A Lightweight Policy System for Body Sensor Networks
Yanmin Zhu, Member, IEEE, Sye Loong Keoh, Member, IEEE, Morris Sloman, Member, IEEE, and Emil C. Lupu, Member, IEEE

Abstract—Body sensor networks (BSNs) for healthcare have more stringent security and context adaptation requirements than required in large-scale sensor networks for environment monitoring. Policy-based management enables flexible adaptive behavior by supporting dynamic loading, enabling and disabling of policies without shutting down nodes. This overcomes many of the limitations of sensor operating systems, such as TinyOS, which do not support dynamic modification of code. Alternative schemes for adaptation, such as network programming, have a high communication cost and suffer from operational disruption. In addition, a policy-driven approach enables fine-grained access control through specifying authorization policies. This paper presents the design, implementation and evaluation of an efficient policy system called Finger which enables policy interpretation and enforcement on distributed sensors to support sensor level adaptation and fine-grained access control. It features support for dynamic management of policies, minimization of resources usage, high responsiveness and node autonomy. The policy system is integrated as a TinyOS component, exposing simple, well-defined interfaces which can easily be used by application developers. The system performance in terms of processing latency and resource usage is evaluated.

Index Terms—Policy-driven management, policy system, body sensor networks, adaptation, authorization, access control.

I. INTRODUCTION

Body sensor networks (BSNs) [6], [34], [36] have recently been employed for various personal applications, in particular in the healthcare domain [30]. In a BSN, biomedical sensors are attached to, or possibly implanted in, patients to continuously monitor physiological parameters for health management. Abnormal events indicating coronary problems such as high heart rate or blood pressure can be detected and reported to a doctor for immediate medical action. Such BSNs are particularly suitable for post-operative care in hospitals and for treatment of chronically ill or elder patients at home.

There is typically little functional redundancy between the nodes in a BSN compared to a large-scale sensor network, e.g., for environment monitoring where most nodes perform similar functions and have the same sensors. BSNs exhibit several unique requirements when compared to traditional sensor networks. First, sensors in a healthcare BSN often need to adopt their behaviors to changes in the patient’s medical condition or activity. The sensors should be configured according to reflect such changes. For example, when a patient is suspected to have a cold, the temperature sensors should become more sensitive and report more temperature data for better monitoring. In some situations, the doctor may want more detail on blood sugar level of the patient, so glucose sensors which have been turned off for power conservation must be enabled.

Second, security is essential for practical use of BSNs in healthcare where privacy concerns about access to a patient’s health condition data can be important. Preventing unauthorised access to actuators, such as insulin or other drug pumps, may be even more critical as this involves the patient’s safety. However, there is a need for different types of medical staff to have differentiated privileges for access to a patient’s sensors and actuators. There is also the need to protect against malicious attackers, particularly for celebrities and other high profile patients. Thus, only authorized access to body sensors should be permitted, for both accessing data and performing actions, while unauthorized access must be blocked.

Little existing work fulfils the above requirements. TinyOS [4], the de facto standard operating system for sensors, does not support dynamic modification of code once the program is deployed. Thus, it is difficult for a sensor to adapt its behaviour – a typical solution is to shut down the network and reprogram the sensors. Most network programming protocols [18], [23], [25] require the whole program code image to be disseminated to the sensors through wireless communications.

Fig. 1. The policy service architecture for body sensor networks. Each sensor maintains its own policies and implements both PDP and PEP.
This not only incurs large wireless communication overhead, which is the main source of power consumption on sensors, but also interrupts the operation of the network. For data confidentiality, symmetric cryptography has been used in sensor networks. Key management schemes [15], [17], [27] ensure that sensors trying to communicate with each other share common keys. However, such approaches achieve only in-transit data confidentiality but do not perform access control on individual nodes.

Policy-driven management has been widely recognized as an important technology for managing distributed systems [35]. By separating policies from the system implementation, a policy-driven system can adapt to changes by dynamically changing policies. In addition, fine-grained access control can be realized by making use of authorization policies. We have developed a policy-based system [22], [30] for pervasive healthcare, which uses a PDA as a coordinator that provides functions, such as discovery of sensors, event routing and external remote communication. It uses a policy system called Ponder2 [3] which runs on a relatively powerful PDA hosting a java virtual machine environment. Sensors are treated as passive, managed objects, polled at regular intervals for readings.

Sensor nodes are becoming increasingly powerful [33] and can implement a policy system to support intelligent sensing services. For example, sensors can generate events indicating thresholds have been exceeded. Moreover, they can adapt their behaviors in response to context changes or changes in application requirements. For example, a sensor may adapt to the current activity of the user. Direct interaction between sensors becomes possible and at the same time, access to sensor resources is regulated by authorization policies.

Various programming paradigms have been explored to cater for the severe resource constraints and potentially large-scale of distributed sensor networks. Declarative languages [7], [12] provide high-level programming models, which can capture application semantics in a more natural way. This reduces development time compared to programming using languages such as nesC. Although such declarative languages are suitable in application scenarios like data acquisition, they are not applicable for adaptation or reconfiguration. Both declarative languages and low-level programming languages provide no support for adaptation to future unknown changes. However, policy-driven approaches provide a flexible approach to reconfiguration and adaptive behavior.

This paper presents the design, implementation and evaluation of Finger, an efficient policy system running on sensors. This system supports interpretation and enforcement of both obligation policies, which are event-condition-action rules that perform an action in response to an event, and authorization policies, which define what resources or services a subject can access on a target sensor. As illustrated in Fig. 1, each sensor manages its own policies and implements both a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP). A PDP interprets policies and makes policy decisions. Following the decision made by the PDP, the PEP enforces the policy, i.e., it invokes the action specified by the obligation policy, or permits/denies a subject from performing a requested action. In essence, Finger supports a considerably simplified version of the Ponder2 language for policy specification [3]. The effective simplification makes the policy language suitable for processing on resource-constrained sensors. Compact design and implementation on TinyOS makes Finger efficient and responsive while introducing modest resource consumption overheads. Note that the preliminary work of this research has been reported in [37]–[38]. This paper provides more complete design and analysis of the policy system.

The paper is structured as follows – Section II describes the system background, gives a motivating example, and highlights the attack model and design objectives. In Section III we look at the architectural design, followed by implementation details in Section IV. We then present performance evaluation results in Section V. Finally, Section VI discusses some limitations of the current implementation. We review related work in Section VII, and conclude the paper and discuss future research directions in Section VIII.

II. BACKGROUND AND DESIGN OBJECTIVES

This section gives a brief overview of the system background and a motivating example that shows the need to exploit policy-based management for BSNs. We discuss the attack model considered in the design of Finger and summarize design considerations.

A. Body Sensor Networks

A BSN consists of a controller, body sensors, and possibly dynamic nodes. The controller manages the whole network and can be a PDA or Smartphone, which is relatively powerful, compared to sensors. Body sensors are attached on the body or implanted within the body for monitoring various healthcare parameters. A body sensor is subject to severe resource constraints as it has small memory and limited processing capability. Dynamic nodes represent medics, such as nurses and doctors, which may intermittently interact with a patient BSN for short periods to obtain readings or change settings.

Finger is intended for the hardware platform of body sensor node [1], equipped with a TI MSP430F149 microcontroller – see Fig. 2. This microcontroller has a 16-bit RISC processor and works at 16MHz. It has 60KB, read-only program memory for executable code and 2KB writable data memory which serves as a data stack. A CR2430 button Li-battery is used to power a node. The sensing board integrates sensors such as temperature sensor and accelerometer, which can be
connected to the processing board. A Chipcon CC2420 radio transceiver supports communication with other nodes using the IEEE 802.15.4 protocol [2] at 2.4G Hz. The maximum communication bandwidth is 250 kbps.

TinyOS is an operating system designed for resource-constrained sensors. A TinyOS application program is a graph of software components with well defined bi-directional interfaces. Its event-driven execution model is effective for energy-efficient design. The processor switches to the idle mode when there is no task to perform, which consumes significantly less power compared to the active mode. TinyOS is a very simple operating system and is thus suitable for memory-constrained sensors. Complex OS functionalities such as dynamic memory allocation are not supported.

B. Motivating Example

Consider a simple healthcare scenario where a BSN, attached to a user for on-body monitoring, includes a controller, a temperature sensor and an accelerometer sensor which can be used to determine user activity, e.g., walking or sitting. The controller typically performs important tasks such as data aggregation, policy deployment and security management.

To detect the activity of the user, an accelerometer sensor starts a timer and regularly (e.g., every 5 seconds) reads accelerometer data. The timer frequency is an important parameter that determines the ability to detect activity changes. A higher frequency allows the sensor to detect more rapid movement changes but then the sensor consumes more energy. It is intuitive that when the acceleration is over a certain threshold, it is likely that the user is starting to walk. Thus, a sensor should increase its measurement frequency so that more accurate estimation can be obtained for more accurate estimation. When the acceleration becomes smaller than the threshold, it is probable that the user is sitting or standing. Thus, the measurement rate can be reduced for energy conservation. Two obligation policies can realize such adaptation.

\[
\text{oblig on accel\_event (acceleration)}
\]
\[
(1) \text{ do adjust\_measurement\_interval (1s)}
\]
\[
(2) \text{ do adjust\_measurement\_interval (5s)}
\]

The measurement interval can thus be re-configured according to application requirements by updating the two policies.

A doctor may decide that it is useful to study the relation between body temperature and user activity, so the temperature sensor should record the body temperature when activity changes occur. However, this function has not been pre-programmed on the sensor. But, this could be achieved by deploying new policies. The accelerometer sensor should notify the temperature sensor of new activities. Thus, it needs an obligation policy for this.

\[
\text{oblig on new\_activity\_event (activity)}
\]
\[
(3) \text{ do send\_event (new\_activity\_event, temperature\_sensor)}
\]

The temperature sensor accordingly needs an obligation policy to record the current body temperature when a new activity occurs.

\[
\text{oblig on new\_activity\_event (activity)}
\]
\[
(4) \text{ do record\_temperature}
\]

The accelerometer sensor raises an event on the temperature sensor and this should be subject to authorization control. Otherwise, unauthorized nodes may send arbitrary events and do harm to the node. Thus, the temperature sensor needs an authorization policy to permit the event-raising action for the accelerometer sensor.

\[
\text{auth+ subject acceleration\_sensor}
\]
\[
(5) \text{ target temperature\_sensor}
\]
\[
(6) \text{ action raise\_event}
\]

The controller often needs to re-configure the sensor network by changing policies on sensors. Policy management tasks include loading, unloading, enabling and disabling policies. Thus, the accelerometer sensor should have an authorization policy to allow the controller to change its policies.

\[
\text{auth+ subject controller}
\]
\[
(6) \text{ target acceleration\_sensor}
\]
\[
(7) \text{ action manage\_policy}
\]

The temperature sensor should also have a similar authorization policy.

\[
\text{auth+ subject controller}
\]
\[
(7) \text{ target temperature\_sensor}
\]
\[
(8) \text{ action manage\_policy}
\]

This example demonstrates that sensors must frequently adapt to both context changes and application requirements. They also need to cooperate with each other to achieve application goals. Obligation and authorization policies provide a flexible and easily modified means of specifying what interactions must be performed and what interactions are permitted.

Note that the subject and the target in an authorization policy can be a role in a domain hierarchy rather than a hard coded node ID [3]. This allows policies to be defined for groups of nodes rather than just individual ones so policies do not have to be modified when sensors are added or removed from the group.

C. Attack Model

Body sensor networks make use of short range wireless communication and are vulnerable to various attacks. The paper considers the following attacks.

**Eavesdropping.** Wireless transmissions are susceptible to passive eavesdropping. Sensitive data may be therefore accessed by unauthorized users, which compromises data confidentiality. Therefore, data to be transmitted over the wireless channel should be encrypted to prevent passive eavesdroppers.

**Active attacks.** Other than passive listening, active attackers may alter or replay messages causing serious harm to the patient. For example, replaying a message to a drug pump can cause severe drug overdose. Authorization and authentication mechanisms are needed to prevent active attacks.

**Impersonation.** Authentication is essential to prevent an attacker from impersonating a legitimate user, such as a medic,
in order to modify the behavior of the BSN and hence harm the patient.

D. Design Objectives

There are a number of challenges in implementing a policy system on small sensors. It is crucial to make efficient use of the limited resources such as small memory. Policy based systems such as Ponder2, or XACML are inappropriate for resource constrained sensors. It is impractical to pre-load all required policies so dynamic management of policies with each node responsible for maintaining and managing its own policies is required, i.e., it must be possible to load, unload, enable and disable policies but at the same time protect these important operations from unauthorized access.

The following design objectives were considered important in the implementation of Finger:

- **Dynamic management of policies.** When a node is discovered and joins a network the policies appropriate to its role within the network must be dynamically loaded [30] and possibly modified at a later time to enable context adaptation.
- **Minimum memory footprint.** The policy service memory requirements must be minimized.
- **Responsiveness.** Policy software should introduce minimal processing latency, as some applications may require critical response times.
- **Well defined APIs.** These are needed to enable easy development of new policy-based applications.
- **Energy efficiency.** Policies should introduce minimal processing overheads in order to minimize power consumption.

### III. DESIGN OF FINGER

This section discusses the design of Finger. The overview of Finger is first presented and then the authentication protocol is described.

A. Overview

The architectural overview of Finger is depicted in Fig. 3. The core model comprises two components – the Obligation Interpreter (OI) and the Authorization Interpreter (AI) for interpreting and enforcing obligation policies and authorization policies, respectively. Both the OI and the AI provide a repository for storing policies but the dynamic management of stored policies is implemented in an independent component that provides policy management actions. This enables requests to manage policies to also be governed by authentication and authorization checks.

The OI receives events generated from the internal TinyOS components controlling sensors, e.g., temperature sensors, as well as external events received as incoming messages from the network. It can perform actions on software or hardware components within the node. An action on a software component could generate an event or message to be sent out to the network. On receiving an event, the OI searches the policy repository for all policies matching the event type. It then checks whether the condition part of the corresponding obligation policy evaluates to true and if so, the OI invokes the specified action through the Action interface.

All incoming requests from external nodes must be authenticated and authorized. Incoming requests could be either an incoming event or a request to perform an action on a hardware...
or software component, including policy management operations. Incoming requests are of the form \(<\text{subject}, \text{action}, \# \text{ of paras}, \text{paras}>\). The Request Manager (RM) receives incoming requests and authenticates the requesting subject by invoking the Authentication Manager (AM). The design of this module is discussed in the next subsection.

If the subject is successfully authenticated, the request is passed to the AI via the \texttt{ProcessRequest} interface. The AI then searches its authorization policies. If a policy for the subject and the requested action is found, the associated condition is checked and if positive, the associated action is then invoked. For incoming, authorized events, the associated action is treated as raising an event. The first parameter of the request indicates the event type and the second one indicates the event value. The AI invokes, through the \texttt{RaiseExternalEvents} interface, the \texttt{ExternalEvents} component, which immediately triggers the OI.

As previously discussed, the code size must be minimized. There is considerable common functionality between the OI and the AI such as the way they both evaluate constraints. Thus, we factor out the implementation of condition evaluation and make it as an independent component, so this common code can be shared by both the OI and the AI.

B. Authentication Protocol

To make access control effective, the target node must authenticate the requesting node before making the authorization decision. The requesting node provides its \(<\text{ID}, \text{role}>\) in messages to a target node. The AM must decide whether the requester really possesses the ID and whether it has the claimed role. In Fig. 4, a simple example is illustrated. The example BSN consists of a controller and four sensors. Sensor 3 sends a request to sensor 4 and sensor 4 wants to authenticate sensor 3.

We have developed an efficient authentication protocol based on the Diffie-Hellman (DH) key agreement. Both public-key and symmetric cryptography are employed. In the initialization phase, each sensor \(i\) generates a secret \(s_i\), and computes a keyshare \(p_i\) based on its secret, \(p_i = g^{s_i}\). It is computationally infeasible to recover the secret, given the keyshare. The sensor obtains the group key from the controller, and exchanges its keyshare with the controller. The channel by which the group key is obtained and the keyshares are exchanged is physically secure e.g., by plugging it into the controller’s USB port.

The controller creates and maintains a membership list of node ID, role, and keyshare. Table I shows the content of the table for the example. Using the group key, the controller can periodically publish the membership to all members in the network whenever there are changes in the membership. The controller encrypts the membership list only once for each release and this only incurs a single broadcast transmission. All the sensors in the network can decrypt the membership list. However, this is based on the assumption that nodes which have been admitted into the network do not behave maliciously by spoofing the membership list.

With the membership list, a pair of sensors \(i\) and \(j\) can then establish a pairwise shared key \(K_{ij}\). Sensor \(i\) computes the shared key as follows,

\[
K_{ij} = (p_j)^{s_i} = (g^{s_j})^{s_i} = g^{s_is_j}.
\]

Sensor \(j\) can compute \(K_{ij}\) in a similar way. The group key is renewed whenever a member is detected to have left the network or been compromised, or when it has been used for an extended period of time. When renewing the group key, the controller sends the new group key to every member individually. The new key is encrypted using the shared key of the controller and each member.

With the pairwise shared key, we develop a challenge-response exchange procedure for a sensor to authenticate a requesting node. Consider the scenario in the example that sensor 4 wants to authenticate sensor 3. The process is initiated by sensor 3 sending a request to sensor 4. Sensor 4 can compute the pairwise shared key \(K_{13}\) according to (1). Sensor 4 then challenges sensor 3 by sending its nonce encrypted with the shared key \(K_{34}\). Sensor 3 should decrypt the encrypted nonce and respond with a \(<\text{nonce +1}>\) encrypted with the shared key. Sensor 4 authenticates sensor 3 if the response content is indeed \(<\text{nonce +1}>\).

The three-way handshake is costly since it introduces two additional transmissions. This not only wastes power but also introduces overall latency for request processing. We propose a ticket technique to avoid a three-way handshake each time a request is processed. After successful authentication, sensor 4 creates a ticket which is essentially a random number and sends it to sensor 3. Later, each time sensor 3 requests an action on sensor 4, it increases the ticket by one and appends it to the request. Sensor 4 decrypts the request and checks the ticket. If the ticket is indeed the ticket plus one, it is able to ensure that the requesting node is sensor 3. Such a ticket is renewed, through the target starting a new challenge-response procedure, after it has been used for an exceeded period.

Note that the exponential notations in the DH-based protocol are for conceptual description only. Exponential computation with big integers is too computationally expensive to be implemented on body sensors. Instead, we use elliptic curve cryptography (ECC), as discussed in detail in Section IV-E.
IV. IMPLEMENTATION

In this section, we discuss implementation details. We have implemented Finger using nesC [16] with TinyOS v1.15 on the BSN hardware platform introduced in Section II.

A. Policy Specification

We have to scale down the complexity of policies since small sensors cannot afford to process complex policies used in traditional distributed systems. We designed a simple and efficient policy language with a syntax suitable for efficient processing by body sensors yet it is expressive and able to fulfill most management needs of sensor networks. The syntax of policies is given in Table II.

An obligation policy specifies the event, the action and the condition under which the action must be performed. Note that an action is also associated with several parameters to be used when this action is invoked. An authorization policy defines the subject, the target, the action and the condition. A subject or target is a role in a domain hierarchy. Details of how nodes are discovered and assigned to roles are described in [30]. The policies used in the motivating example can be specified as shown in Table III. For policy (1) the first “0” indicates it is an obligation policy and “#1” is its ID. The obligation-triggering event “1?” refers to the acceleration event. The condition 30 refers to the acceleration context variable “1^”. The obligation has an action of “1” to adjust measurement interval and the action takes a parameter of 1.

B. Memory Organization

It is important to manage the sensor’s limited memory. Since TinyOS does not support dynamic memory allocation, we need to allocate space to hold the maximum number of policies statically. For each type of policy, we maintain two lists: one for the available policies and the other for vacant cells. Each time a policy is loaded, a vacant cell is obtained from the vacant list and inserted into the available list. Conversely, when an existing policy is removed, its cell is returned to the vacant list.

It is difficult to predict the maximum number of policies that a sensor will need. However, as resources are limited we assume this is likely to be in the order of 10 to 40. Our approach is to estimate the number of policies needed for the current application and allow for twice that number for future adaptivity.

C. Dynamic Policy Management

Dynamic management of policies is crucial to the adaptation ability of sensors. As discussed, management operations are treated as regular authorization requests and are controlled by authorization policies. Authorized management requests result in performing an action on the ManagePoliciesActionM component.

The ManagePoliciesActionM implements all policy management operations and provides the Action interface to the AI. The first parameter of the action is used to indicate the type of policy management – loading, unloading, enabling or disabling. For loading a policy, the second parameter is a string containing the policy text. For the other three types, the second parameter indicates the ID of the policy to be operated. To load a policy, the management component parses the policy text by invoking the PolicyTextParser component. Through the PolicyControl interface, the resultant parsed policy is passed to the AI or the OI, and then inserted into the available policies. The ability of enabling and disabling policies adds flexibility to the system and reduces communication cost.
D. Trigger and Dispatcher

We exploit the design pattern of trigger and dispatcher to support the libraries of event sources and actions. Each event source causes the OI to interpret an obligation policy for this event. Following the trigger design pattern, we separate the library of event sources from the OI. Each event source is a TinyOS module and supports the EventSource interface to activate a policy in the OI. The attributes of this event, such as id and value, are passed to the OI through interface parameters.

For both the OI and the AI, a policy may result in the invocation of an action identified in the policy. Following the dispatcher pattern, we separate all actions from the OI and the AI. Each action is a TinyOS module providing the Action interface. By using the same interface, the interpreters are able to determine the action to invoke at run time based on the action identifier provided by the policy.

The patterns not only make the design of Finger modular, but also extensible to facilitate easy application evolvement. Adding event sources or actions can be simply achieved by coding TinyOS modules supporting the interfaces of EventSource and Action. The implementation of the OI and the AI stays intact.

E. Authentication Manager

To overcome the heavy cost of exponential computation in the traditional DH key agreement protocol, we exploit Elliptic Curve Cryptography (ECC) to implement the authentication protocol. ECC public-key cryptography has much shorter key length and less computation overhead than RSA. We slightly modified the TinyECC [26] package to migrate it to the body sensor node platform.

We implemented the authentication protocol using point multiplication in ECC. First, a base point is chosen and made publically known to all sensors. Next, each sensor i generates a random point as its secret $s_i$. The keyshare $p_i$ of sensor i is computed by multiplying secret $s_i$ with the base point $G$, $p_i = s_iG$. To compute the pairwise shared key with of sensor $i$, sensor $j$ multiplies its own secret with the keyshare of sensor $i$, $K_{ij} = s_j(p_i) = s_j s_iG$. In a similar way, sensor $i$ can compute the shared key with $j$, $K_{ji} = s_i(p_j) = s_i s_jG = K_{ij}$. Although a point on an elliptic curve is two dimensional and represented by $(x, y)$, only the $x$ value is used to generate the shared key. The $x$ value is hashed to produce a 160-bit key as the pairwise shared symmetric key.

We adopted Skipjack, implemented in TinySec [21], for symmetric encryption with a 160 bit key length. Skipjack is a block-cipher with the block size of 8 bytes. We use the Cipher Block Chaining (CBC) operation mode with non-repeating Initialisation Vector (IV). The battery level or sensor readings can be used as the seed of a pseudo-random number generator to generate the initial IV.

F. Exposed APIs

Finger provides simple application programming interfaces (APIs) to application developers. The components of Finger are packaged as a single TinyOS configuration component, called FingerC, which hides the implementation details of Finger from developers. Three TinyOS interfaces are exposed as shown in Fig. 5.

The detailed definitions of the interfaces are shown in Table IV. FingerC should be included in the application configuration to use policies. The Main module of the application wires its StdControl to that of the policy system, which initializes the embedded components with Finger. Note that, the policy system provides only a small set of basic event sources and actions, such as temperature event and data report action. To extend the functionality, application-specific event sources and actions can be developed. Event sources should connect to the EventSource interface to trigger obligation policies. Similarly, all actions to be regulated by authorization policies should connect to the Action interface.

V. PERFORMANCE EVALUATION

We have conducted a series of experiments to measure the performance of Finger in terms of memory overhead and processing delay metrics.

A. Methodology and Setting

To facilitate performance measurements, we developed a simple TinyOS application SimApp making use of Finger. This application implements an event source related to acceleration, and two actions which toggle the red light and the green light, respectively. An obligation policy is deployed for this event, which specifies that the green light be toggled when the acceleration is larger than a threshold. It also has an authorization policy which controls access to the red light action.

We investigate memory overhead solely introduced by Finger. More specifically, we look at the ROM and RAM sizes. The ROM stores the program executable and the RAM serves as the run-time data stack. To this end, we develop a simple application (SimApp) that makes use of Finger. SimApp is so simple that it adds little memory overhead. Its code size is approximately equal to the size of Finger plus the TinyOS basics and the communication subsystem. In order to separate the memory usage of different component and modules, we use the script called module_memory_usage available in TinyOS at contrib/SystemC/scripts. This script shows the ROM and RAM usage of every single module in an application binary code.

The ECC and TinySec libraries require a considerable amount of memory. In order to evaluate the core policy system, which solely interprets and enforces obligation and authorization policies, we used two versions of the policy system,
Finger(w) and Finger(w/o) – with and without authentication, respectively.

We exploited the timing facility provided by TinyOS to measure processing delays, and all measurements were directly made on the sensor running the policy system. We developed a TinyOS module MeasureTimeM for delay measurement. It employs the system interface LocalTime provided by the TimerC hardware module to read the current local time on the sensor. MeasureTimeM records timestamps and sends them back to the PC for delay calculation. This guarantees that no other delays are included in calculated processing delays. Each of the following result is averaged over 20 independent measurements.

All optimization switches of TinyECC were turned on for minimization of processing latency. However, some of the switches can be turned off to trade memory consumption for cryptography performance. Note that a body sensor node has only 2K bytes RAM so it cannot host Finger(w), consequently we instead had to use a Tmote Sky node for experiments with Finger(w). A Tmote node shares the same processor with a body sensor, but it has a larger RAM size (10K bytes).

B. Memory Footprint

The resultant memory size of Finger is dependent on the maximum number of policies deployed. All the following measurements are based on a maximum number of 20 policies.

We compiled SimApp into TinyOS executable on the body sensor node platform. Table V shows the memory usage breakdown of SimApp, where ROM and RAM are shown separately. The executable without authentication occupies 16.11Kbytes of ROM and 1.06Kbytes of RAM, and the one with authentication occupies 31.77K bytes of ROM and 2.88K bytes of RAM.

Finger occupies a total of 20.65Kbyte ROM and a total of 2.35Kbytes RAM. To gain insight into the Finger memory usage, we show the values for several major modules separately in Table V. It is apparent that the authentication module, based on TinyECC, uses 15.66Kbytes of ROM and 1.82Kbytes of RAM and dominates the memory usage of Finger. Other core modules, such as AI, OI and Parser, are efficiently implemented and use less than 1Kbyte ROM and 0.15Kbyte RAM.

The small memory footprint demonstrates that the policy system is feasible for the majority of available sensor platforms.

C. Processing Delay

We examine various processing delays introduced by the policy system. The experiments were conducted with 7 deployed obligation policies and 8 deployed authorization policies. The delays for processing obligations and authorizations are shown in Table VI.

The obligation interpretation delay is measured from the time the OI is triggered by an event source to the time the OI invokes the corresponding action. The authorization interpretation delay is from the time when the RM passes an incoming request to the AI to the time when the AI invokes the associated action. The time required by the specific action is not taken into account since different actions may incur substantially different times. From the table we can see that it takes as little as 62μs to process an obligation policy and 81μs to process an authorization policy. The latency of raising an external event on a sensor which includes processing an authorization policy and then an obligation policy is 140μs. We also measured the latency caused by policy management. It takes 375μs to load an authorization policy. Thus, it takes in total 437μs to process a loading-policy request and load the policy.

We also evaluated delays for various cryptographic operations in the authentication process. With TinyECC, it takes an average of 9530 ms to encrypt a 52-byte message, whose content are randomly generated, and 5281ms to decrypt the encrypted message. With the Skipjack library, it takes significantly less time, 150μs to encrypt the same message and 90μs to decrypt the encrypted message. This big difference shows that it is essential to use shared keys for most encryption. Based on the shared key, the authentication can be performed more efficiently using the symmetric Skipjack cryptography. The ECC cryptographic delays are acceptable since it is rarely used, as our system uses the ticket technique.

D. Security Analysis

Finger enforces an array of security countermeasures, including authentication, authorization, encryption, and time stamping to guard against the possible attacks discussed in Section II-C.

Based on the DH key agreement, each pair of sensors establishes a shared key to encrypt all data transmitted over the air between the nodes. Even if an eavesdropper can record the complete packet transmitted between the two nodes, it is extremely difficult for the eavesdropper to recover the original content inside the encrypted packet.

Active attackers may try to alter data or replay a previous session. To guarantee data integrity, a data digest is concatenated with the original data before the entire data are
encrypted. By computing a digest of the received data and comparing it with the received digest, a receiver is able to determine if data has been altered. To prevent replay attacks, a timestamp is appended to the original data before encryption. By checking the freshness of the received packet, the receiver can determine if the received packet has been replayed. Time stamping depends on the availability of time synchronization, but since a body sensor network is of relatively small scale, it is not difficult to synchronize the clocks of the sensors in the network.

Finger employs authentication to prevent impersonation attacks. A subject sensor or dynamic node must be authenticated and authorized to invoke an operation on the target sensor. An impersonating sensor would not have the private key share of the sensor that it is trying to impersonate, so would not be authenticated.

Authorization policies specify fine-grained access control to protect target sensors, for example, from careless operations.

### E. Ease of Use Analysis

For the policy system, it is difficult to quantitatively evaluate the degree of ease of use. There are three factors pertaining to ease of use: First, the policy language described in this paper is a declarative language as is DSN [12], although Finger is much simpler. DSN has demonstrated the use of a declarative language can considerably reduce the number of lines of program code required to program a variety of sensor protocols. Finger shares similar advantages in that by using declarative policies the program logic can be simplified and hence the program length can be reduced. We contend that designing the adaptive strategy with low-level control actions are programmed in the conventional language such as nesC.

Finally, user requirements may change over time and the working environment may also change. This means that it is very difficult, if not impossible, to hard-code all decisions beforehand. Instead, one needs a way to change the behavior of a sensor in order to adapt to the changes of user requirements or the current context. The policy system enables such adaptation or reconfiguration by dynamically adding/removing policies or enabling/disabling existing policies, without re-programming the entire sensor network or shutting it down. With DSN, a complete program must be installed on a node so adaptation requires re-installation.

### VI. Discussion

Currently, Finger supports a condition referring to a single context variable for both obligation and authorization policies. Realistic applications may need conditions on multiple context variables. A full constraint is formed by connecting several clauses using logical operators. Finger can be extended to support setting a condition on multiple variables. This complicates the design of Finger and requires additional space to store the condition. Moreover, additional latency will be introduced for condition evaluation.

The maximum number of policies is limited by the data memory size. If only a small number of policies can be accommodated, a sensor will have limited adaptability and self-configuration abilities. This can be mitigated by exploiting the external flash memory which can be up to 2M bytes on new sensors [5]. Although, external flash memory can accommodate a large number of policies, there are several issues with using flash memory. First, accessing flash memory consumes considerable energy. Second, flash memory does not support random access and must be read and written in blocks. This could make it impractical for storing current policies but alternative sets of policies could be stored in flash and swapped with current policies for predefined adaptation.
Limitations of Finger. Note that the policy language provided in Finger is not a general purpose programming language. Loading, unloading, enabling and disabling policies on a sensor cannot be used to change the behaviours of the node at arbitrary granularity. Instead, constrained changes of the sensor’s behaviour can be performed in terms of existing events and actions, or changing authorization rules with respect to a subject. In the current implementation, Finger does not support the dynamic addition or removal of actions and events due to the TinyOS limitation of not supporting dynamic code modification. This results in some limitations on the adaptive capability. However, it is possible to compose new actions which are a combination of existing implemented actions.

VII. RELATED WORK

Finger is motivated by many successful policy systems for traditional distributed systems. Policy-based management of networks and distributed systems [10], [35] has received significant attention. It has been applied to security management and privacy preservation [20]. Considerable effort has been applied to develop expressive languages for specifying policies [9], [13], [19], [29]. Nevertheless, these languages are not suitable for sensor networks due to resource constraints. For example, SecPAL [9] developed at Microsoft Research is a language for expressing decentralized authorization policies for large-scale computing environments. The policy language developed for Finger is a very simplified version of Ponder2. The idea of policies for sensor network management has been investigated in [10], which uses policies to enable end users to detect and handle events, and therefore configure the sensor network. However, the polices are not processed at the sensor level, but follows the traditional policy-driven approach.

Network reprogramming [18], [23] has been used to realize sensor network adaptation to application requirements. Basically, the image of the new application executable has to be conveyed to every sensor in the network. The image file is temporarily stored on external Flash memory. After a sensor has received the complete image, it restarts itself and reads the new application binary from the Flash to the main memory. Deluge [18] has techniques to control communication overhead and to reduce reprogramming latency. However, network reprogramming still incurs very high energy overhead due to the large volume of data communication. In addition, the current operation of the network will inevitably be interrupted. This is unacceptable for mission-critical applications.

Programming sensor networks is notoriously difficult. Thus, a large body of work explores high level programming models. TinyDB [31] considers the main task of a sensor is to request relevant data and provides a declarative, SQL-like query language. DSN is a declarative platform for sensor networks [12] which focuses on both data acquisition and network protocol processing. It consists of a declarative language Snlog, compiler and runtime implementation, based on Datalog [7]. With DSN, programming efficiency is improved compared with programming in nesC.

Mate [24] is another technology for reconfiguring a sensor network and reduces the communication cost of completely reprogramming the sensor network. Every sensor is running a virtual machine. Applications can be developed using a simple high-level language. Such applications can be run by a sensor through the virtual machine interpreting the program script.

Finger adopts a policy-based approach rather than using a full declarative language or scripts. Policies provide a much more constrained form of ‘programming’ the adaptability and reconfiguration required for sensor networks. When changing behaviours of a sensor in order to adapt to various changes, one can simply change the set of policies in the sensors, without changing the entire program or script. Low-level control actions can still be implemented using the traditional nesC language. The developers can leverage on the advantage of policy driven management by easily including the Finger component in their application programs. Furthermore, policies can be transformed into logic formalism for analysis to detect conflicts [29], which is not practical for general script languages. In addition, security of the network has also been considered by enforcement of authorization policies, which is not taken into account by network programming or virtual machines.

Cryptography provides the foundation for sensor networks to achieve security goals. Many symmetric cryptography schemes [21] have been studied since they are affordable for such resource-constrained platforms as sensors. Data confidentiality in sensor networks using symmetric cryptography cannot be achieved without key management [14], [17], [28]. However, existing key management schemes cannot fulfill the fine-grained access control requirement in BSNs where an efficient authentication protocol is required as well as specification and implementation of authorization.

It is important to enforce secure operations in wireless sensor networks. Cryptography is necessary so key management is a central issue. There has been considerable work on managing secret keys in sensor networks. Chan et al. [11] proposed a method for identifying and excluding compromised sensor nodes.

Public-key cryptography provides an attractive infrastructure for network security. However, public-key cryptography is too costly for implementation on resource-constrained sensors. Nevertheless, some studies [26], [32] have shown that it is viable to implement elliptic curve based public key cryptography on sensors, e.g. the TinyECC [26] configurable library. It allows tradeoffs between resource usage and performance — cryptographic performance can be enhanced by using more memory if available.

Little work has explored the realization of policy-driven management for resource-constrained sensor networks. The Self-Managed Cell (SMC) [22] is an early attempt to introduce a policy-based approach for managing wireless sensors. However, it uses the full Ponder2 which requires a Java runtime environment that cannot be accommodated on an extremely resource-constrained sensor. To the best of our knowledge, Finger is the first functional policy system running on sensors, which supports fine-grained policy interpretation and enforcement.
VIII. CONCLUSION AND FUTURE WORK

We have presented a novel policy system called Finger for BSNs. Finger supports efficient on-node interpretation and enforcement of both obligation and authorization policies. It realizes policy-based dynamic adaptation to context changes or changes in application requirements without interrupting the current network operation. Fine-grained access control is also enabled such that sensitive data or operations can be protected against unauthorized access. Performance measurements of Finger indicate that it is viable and practical for resource-constrained BSNs. With Finger, application development can also be accelerated since developers only need to focus on developing event sources and actions, and composing policies. Although Finger has been implemented on the body sensor node platform, it is extensible and can be deployed to many other platforms including Mica2, Telos and TMote Sky.

Future work will extend the current condition model to support more flexible condition constraints and provide an easier to use platform for generating finger policies. It is also necessary to deploy conflict detection and compliance mechanisms [8]. The simplicity of sensor node policies suggests that such techniques can be developed without much difficulty although the emphasis will be on detecting conflicts before loading policies onto nodes.

ACKNOWLEDGMENT

We gratefully acknowledge support from the National Science Foundation of China through grants 60903190 and 60803124, from the Ministry of Industry and Information Technology of China through grant 2009ZX03006-001, from the China 973 Program through grant 2006CB303000, from the National Natural Science Foundation of China through grant 2009ZX03006-001, from the MOE (Ministry of Education of China) through grant 20090303110011, from the AVS (Academic Venture Support Scheme) grant to ZHU (Atmographic Computing Technology), and I. Stoica, “The design and implementation of a declarative sensor network management policy language,” in Proc. 25th IEEE GLOBECOM, 2002.

We would also like to thank L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, “The design and implementation of a declarative sensor network system,” in Proc. ACM SenSys, 2007.

REFERENCES


Yannin Zhu obtained his PhD in computer science from Hong Kong University of Science and Technology in 2007, and his BEng in computer science from Xi’an Jiao Tong University in 2002. He is a lecturer in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. His research interests include ad-hoc sensor networks, vehicular
ad hoc networks, grid computing, and resource management in distributed systems. He is a member of IEEE and the IEEE Communication Society.

Sye Loong Keoh obtained his PhD in computing science from Imperial College London in 2005. He was a Research Associate in the Department of Computing, Imperial College London after graduation through 2008. He is currently a Senior Scientist in the Department of Information and System Security (ISS), Philips Research Lab in the Netherlands. His research interests include policy-based management, security, authentication, trust for wireless sensor networks, and digital rights management (DRM).

Morris Sloman has research interests that include the autonomic management of pervasive and distributed systems, adaptive security management, and privacy and security for pervasive systems. He is on the steering committees for conferences on Policies for Distributed Systems and Networks, Integrated Management (IM), and Network Operations and Management (NOMS). See http://www.doc.ic.ac.uk/~mss for more details and selected papers.

Emil Lupu obtained his PhD in computer science from Imperial College London in 1998. Dr. Lupu is currently a reader in the Department of Computing at Imperial College London. His research interests include network and distributed systems management, design, security, and multimedia and mobility issues in large distributed systems.