Finger: An Efficient Policy System for Body Sensor Networks

Yanmin Zhu[†], Sye Loong Keoh[†], Morris Sloman[†], Emil Lupu[†], Yu Zhang[‡], Naranker Dulay[†], Nathaniel Pryce[†]
[†] *Imperial College London,* [‡] *IBM China Research Lab*

Abstract

Body sensor networks (BSNs) for healthcare put more emphasis on security and adaptation to changes in context and application requirement. Policy-based management enables flexible adaptive behaviour 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 network adaptation, such as networking programming, suffer from high communication cost and operational interruption. In addition, the policy-driven approach enables fine-grained access control through specifying authorization policies. This paper presents 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.

1. Introduction

Body sensor networks (BSNs) [1-3] have recently been employed for various personal applications, in particular healthcare applications [4]. 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 crucial 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. Only authorized access to body sensors should be permitted, for both accessing data and performing actions.

Little existing work fulfils the above requirements. TinyOS [5], 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 [6-8] 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 [9-11] ensure that sensors trying to communicate with each other share common keys. However, 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 [12]. By separating policies from the system implementation, a policy-driven system can adapt

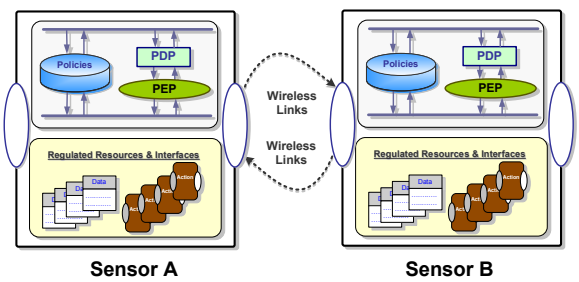

Figure 1: The policy service architecture for body sensor networks. Each sensor maintains its own policies and implements both PDP and PEP.

This research is supported by UK EPSRC through grant EP/C547586/1.

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 [4, 13] 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 [14] 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.

Sensor nodes are becoming increasingly powerful [15] 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 behaviours in response to context changes or application requirement evolvement. For example, a sensor adapts to the current activity of the user. Direct interaction between sensors becomes possible and at the same time, accesses to sensor resources are regulated by authorization policies.

This paper presents the design 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. As illustrated in Figure 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 [14]. The effective simplification makes the policy language suitable for processing on resource-constrained sensors.

2. Motivation

A BSN consists of a *controller*, *body sensors,* 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. Body sensors are attached on the body or implanted within the body for monitoring various aspects of body condition. 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 platform of body sensor node [16], equipped with a TI MSP430F149 microcontroller of a 16-bit RISC processor working at 16MHz. It has only 60KB, read-only program memory for executable code and 2KB data memory as a data stack.

Motivating Example. Consider a simple 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 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 of detecting activity changes. A higher frequency allows the sensor to detect more rapid movement changes but then costs the sensor 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 data can be obtained for more accurate estimation. When the acceleration becomes smaller, it is probable that the user is sitting or standing. Thus, the measurement rate can be reduced for energy conservation. Two obligation policies, (1) and (2) shown in the table, can realize such adaptation. The important parameter of the measurement interval can be re-configured according to application requirements by updating the two policies. 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 (policy (3)) to allow the controller to change its policies.

oblig (1)	**on** accel_event (*acceleration*) **do** adjust_measurement_interval (1s) **if** acceleration >= 30
oblig (2)	**on** accel_event (*acceleration*) **do** adjust_measurement_interval (5s) **if** acceleration <= 20
auth+ (3)	**subject** controller **target** acceleration_sensor **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.

Design Objectives. There are a number of challenges in implementing a policy system on small sensors. It is critical to make efficient use of the limited resources such as small memory. Policy based systems such as Ponder2 are inappropriate for resource constrained sensors. It is also impractical to pre-load all

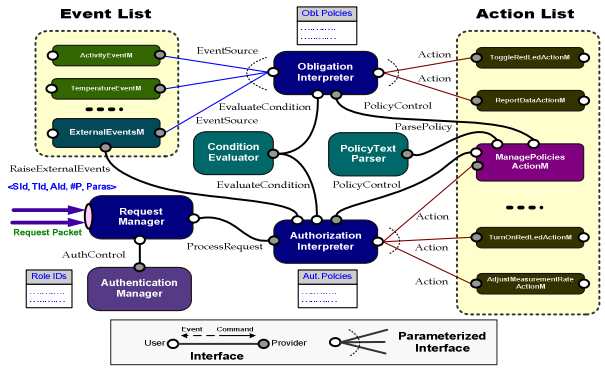

Figure 2. The architectural overview of Finger.

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. In the processing of developing Finger, we have considered several design objectives as important, including dynamic management of policies, minimum memory footprint, responsiveness, well defined APIs and energy efficiency.

3. Design of Finger

The architectural overview 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 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 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). The design of 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 then triggers the OI.

3.1. Authentication Protocol

To make access control effective, 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 which is 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. 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 having been admitted into the network do not behave maliciously by spoofing the membership list.

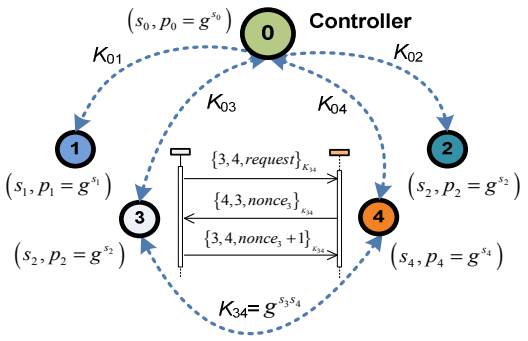

Figure 3: Diffie-Hellman based key establishment, and three-way handshake authentication procedure

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,

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

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 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_{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 and checks the ticket. 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 starting a new challenge-response procedure, after it has been used for an extended period.

The exponential notations in the DH-based protocol are for conceptual description only. Exponential computation with big integers is too computationally expensive for body sensors. Instead, we make an efficient implementation based on elliptic curve cryptography (ECC).

TABLE 1: POLICY TEXTS

"0 # 1 & 1 ? 1^ >=30 ~ 1 (1)"	(1)
"0 # 2 & 1 ? 1^ <=20 ~ 1 (5)"	(2)
"1 # 6 & 0 @ 1 ? always ~ 5"	(3)

4. Implementation

We have implemented Finger using nesC [17] with TinyOS v1.15 on the platform of body sensor node [16]. This section describes several important aspects of Finger implementation.

We have to scale down the complexity of policies since small sensors cannot afford to process complex policies used in traditional distributed systems. We have designed a simple and efficient policy language with a syntax suitable for efficient processing by body sensors yet it is expressive and able to fulfil most management needs of sensor networks. 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 1. 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".

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. This component 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

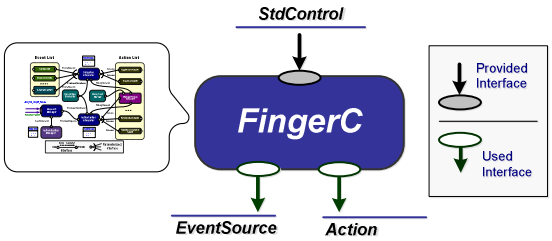

Figure 4: The exposed TinyOS interfaces to be used by application developers.

of enabling and disabling add flexibility but reduce communication cost.

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 body sensor 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 secret with the keyshare of sensor i, $K_{ji} = s_j(p_i) = s_js_iG$. In a similar way, sensor i can compute the shared key with j, $K_{ij} = s_i(p_j) = s_is_jG = K_{ji}$. 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 IV.

Finger provides easy-to-use 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. 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.

5. Performance Evaluation

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. 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. We propose a technique of taking difference of SimApp with other simple TinyOS programs. 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. To compute the sizes of TinyOS basics and the communication subsystem, we use two other simple applications. Blink is a simple application that regularly toggles the red light. It only contains the basic TinyOS components. CntToRadio is also a simple application that maintains a counter and periodically broadcasts the counter value, so it includes the components for both basic TinyOS and the communication subsystem. By taking the code size difference between SimApp with the two basic applications, we can derive the code size of Finger. 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.

We exploited the timing facility provided by TinyOS to measure processing delays precisely, 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. This interface allows us to read the current local time on the sensor. Our measurement component records timestamps and sends them back to the PC end for delay calculation. This guarantees that no other delays are included in calculated processing delays. The results each are averaged over 20 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 ROM 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 body sensor, 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 body sensor 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. We 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 the seven deployed obligation policies and eight deployed authorization policies. 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. From the table we can see that 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 caused by policy management. It takes 375 μs to load an authorization policy. Thus, it takes in total 437 μs 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 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.

6. Conclusions

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 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 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 for the body sensor platform, it is extensible and can be deployed to many other platforms including Mica2, Telos and TMote Sky.

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] G. Zhou, J. Lu, C.-Y. Wan, M. Yarvis, and J. Stankovic, "BodyQoS: Adaptive and Radio-Agnostic QoS for Body Sensor Networks," *Proc. IEEE INFOCOM*, 2008.

[4] 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.

[5] TinyOS Community Forum, http://www.tinyos.net/.

[6] 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.

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

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

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

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

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

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

[13] 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.

[14] Ponder2, http://wwww.ponder2.net/.

[15] J. Polastre, R. Szewczyk, and D. E. Culler, "Telos: Enabling Ultra-low Power Wireless Research," *Proc. ACM/IEEE IPSN*, 2005.

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

[17] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems," *Proc. ACM PLDI*, 2003.

[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.