2008 14th IEEE International Conference on Parallel and Distributed Systems

An Efficient Policy System for Body Sensor Networks[1]

Yanmin Zhu, Sye Loong Keoh, Morris Sloman, Emil Lupu, Naranker Dulay, Nathaniel Pryce
Department of Computing, Imperial College London

Abstract

Body sensor networks (BSNs) have become a promising technology for healthcare, in which biosensors continuously monitor physiological parameters of a user. Distinct from conventional sensor networks for environmental monitoring, such networks need to be adaptive and can therefore be easily managed. In addition, security becomes a necessity. To this end, we design a policy system that implements policy-driven management on the sensor level. Biosensor adaptability is realized through support of dynamic loading, enabling and disabling of policies without shutting down nodes. In addition, fine-grained access control becomes possible through authorization policies on biosensors. Design and implementation details of the policy system are presented. Experimental results demonstrate that the policy system is viable and can accelerate application development of biosensor networks for healthcare.

1. Introduction

Body sensor networks (BSNs) [1, 2] have recently been employed for various personal applications, in particular healthcare applications [3]. In a BSN, biomedical sensors are attached to, or possibly implanted in, patients to monitor physiological parameters continuously 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 actions. Such BSNs are particularly suitable for post-operative care in hospitals and for treatment of chronically ill or aged 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 adapt their behaviours to changes in the patient's medical condition or activity. The sensors should be configured accordingly 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 with respect to access of 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 biosensors should be permitted, for both accessing data and performing actions and unauthorized access must be blocked.

Little existing work fulfils the above requirements. TinyOS, 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 [4-6] require the whole program code image to be disseminated to the sensors through wireless communications. This not only incurs large overhead of wireless communication, which is the main source of power consumption on sensors, but also interrupts the current operation of the network. For data confidentiality, symmetric cryptography has been used in sensor networks. Key management schemes [7-9] ensure that sensors trying to communicate with each other share common keys. How-

[1] This research is supported by the UK EPSRC grant EP/C547586/1 (BiosensorNet Project).

1521-9097/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPADS.2008.52

IEEE computer society

ever, such approaches achieve only 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 [10]. 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 also be realized by making use of authorization policies. We have developed a policy-based system [3, 11] for pervasive healthcare, which use 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 [12] which runs on the relatively powerful PDA hosting a java virtual machine environment. Sensors are treated as passive, managed objects, polled at regular intervals for readings.

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 *authorisation policies,* which define what resources or services a subject can access on a target sensor. 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 [12]. 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 overhead of resource consumption.

Finger is motivated by many successful policy systems for traditional distributed systems. Policy-based management of networks and distributed systems [10] has received significant attention. It has been applied to security management and privacy preservation [13]. Considerable effort has been applied to develop expressive languages for specifying policies [14-16]. Nevertheless, these languages are not suitable for sensor networks due to resource constraints. The policy language developed for Finger is a very simplified version of Ponder2.

The paper is structured as follows – Section 2 describes the background and gives a motivating example. In Section 3, we present the architectural design,

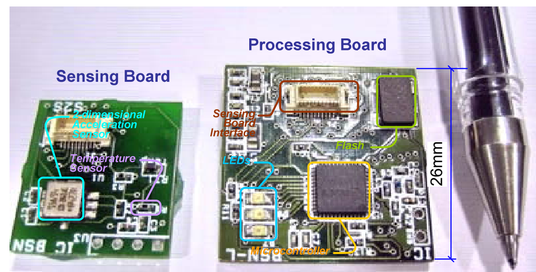

Figure 1: A biosensor node consisting of a processing board and a pluggable sensing board, compared to the size of a ballpoint pen.

followed by implementation details in Section 4. We present experimental results in Section 5, and conclude the paper in Section 6.

2. Background and motivation

A BSN consists of a *controller*, *biosensors,* and possibly *dynamic nodes*. The controller manages the whole network and can be a PDA or a Smartphone, which is relatively powerful, compared to sensors. Biosensors are attached on the body or implanted within the body for monitoring various aspects of body condition. A biosensor 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 biosensor node [17], equipped with a TI MSP430F149 microcontroller – see Figure 1. 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 serving as a data stack. A CR2430 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. With a radio transceiver of Chipcon CC2420, a node communicates with other nodes using the IEEE 802.15.4 protocol 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 reduced power compared to the active mode. TinyOS is a very simple operating system and is thus suitable for mem-

ory-constrained sensors. Complex OS functionalities such as dynamic memory allocation are unavailable.

Consider a healthcare scenario where a BSN is attached to a user for on-body monitoring. In the network, there are a controller, a temperature sensor and an accelerometer sensor which can be used to determine user activity, *e.g.*, walking or sitting. The controller 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 of detecting 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 acceleration data 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.

oblig **on** accel_event (*acceleration*)
(1) **do** adjust_measurement_interval (1s)
if *acceleration* >= 30

oblig **on** accel_event (*acceleration*)
(2) **do** adjust_measurement_interval (5s)
if *acceleration* <= 20

The important parameter of the measurement interval can 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 an activity change occurs. 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.

oblig **on** new_activity_event(*activity*)
(3) **do** raise_event (*activity_event*, *activity*)

The temperature sensor accordingly needs an obligation policy forcing it to record the current body temperature on perception of a new activity.

oblig **on** new_activity_event (*activity*)
(4) **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 raise 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.

auth+ **subject** acceleration_sensor
(5) **target** temperature_sensor
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.

auth+ **subject** controller
(6) **target** acceleration_sensor
action manage_policy

The temperature sensor should also have a similar authorization policy.

auth+ **subject** controller
(7) **target** temperature_node
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.

3. Design of Finger

Several challenges exist 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 unauthorised 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 [3] and possibly modified at a later time to enable context adaptation. The policy system should support such dynamic management of policies.
- **Efficient resource usage.** The memory used by the policy system must be kept to a minimum, as it should be conserved for application codes.

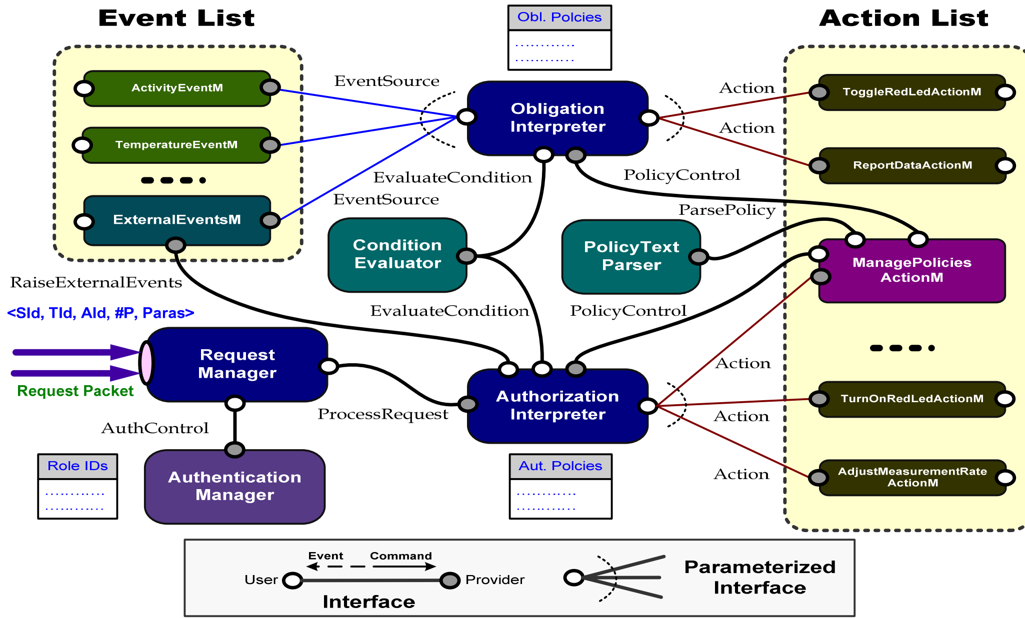

Figure 2. The architectural overview of Finger. Components of Finger are connected by interfaces that define commands and events. The gray end of an interface provides commands and fire events, while the white end uses commands and handle events.

- **Responsiveness.** Policy software should introduce minimal processing latency, as some applications may require critical response times.
- **Programming interfaces.** Clear and easy-to-use APIs are needed to enable rapid development of new policy-based applications for healthcare.

3.1. System overview

The architectural overview of Finger is depicted in Figure 2. The core of Finger comprises two components, *i.e.,* 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. By this means, requests to managing policies on a sensor can be governed by authentication and authorisation checks as normal requests.

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 if the condition 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 are checked for authentication and authorisation. 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 <*subject*, *action, # of paras*, *paras*>**.** The Request Manager (RM) receives incoming requests and authenticates the requesting subject by invoking the Authentication Manager (AM). This module is discussed in the next subsection.

If the subject is authenticated, the request is passed to the AI via the 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 RaiseExternalEvents interface, the *ExternalEventsM* component, which immediately triggers the OI.

As previously discussed, the code size must be minimized. We find that the OI and the AI share much

Table 1: Membership List

Node ID	Role	Keyshare
0	*controller*	g^s_0
1	r_a	g^s_1
2	r_b	g^s_2
3	r_b	g^s_3
4	r_c	g^s_4

in common. In particular, they both evaluate constraints. Thus, we factor out the implementation of condition evaluator and make it as an independent component. By this means, only one copy of code is needed for both the OI and the AI.

3.2. Authentication design

The target node must authenticate the requesting node before making the authorization decision. The requesting node presents the target node the information of <*ID*, *role*>. The AM must decide whether the requester really possesses the *ID* and whether it has the claimed role. In Figure 3, 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 1 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 using the group key. 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,

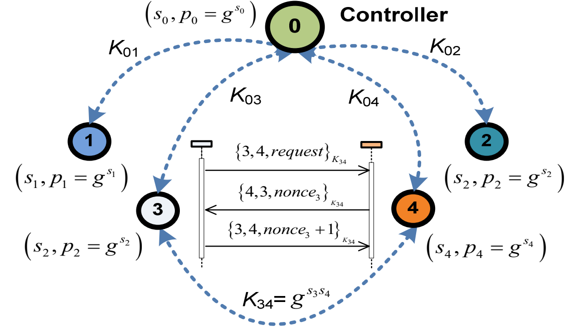

Figure 3: Diffie-Hellman based key establishment, and three-way handshake authentication procedure

$$K_{ij} = (p_j)^{s_i} = (g^{s_j})^{s_i} = g^{s_i s_j}. \quad (1)$$

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.

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_{43} 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 (*nonce* +1) encrypted with the shared key. Sensor 4 authenticates sensor 3 if the response content is indeed (*nonce* + 1).

The three-way handshake is costly since it introduces two additional communications. This not only wastes power but also introduces overall latency for request processing. We propose a ticket technique to avoid three-way handshake each time a request is processed. After the authentication is passed successfully, 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. If the ticket is indeed the ticket plus one, it is able to ensure that the requesting node is sensors 3. Such a ticket is renewed, through the target initiating a new challenge-response procedure, after it has been used for an exceeded period.

4. Implementation details

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 biosensors yet it is expressive and able to fulfil most management needs of sensor networks. The syntax of policies is specified in Table 2. An obligation policy specifies the event, the action and the condition under which this 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 [4]. The policies used in the motivating example can be specified as shown in Table 3. 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".

Table 2: Language syntax summary

```
policy:
    obligation_policy
    authorization_policy
obligation_policy :
    type # pid & eid ? condition ~ action
authorization_policy:
    type # pid & sid @ tid ? condition ~ action
condition:
    contextId ^ comparator value
    contextId ^ range_condition
    always
comparator: one of
    >=  <=  ==  !=
range_condition:
    [ value , value ]
action:
    actionId
    actionId (paras)
paras:
     para, paras
```

Memory Organization. It is of great importance to manage the limited memory of a sensor. 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 recycled back to the vacant list. It is difficult to predict the maximum number of policies that a sensor will need. However as the resources are limited we assume this is likely to be in the order of 10-40. Our approach is to estimate the number of policies needed for the current application and allow for twice that number for future adaptivity.

Table 3: Policy texts used in the example

Policy text	No.
"0 # 1 & 1 ? 1^ >=30 ~ 1 (1)"	(1)
"0 # 2 & 1 ? 1^ <=20 ~ 1 (5)"	(2)
"0 # 3 & 2 ? always ~ 2 ()"	(3)
"0 # 4 & 2 ? always ~ 3 ()"	(4)
"1 # 5 & 1 @ 2 ? always ~ 4"	(5)
"1 # 6 & 0 @ 1 ? always ~ 5"	(6)
"1 # 7 & 0 @ 2 ? always ~ 5"	(7)

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, *i.e.*, 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 two types of enabling and disabling add flexibility but reduce communication cost.

Authentication Manager. To overcome the heavy cost of exponential computation in the traditional DH key agreement protocol, we exploit the Elliptic Curve Cryptography (EEC) 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 [18] package to migrate it to the platform of biosensor node. We implemented the authentication protocol using point multiplication in ECC. First, a base point is chosen and made publicly 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

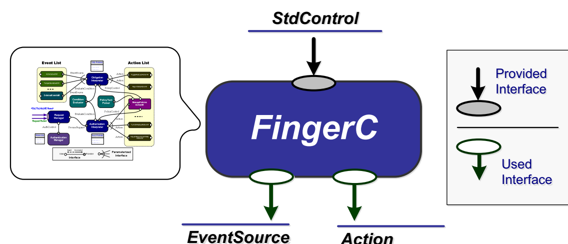

Figure 4: Programming interfaces of Finger.

secret with the keyshare of sensor i, $K_{ji} = s_j(p_i) = s_j s_i G$. In a similar way, sensor i can compute the shared key with j, $K_{ij} = s_i(p_j) = s_i s_j G = K_{ji}$. 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 [19], 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.

Programming Interfaces. 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 Figure 4. To use policies *FingerC* should be included in the application configuration. The *Main* module of the application wires its StdControl to that of the policy system, which initializes the embedded components with Finger. To extend the functionality, application-specific event sources and actions can be developed. Event sources connect to the EventSource interface to trigger obligation policies. Similarly, all actions to be regulated by authorization policies should connect to the Action interface.

5. Experiment results

To facilitate performance measurements, we developed a simple TinyOS application SimApp making use of Finger. This application implements an event source of 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 accesses 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 servers as the run-time data stack. Nevertheless, it is difficult to compute the binary code size of Finger precisely since in TinyOS we only have access to the aggregate code size of an entire application. We need to separate the Finger's code from basic TinyOS and communication components.

The ECC and TinySec libraries require considerable 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.

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 biosensor node has only 2K bytes RAM so cannot host Finger(w), so we had to use a Tmote Sky node instead for experiments with Finger(w). A Tmote node shares the same processor with a biosensor, but it has a larger RAM size (with 10K bytes). 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 biosensor node platform. The executable without authentication occupies 15.62K bytes of ROM and 1.06K bytes of RAM, and the one with authentication occupies 31.28K bytes of ROM and 2.88K bytes of RAM. Table 4 shows the code size breakdown of SimApp, where ROM and RAM are shown separately. From this breakdown, we can calculate that the authentication module using TinyECC takes 15.66K bytes of ROM and 1.82K bytes of RAM.

We examine various processing delays introduced by the policy system. The experiments were conducted with 7 obligation policies and 8 authorization policies. The delays of processing obligations and authorizations are shown in Table 5. 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. It takes as little as 62 μs to process an obligation policy and 81 μs to process an authorization policy. We also measured the latency of raising an external event on a sensor. The whole process includes processing an authorization policy and then an obligation policy. The latency is 140 μs. We

Table 4: Code size breakdown of SimApp

Component	ROM (KB)	RAM (KB)
Finger (w)	20.65	2.35
Finger (w/o)	4.99	0.53
Comm.	8.08	0.49
Basics	2.55	0.04
Total (w/o)	15.62	1.06
Total (w)	31.28	2.88

Table 5: Processing delays

Operation	Delay
Obligation Interp.	62 μs
Authorization Interp.	81 μs
Public Encrypt.	9530 ms
Public Decrypt.	5281 ms
Symmetric Encrypt.	150 μs
Symmetric Decrypt.	90 μs

also measured the latency caused by policy management. It takes 375 μs to load an authorization policy.

We also evaluated delays for various cryptographic operations in the authentication process. It takes on average 9530 ms to encrypt a 52-byte message, whose content are randomly generated, and 5281 ms to decrypt the encrypted message. With the Skipjack library, it takes significantly less time, 150 μs to encrypt the same message and 90 μs 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 more efficiently performed using the symmetric Skipjack cryptography. These delays are acceptable since it is rarely used since the ticket technique is used.

6. Conclusions

We have presented an efficient policy system for body sensor networks. Finger supports efficient sensor-level interpretation and enforcement of both obligation and authorization policies. It realizes policy-based adaptation to changes in context or 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 biosensors. With Finger, application development can also be accelerated since developers only need to focus on developing event sources and actions, and composing policies.

References

[1] L. Schwiebert, S. Gupta, J. Weinmann, A. Salhieh, V. Shankar, V. Annamalai, M. Kochhal, and G. Auner, "Research Challenges in Wireless Networks of Biomedical Sensors," *Proc. ACM MobiCom*, 2001.

[2] G.-Z. Y. (Ed.), "Body Sensor Network," Springer-Verlag, 2006.

[3] E. Lupu, N. Dulay, M. Sloman, J. Sventek, S. Heeps, S. Strowes, S. L. Keoh, A. Schaeffer-Filho, and K. Twidle, "AMUSE: Autonomic Management of Ubiquitous e-Health Systems," *Concurrency and Computation: Practice and Experience,* 2007.

[4] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," *Proc. USENIX/ACM NSDI*, 2004.

[5] J. W. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," *Proc. ACM SenSys*, 2004.

[6] S. S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," *Proc. IEEE ICDCS*, 2005.

[7] D. Liu and P. Ning, "Establishing Pairwise Keys in Distributed Sensor Networks," *Proc. ACM CCS*, 2003.

[8] L. Eschenauer and V. D. Gligor, "A Key-management Scheme for Distributed Sensor Networks," *Proc. ACM CCS*, 2002.

[9] A. P. H. Chan, and D. Song, "Random Key Predistribution Schemes for Fensor Networks," *Proc. IEEE Symposium on Security and Privacy*, 2003.

[10] J. Strassner, *Policy-based Network Management*: Morgan Kaufma, 2004.

[11] S. L. Keoh, N. Dulay, E. Lupu, K. Twidle, A. E. Schaeffer-Filho, M. Sloman, S. Heeps, S. Strowes, and J. Sventek, "Self Managed Cell: A Middleware for Managing Body Sensor Networks," *Proc. MobiQuitous*, 2007.

[12] Ponder2, http://www.ponder2.net/.

[13] L. Kagal, T. Finin, A. Joshi, and S. Greenspan, "Security and Privacy Challenges in Open and Dynamic Environments," *IEEE Computer,* vol. 39, pp. 89-91, June 2006.

[14] J. Lobo, R. Bhatia, and S. Naqvi, "A Policy Description Language," *Proc. AAAI*, 1999.

[15] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," *Proc. IEEE POLICY*, 2001.

[16] L. Kagal, T. Finin, and A. Joshi, "A Policy Language for Pervasive Computing Environment," *Proc. IEEE POLICY*, 2003.

[17] The BSN Specification, http://ubimon.doc.ic.ac.uk/bsn/.

[18] A. Liu and P. Ning, "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks," *Proc. ACM/IEEE IPSN/SPOTS*, 2008.

[19] K. Chris, S. Naveen, and W. David, "TinySec: a Link Layer Security Architecture for Wireless Sensor Networks," *Proc. ACM SenSys*, 2004.