties are rewritten in this way. Finally, duty definitions are rewritten to command definitions by also substituting command for duty.

6.2.2. Authorization Interfaces. The rewriting process makes use of two interfaces that mediate the interaction between the Sprocket\textsubscript{RT} generated code and the security processing components of the run-time system. Fig. 11 shows how a message, entering from the left, is extended with authorization information by the client and then passed to the server where the authorization information is checked.

The AuthorizationClient interface abstracts the details of how an authorized message is prepared before being sent. The AuthorizationServer interface abstracts the details of how authorized messages are processed after they are received. This design allows for pluggable authorization mechanisms. Future versions of Sprocket\textsubscript{RT} could potentially support other authorization schemes than those described here, in a modular fashion.

The authorization interfaces provide their services in a split-phase manner so that potentially long-running authorization computations can be performed while allowing the node to continue other functions.

In the current implementation, two kinds of authorization are supported. On the client side the precise method used depends on the dynamic wire over which a particular communication takes place. On the server side it depends on the presence of a requires clause on the remotely provided interface.

The full \textsubscript{RT}\textsubscript{0} mechanism is supported by client and server components \textsubscript{ACRT0C} and \textsubscript{ASRT0C} respectively (details about this mechanism are discussed in Sec. 4 and Sec. 6.3). In addition a “null” authorization is supported by client and server components \textsubscript{ACNullC} and \textsubscript{ASNullC} respectively. The null authorization components perform no operation. They are used for dynamic wires that do not require authorization and remote interfaces provided publicly by servers.

6.2.3. Dynamic Wires. In the following, we use italics for metavariables that range over arbitrary identifiers. The reader is referred to the rewriting schema defined in Fig. 12. Configurations containing dynamic wires are rewritten to configurations that statically wire the using component \textsubscript{ClientC} to a stub \textsubscript{Spkt\_n} that interacts with the appropriate component manager \textsubscript{SelectorC} and that handles radio communication.
Dynamic Wire

\[ ClientC.I \rightarrow \{SelectorC\}.I; \]

Rewritten as...

components Spkt_n;
ClientC.I \rightarrow Spkt_n;
Spkt_n.ComponentManager \rightarrow SelectorC;
Spkt_n.AuthorizationClient \rightarrow AuthorizerC;
Spkt_n.Packet \rightarrow AMSenderC;
Spkt_n.AMSend \rightarrow AMSenderC;

Fig. 12. Dynamic Wire Rewriting

Every stub generated by Sprocket_{RT} is uniquely identified over the scope of the entire program by an arbitrary integer \( n \).

The AuthorizerC component is ACNullC in the case where no authorization is requested.

In contrast a dynamic wire using either an enable or an as clause is rewritten the same way except that the AuthorizerC component is ACRT0C. Furthermore, the list of enabled credentials is added to local certificate storage by Sprocket_{RT}. Certificates in storage are periodically beaconed at run-time; see Sec. 6.3 for more discussion of certificate beaconing. Finally, the entity on whose behalf the RPC invocation is performed is specified in the session key negotiation message sent to the server; see Sec. 6.3 for details.

The Spkt_n stub provides the same interface provided by ClientC. Wherever a duty is posted by ClientC in source code, the rewritten call invokes code in the stub that was specialized to handle that duty. The stub calls into the component manager at run time to obtain a list of the dynamic wire’s endpoints and then prepares a data packet containing remote endpoint addresses and marshaled duty arguments. Finally the stub calls through the AuthorizationClient interface to perform whatever authorization is needed.

6.2.4. Remote Services. For nodes supporting RPC services, Sprocket_{RT} generates a skeleton component for each remote interface provided. This skeleton contains a task corresponding to each duty provided in the interface, and every generated skeleton is distinguished by a unique integer \( n \) taken from the same numbering space as the generated stubs.

When messages are received on a node that provides RPC services, they are examined to see if they are duty postings and thus to be handled by a skeleton. If so, the AuthorizationServer interface is used to authorize the message. If authorization succeeds, the task corresponding to the specified duty is posted. That task simply calls into ServerC through the original interface \( I \). Thus the task-like behavior of duties is ultimately implemented using actual nesC tasks inside the server skeletons. Duty parameters are conveyed via module-level variables accessed by the duty tasks (since nesC tasks do not take formal arguments).

For each component that provides at least one remote interface, Sprocket_{RT} creates a configuration that wires the corresponding skeleton(s) to that component. This new configuration wraps the original component and replaces uses of the original component in other configurations in the program.
6.3. Runtime Support
Sprocket uses an extensive runtime system to manage certificates, credentials, session keys, and related data. In this section we discuss the implementation of this runtime system.

6.3.1. Certificate Processing. Every cooperating node contains a certificate sender component and a certificate receiver component. These components transfer RT₀ certificates between nodes and take care of verifying and storing them. Neither of these components provide any interfaces. Instead they run as background daemons, performing their function asynchronously. Fig. 13 shows the certificate processing architecture of a node.

Nodes are deployed with a collection of certificates in read-only storage. This collection is generated by Sprocketₖₜ based on the enable clauses of all dynamic wires used in the program. At run time a certificate C is activated the first time a dynamic wire is used where C is mentioned in an enable clause. Once activated a certificate is never deactivated.

At boot the certificate sender starts a periodic timer. When the timer fires, the node broadcasts all activated certificates in its certificate storage. Limiting these broadcasts to activated certificates prevents a new node from immediately overwhelming its neighbors with unnecessary computation; the certificates that are actually used will be transmitted by clients and processed by servers on an as-needed basis.

To prevent adjacent node certificate broadcasts from colliding, the certificate broadcast interval is modulated randomly by ±10%. For example if the nominal broadcast interval is one minute, the actual time varies randomly between 54 and 66 seconds.

The certificate distribution strategy is robust in the face of new nodes being added to the network or intermittent radio connectivity. If a node fails to receive certain certificates from its neighbors it will have another opportunity to do so when those neighbors rebroadcast their certificates. There is a trade off between the broadcast interval, responsiveness, and network energy consumption. A short broadcast interval allows authorizations to succeed “quickly” since neighbors become aware of the necessary credentials early, but at the cost of increased radio traffic and power consumption.

The certificate sender never tries to broadcast certificates received from other nodes. Doing so would cause every certificate to flood the network, creating excessive network traffic and computational workload. Since SpartanRPC is a link level RPC mechanism there is no point in propagating certificates beyond the neighborhood.

Certificates range in size from 124 bytes for the membership credential to 166 bytes for the intersection credential. This is larger than the maximum payload size limit of TinyOS T-Frame Active Message packets as transported by IEEE 802.15.4 [Society 2003; Hui et al. 2008]. It is much larger than the default maximum payload size of 28
bytes used by TinyOS [Levis]. Consequently the certificates are fragmented into four fragments requiring a maximum payload size of 43 bytes. These fragments are sent back to back with a 200 ms delay between each to allow the receiver time to assemble them. Fragments are sent in order with no fragment identifiers. To stay synchronized with the sender, receivers expect to receive all the fragments in a timely manner. If a fragment is not received within 750 ms of the previous fragment, the partial certificate is discarded on the assumption that the expected fragment was lost.

Once a received certificate is known to be new and has been verified, the credential it contains is extracted and stored. Public keys representing $RT_0$ entities are stored in a key storage component; all references to those keys in the credential storage are by way of a small integer index. This saves memory in the common case where the same key is used in multiple credentials.

The credential storage also contains the $RT_0$ minimum model implied by the currently known collection of credentials. Each time a new credential is added to storage, the minimum model is updated. This is done by repeatedly applying the Datalog rules implied by the credentials to the current model until a fixed point is reached [Li and Mitchell 2003a]. If the key storage overflows, credentials using the removed keys are also removed.

6.3.2. Session Key Negotiation. Public key cryptography is much too computationally expensive to use for routine duty postings. Sprocket’s run time system addresses this by negotiating session keys between the sender (client) and receiver (server) nodes. Fig. 14 shows the session key processing architecture of a node.

![Session Key Processing Architecture](image)

The client maintains a session key storage that is indexed by the triple $(N, C, I)$ where $N$ is the remote node ID, $C$ is the remote component ID, and $I$ is the remote interface ID. A session key is thus created for each combination of these IDs. The server also maintains a session key storage indexed by $(N, C, I)$. In this case $N$ is the node ID of the client and $C, I$ are the component and interface IDs on the server to which that client is communicating. Since any given node can be both server and client, each session key storage entry has a flag to indicate the nature (client-side or server-side) of the session key.

When a server receives a session key negotiation request message from a client node $N$ containing the public key $K_{cp}$ of the requesting entity (as mentioned in the as clause of the dynamic wire) and the $(C, I)$ address of the desired service, the following steps are taken:

1. Authorization of $K_{cp}$ for service $(C, I)$ is checked using the RT minimum model computed by the certificate receiver. If authorization fails nothing more is done.
A session key is computed using elliptic curve Diffie-Hellman and added to the session key storage under the proper \((N, C, I)\) value. The key is stored as a remote client key.

A message is returned to the client containing the server’s public key \(K_{sp}\) and the original \((N, C, I)\) values used by the client. This is so the client is able to compute the same session key and associate it with the proper endpoint from its perspective.

Note that under this protocol every session key computed between nodes \(N_A\) and \(N_B\) for the same requesting entity will be the same. This is not a problem since the server node uses \((C, I)\) to look up the session key in its storage. If a node attempts to access an unauthorized service, no entry for that \((C, I)\) will exist in the server’s session key storage and access will fail.

This strategy enables an optimization we call session key stealing that allows the server to skip the expensive ECC Diffie-Hellman computation in some cases. Suppose the server has already negotiated a session key with one node acting as domain \(B\), say \(N_{B1}\). If a second node acting as domain \(B\), say \(N_{B2}\), requests a session key, the server can simply copy the key from the \((N_{B1}, C, I)\) entry of its storage to the \((N_{B2}, C, I)\) entry. Ultimately this optimization relies on the fact that the credential receiver component keeps the RT minimum model up to date. As a result authorizations are very fast. Once the server has created a session key with one node in a domain \(B\) it can fully authorize a second node in that domain trivially.

### 6.4. Duty Authorization

The current implementation uses the hardware support for AES encryption built into the Chipcon CC-2420 radio chips [Chipcon 2004] to create a message authentication code for each outgoing duty post using the previously negotiated session key. This saves time but it especially saves space since software implementations of AES require non-trivial program memory space [Chapin and Skalka 2010].

### 7. EMPIRICAL RESULTS

The practicality of our system depends on its cost in terms of memory and processor overhead. In this section we report on performance measurements made on our implementation. In summary, we show that our combined use of public and private key cryptography in the underlying security protocol imposes a low amortized cost over time, despite high costs for initial authorizations.

**Test Environment and Programs.** Since many communication chips now support hardware AES encryption, we were primarily interested in demonstrating performance using that feature. In particular, the popular Tmote Sky wireless sensor mote [moteiv 2006] uses a Chipcon CC2420 transceiver with hardware encryption. Unfortunately, the standard TOSSIM simulation environment does not model hardware encryption for TinyOS 2.1, so all of our tests were performed on real hardware. We used Tmote Sky motes, with 16Kb of RAM, 48Kb of ROM and an 8MHz MSP430 microcontroller running TinyOS 2.1.2 [TinyOS ].

We exercised our system using both small test programs and using our implementation of the directed diffusion example described in Sec. 5. The small test programs consisted of a simple client/server pair where the client repeatedly sent a message containing a single 16 bit value to the server. The purpose of these tests was to explore the overhead induced by our system with minimal obscuring effects from application logic. The percentage overhead observed with the small programs is thus a worst case overhead. In contrast, the directed diffusion example allowed us to test the behavior of the system in a more realistic, long-running setting. It serves as a demonstration that our
system is feasible in practice, and allowed us to exercise our system in a multi-mote, multi-hop network environment.

7.1. Memory Overhead for Security Features
The Sprocket\textsubscript{RT} run time system uses several memory caches to hold key material, credential information, and the minimum model implied by the set of known credentials. These caches are statically allocated but must be stored in RAM since their contents are dynamic. Table I summarizes the RAM consumption of the various storage areas used by the current implementation.

The number of items in each cache are tunable parameters. The optimum settings depend on the intended application. The values in Table I attempt to strike a balance between usability and flexibility on one hand and excessive memory consumption on the other. In applications where these needs are more clearly known a priori, the sizes of the caches can be adjusted to potentially result in lower memory consumption.

Table II shows the overall memory consumption of two small client/server pairs. The baseline pair handle all communication through normal Active Message packets that are explicitly programmed by the user. The SpartanRPC pair uses our system which includes support for certificate distribution and verification, session key management, authorization logic, and MAC computations.

Although the overhead incurred by the Sprocket\textsubscript{RT} runtime system is significant on our test platform, nearly 80% of RAM and 50% of ROM resources are still available. Furthermore, these memory usage numbers scale to denser neighborhoods and extended RPC services.

7.2. Transient and Steady State Processor Overhead
The execution performance of our system displays two distinct behaviors. The first is a transient behavior that occurs after a node boots when certificates are exchanged and session keys are negotiated between the new node and its neighbors. The second is a steady-state behavior that occurs during normal operation. The transient overhead of our system is large but the steady state overhead is not. In a quasi-static environment where new nodes enter the network infrequently the transient costs are amortized and it is the small, steady state overhead that dominates.

To explore the steady state overhead three tests were conducted.

(1) A baseline test where the message handling was done explicitly using traditional Active Message interfaces.
Table III. Maximum message transfer rate

<table>
<thead>
<tr>
<th>Test</th>
<th>messages/s</th>
<th>% Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>128</td>
<td>–</td>
</tr>
<tr>
<td>Duties</td>
<td>119</td>
<td>7.0</td>
</tr>
<tr>
<td>MAC</td>
<td>87</td>
<td>32.0</td>
</tr>
</tbody>
</table>

(2) A duties test where the SprocketRT system was used but no authorization was requested. This is equivalent to using the authorization components ACNullC and ASNullC in Fig. 11.

(3) A MAC test where authorization was requested but where the session key storage areas were preloaded with appropriate session keys.

Table III shows the maximum rate at which messages could be sent and received by the test programs mentioned above. Note that the MAC test made use of the hardware assisted AES support provided by the CC2420 radio chip. These results show that maximum message send rates decrease by a factor of 7% due to the addition of our duties program logic (our security API), and further decreases by a factor of 25% due to MAC calculations. We note that the latter overhead would be incurred in any system using CC2420 MAC calculations.

The transient runtime overhead of our system can be subdivided into three primitive operations: the time required to transmit and verify a certificate, the time required to build the minimum model, and the time required to negotiate a session key. Two of these operations require lengthy public key computations and dominate the transient behavior of our system. Thus the performance of our system in this regard is closely tied to the performance provided by TinyECC, which we used with default settings (no optimizations). Table IV shows the times required for each of the primitive transient operations in our implementation.

Table IV. Processing time for transient operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certificate Verification</td>
<td>82s</td>
</tr>
<tr>
<td>Minimum Model Construction</td>
<td>370μs</td>
</tr>
<tr>
<td>Session Key Negotiation</td>
<td>80s</td>
</tr>
</tbody>
</table>

The time required to build the minimum model is directly related to the number and nature of the credentials involved. In our test we used a collection of five representative credentials, one of each type. In any case this time is entirely negligible compared to the other transient operations.

The time quoted for session key negotiation represents the time required for both negotiating partners to compute the session key. In the current implementation the two negotiating nodes do this sequentially with the server node computing the session key before responding to the client node. This was done in case the session key computation failed on the server to ensure that the client does not falsely believe a session key was successfully negotiated.

7.3. Transient State Times for Directed Diffusion

As argued above, the overhead imposed by our system is primarily the time the network spends in a initial transient state when credentials are verified and session keys are negotiated. Subsequently, the network enters a steady state during which the main cost is a 32% reduction in maximal message send rates due to hardware AES encryption. In order to evaluate the performance of our system in a realistic application, we
therefore quantified the transient state times of the secure directed diffusion application described in Sec. 5. In our experiments we elected a single node to repeatedly express an interest and we observed how long was required for that interest to flood the network. This time depends on three major factors:

1. The number of certificates transferred.
2. The number of neighbors for each node.
3. The number of hops to the “far” edge of the network.

We conducted two experiments, one on a single hop (star) network and another on a multi-hop (mesh) network.

In the single hop case, transient time $T$ can be described by the following equation:

$$T = n_C B + V + n_n K$$

where $B$ is the certificate broadcast interval, $V$ is the certificate verification time, $K$ is the session key negotiation time, $n_C$ is the number of certificates and $n_n$ is the number of neighbors. Since $B$ was set to 90 seconds, which is greater than $V$, certificate verification for $n_C$ certificates takes time $n_C B + V$ given a 90 second system initialization period. And since session keys need to be negotiated with $n_n$ neighbors in turn, $T$ also comprises a $n_n K$ delay. Table V shows the transient time required to flood a network where all nodes are one-hop neighbors of the root node. Values are given for three different policies with different numbers of certificates transferred from the root to the neighbors.

<table>
<thead>
<tr>
<th># neighbors</th>
<th>1 Cert</th>
<th>2 Certs</th>
<th>3 Certs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4m03s</td>
<td>5m27s</td>
<td>6m52s</td>
</tr>
<tr>
<td>2</td>
<td>5m16s</td>
<td>6m50s</td>
<td>8m24s</td>
</tr>
<tr>
<td>3</td>
<td>6m32s</td>
<td>7m57s</td>
<td>9m30s</td>
</tr>
<tr>
<td>4</td>
<td>7m50s</td>
<td>9m22s</td>
<td>10m51s</td>
</tr>
</tbody>
</table>

We explored the behavior of our system in a multi-hop environment by creating a linear mesh network. Each node (except the root) had a single downstream neighbor. All nodes were booted simultaneously and the time required for interest information to reach each node was observed. The policy used required only a single certificate to be transferred between nodes. Table VI shows the results of several runs.

<table>
<thead>
<tr>
<th>Run</th>
<th>1 hop</th>
<th>2 hops</th>
<th>3 hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4m05s</td>
<td>7m24s</td>
<td>9m10s</td>
</tr>
<tr>
<td>2</td>
<td>3m12s</td>
<td>5m12s</td>
<td>6m30s</td>
</tr>
<tr>
<td>3</td>
<td>3m57s</td>
<td>7m37s</td>
<td>9m15s</td>
</tr>
<tr>
<td>4</td>
<td>4m09s</td>
<td>7m15s</td>
<td>8m49s</td>
</tr>
<tr>
<td>Average</td>
<td>3m51s</td>
<td>6m52s</td>
<td>8m23s</td>
</tr>
</tbody>
</table>

The reason for variations in transient times over each run was due to a randomized element in the protocol, specifically a randomized $\pm 10\%$ interval in certificate broadcast times to avoid collisions. In these results it is essential to note that for hops $> 2$, extra transient time is comprised solely of session key negotiation times (80s per session key, see Table IV) that are forced by duty postings as interests propagate through
the network. Certificates are broadcast and verified in parallel throughout the network upon system bootup, during the same time period required for the root’s interest to propagate through the first and second hops.

8. RELATED WORK AND CONCLUSION
We have designed and implemented SpartanRPC, a dialect of nesC with a light weight, link-layer, secure RPC facility. SpartanRPC is a middleware technology supporting secure WSN applications comprising multiple security domains. It is ideal for settings in which multiple networks administered by distinct social entities cooperate to obtain a holistic behavior. A language level, trust management based authorization mechanism provides application programmers with an easy and effective means for specifying and enforcing complex access control policies in a multi-domain setting. Our implementation is based on public keys, supporting an open-world security model where shared secrets need not be known a priori. We have reported on testing and performance evaluations, providing evidence of the practicality of SpartanRPC in its intended application space.

8.1. Related Work
Extending wireless sensor network software platforms with support for secure interactions between domains has been studied in previous research on SSL for WSNs [Jung et al. 2009]. However, this work was focused on extending the Internet to WSNs (aka “IP for WSNs”), whereas SpartanRPC is a more general system for enhancing secure communications within a WSN. Research on WSN security has also addressed secure routing [Karlof and Wagner 2003], link layer security [Karlof et al. 2004], cryptography [Bertoni et al. 2006], key distribution [Camtepe and Yener 2005], and hardware issues [Perrig et al. 2004]. In contrast to these low-level systems, SpartanRPC provides language-level abstractions for secure RPC services.

Previous related work also illustrates interest in and useful applications of RPC in embedded networks. For example, the Marionette system uses network layer RPC for remote (PC-based) analysis and debugging of WSNs [Whitehouse et al. 2006]. The Fleck operating system provides a small pre-defined set of RPC services for WSN applications, while the trustedFleck system extends this with a form a secure RPC [Hu et al. 2009; Hu et al. 2010]. S-RPC provides an RPC facility for sensor networks that allows remote services to be added to the system dynamically [Reinhardt et al. 2011]. SpartanRPC differs from these systems in that it extends the nesC programming language (unlike trustedFleck) to allow programmer definition of secure RPC services (unlike S-RPC) that can be accessed by nodes within the network itself (unlike Marionette). Our system is similar to and inspired by TinyRPC [May et al. 2007], except the latter does not provide security and has a different semantics that are not as expressive as our approach.

TeenyLIME allows application programs to access an abstract “tuple space” that is the union of tuple spaces on the local node and the immediately neighboring nodes [Costa et al. 2007]. This provides an alternative to RPC for uniformly accessing remote and local data. However, interaction with the middleware is by way of a dedicated API; there is no attempt to provide a true RPC mechanism. Also TeenyLIME does not address issues of access control.

Secure Middleware for Embedded Peer to Peer systems (SMEPP) is a general framework for creating security sensitive applications from a distributed network of embedded peers [Brogi et al. 2008]. SMEPP Light [Vairo et al. 2008] is a reduced version of SMEPP to address the resource constraints of wireless sensor networks. SMEPP Light provides a publish/subscribe communication model using directed diffusion to distribute “events” to all subscribers and symmetric key cryptography to provide con-
The presence of multiple security domains. However, SMEPP Light is not integrated into a programming language and does not provide a remote procedure call mechanism. Furthermore, SMEPP Light only supports a simple model of access control based on group membership.

High level macroprogramming languages such as Kairos [Gummadi et al. 2005], Regiment [Newton et al. 2007], and even Flask [Mainland et al. 2008] provide a way to program the entire network as a single entity. These systems attempt to hide not only the inter-node communication from the programmer, but also the entire node level programs. SpartanRPC operates at a much lower level and also, unlike these macroprogramming systems, addresses access control issues in networks containing multiple security domains.

Whole network programming of wireless sensor networks has also been investigated using mobile agents in systems such as Agilla [Fok et al. 2009] and Wiseman [González-Valenzuela et al. 2010]. However, like the macroprogramming systems mentioned previously neither of these systems address issues related to access control in the presence of multiple security domains.

REFERENCES


RPC Authorization in Sensor Networks

A:29


