Proceedings of the 5th International Workshop on Wearable and Implantable Body Sensor Networks, in conjunction with
The 5th International Summer School and Symposium on Medical Devices and Biosensors
The Chinese University of Hong Kong, HKSAR, China. Jun 1-3, 2008

A Policy System to Support Adaptability and Security on Body Sensors

Yanmin Zhu, Sye Loong Keoh, Morris Sloman, Emil Lupu, Naranker Dulay, Nathaniel Pryce
Department of Computing, Imperial College London
{yzhu, slk, mss, ecl1, nd, np2}@doc.ic.ac.uk

***Abstract*—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 node operating systems, such as TinyOS, which do not support dynamic modification of code. This paper presents the design, implementation and evaluation of an efficient policy system called Finger supports both event condition action rules which provide a simple means to 'program' adaptive behaviour and authorisation policies to protect node services and resources from external access. The system performance in terms of processing latency and resource usage is evaluated.**

I. Introduction

Wireless sensors have been recently used for post-operative care in hospitals and for treatment of chronically ill patients or aged users at home [1]. The nodes continuously monitor physiological parameters but may need to only report events indicating thresholds have been crossed in order to conserve battery power. However, some applications may require frequent readings when a threshold is crossed. Thus, the nodes need to adapt behaviour in response to changes in the environment or to new requirements. Nodes often need to cooperate by accessing data or invoking actions on other nodes. However, not all nodes can be trusted, particularly for medical applications, and so access control is needed to permit access only from authorised nodes.

TinyOS [2] is the *de facto* standard operating system for sensor nodes but does not support dynamic modification of code so it is difficult for a sensor node to change its behaviour. A typical solution is to shut down the network and reprogram all nodes by transmitting a whole node code image over wireless links [3-5]. This entails considerable use of power for wireless communication and interrupts the current operation of the network.

By separating policies from the implementation, a policy-driven system can dynamically adapt to changes in either environmental context or application requirements by dynamically changing policies [6]. This paper presents the design, implementation and evaluation of an efficient policy system called **Finger.** This system supports policy 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 node. In essence, Finger supports a considerably simplified version of the Ponder2 language [7, 8] for policy specification, which is suitable for processing on small sensor nodes. Compact design and implementation on TinyOS makes the footprint of the policy system minimal.

The paper is structured as follows – Section II gives a motivating example and Section III describes the architectural design. Implementation details follow in Section IV and performance measurements are presented in Section V. The paper in concluded in Section VI.

II. Motivating Example

Consider a simple healthcare scenario where a body sensor network consists of a controller, a temperature and accelerometer sensor nodes. The accelerometer is used to determine user activity, *e.g.*, walking or sitting. The controller typically performs tasks such as data aggregation, policy deployment and remote communication. The accelerometer node starts a timer and regularly (*e.g.*, every 5 seconds) reads accelerometer data. The timer frequency is an important parameter that determines the sensor's ability to detecting activity changes. A higher frequency allows the sensor to detect more rapid movement changes but then the node 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, the 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 medic may decide that it is useful to study the relation between body temperature and user activity, so the temperature node should record the body temperature when an activity change occurs. However, this function has not been pre-programmed on the node, but, could be achieved by deploying a new obligation policy on the acceleration node to notify the temperature node of new activities:

```
oblig   on new_activity_event(activity)
(3)     do raise_event (new_activity_event, activity)
           on temperature_node
```

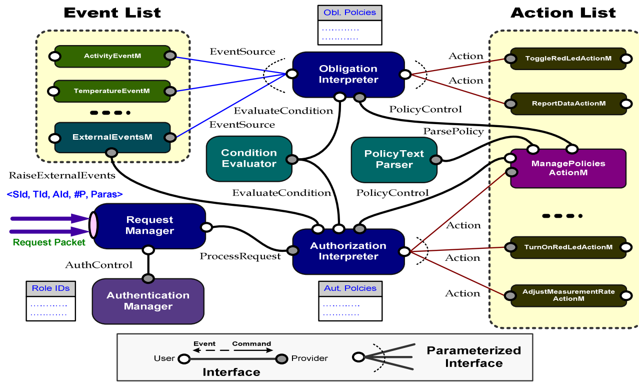

Figure 1. The architectural overview of Finger. Components of Finger are connected by interfaces that define commands and events. The gray end of an interface provides commands and fire events, while the white end uses commands and handle events.

The temperature node 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 acceleration node raises an event on the temperature node and this should be subject to an authorization policy on the temperature node to permit the incoming event from the acceleration node.

auth+ **subject** acceleration_node
(5) **target** temperature_node
action raise_event

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

auth+ **subject** controller
(6) **target** acceleration_node
action manage_policy
auth+ **subject** controller
(7) **target** temperature_node
action manage_policy

This example demonstrates that sensor nodes 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 [7]. This allows policies to be defined for groups of nodes rather than just individual ones.

III. FINGER OVERVIEW

The architectural overview of Finger is depicted in Figure 1. The core of Finger comprises two components, *i.e.,* the Obligation Interpreter (OI) and the Authorization Interpreter (AI) for 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 by an independent component so that normal requests, which are subject to authentication and authorisation checks, can be used to manage the policies.

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. When an event occurs, the event component triggers the OI to search for all policies matching this event type in the policy repository. 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 ID*, *Action ID*, *# of Paras*, *Paras*>. The Request Manager (RM) receives incoming requests and authenticates the requesting subject by invoking the Authentication Manager (AM). The AM maintains a table of all valid roles in its vicinity. New roles emerging in the network can be periodically broadcast by the controller. Cryptographic methods will be exploited to authenticate the requesting node. If the subject is unknown to the target node, this request should be dropped. Note that we are still developing the AM module.

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 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. If such action is permitted, the AI invokes, through the RaiseExternalEvents interface, the *ExternalEventsM* component, which immediately triggers the OI.

In order to reduce code size, we factor out the implementation of condition evaluator and make it as an independent component so only one copy of code is needed for both the OI and the AI.

IV. IMPLEMENTATION DETAILS

In this section we discuss implementation details. We have implemented Finger using nesC [9] with TinyOS v1.15 on the hardware platform of body sensor nodes (BSNs) [10]. The code of the current implementation of Finger is available online at [11].

A. Policy Specification

We have to scale down the complexity of policies since small sensor nodes 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 small sensor nodes yet it is expressive and able to fulfil most management needs of sensor networks. The syntax of policies is specified in Table 1.

TABLE 1: 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*

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. How roles are assigned is beyond the scope of this paper. Following this simple syntax, the policies used in the motivating example in Section are as follows.

Policy	
"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)

Default authorization for invoking an action is negative (*i.e.*, not permit) for those requests failing to match an authorisation policy in the repository.

B. Memory Organization

It is of great importance to manage the limited memory of sensor node. 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 and returned back to the vacant list.

It is not easy to predict the maximum number of policies that a sensor node will require. 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.

C. Dynamic Policy Management

Dynamic management of policies is crucial to the adaptation ability of sensor nodes. 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.

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 should be able to inform the OI of interpreting 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 *event id* and *event 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. This design greatly supports evolvement of Finger, which is particularly valuable for application developers. To add more event sources or actions, the developers can simply develop their own TinyOS modules, implementing the interfaces of EventSource and Action. The implementation of the OI and the AI stays intact, while the developers only need to concentrate on application logic. This design also allows the policy system to expose neat programming interfaces to application developers.

E. Application Programming Interfaces

Finger provides simple application programming interfaces (APIs) for application developers. The components of Finger are integrated as a single TinyOS configuration component, called *FingerC*. This hides the implementation details of Finger from developers. Finger exposes a provided interface StdControl and two used interfaces EventSource and Action. The application should wire its StdControl to that of the policy system, which initializes the embedded components. To extend the functionality, the developer can develop application-specific event sources and actions. All developed event sources should connect to the EventSource interface if these events are governed by obligation policies. Similarly, all actions to be regulated by authorization policies should connect to the Action interface.

V. PERFORMANCE MEASUREMENT

In this section, we present the performance evaluation of Finger. We have conducted a set of experiments and analyzed the performance in terms of two performance metrics, *i.e.*, memory overhead and processing delay.

TABLE 2: CODE SIZE BREAKDOWN OF SIMPLOY.

Component	ROM (Kbytes)	RAM (Kbytes)
Finger	4.99	0.53
Comm.	8.08	0.49
Basics	2.55	0.04
Total	15.62	1.06

Memory overhead. We investigate the memory solely used by Finger. More specifically, we look at the ROM and RAM sizes. The ROM is used for program code and the RAM is for run-time data storage. It is not easy to precisely compute the binary code size of Finger since in TinyOS we can only access the aggregate code size of an entire application. We have built a simple application using Finger, called SimPoly. We compute the memory consumption of the policy system by deducting the memory sizes taken by other system components. More specifically, we compare SimPoly with the basic applications Blink and CntToRfm. Blink is a simple application using only basic TinyOS components. CntToRfm is another simple application but uses the communication subsystem. After compilation into the executable binary, this application SimApp occupies 15.62K bytes of ROM and 1.06K bytes of RAM. Table 2 shows the code size breakdown of SimPoly, where ROM and RAM are treated separately. We can see that the policy system uses minimal memory resource, *i.e.*, 0.49K bytes RAM and 8.08K bytes ROM.

Processing delay. We examine the processing latency introduced by the policy system. To precisely measure processing delays, we exploited the timing facility provided by the TinyOS and all measurements were made on the sensor node 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 enables us to read the current local time on the sensor node. *MeasureTimeM* records the 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 following results each are averaged over 20 independent measurements. The delays of processing obligations and authorizations are measured on a BSN node running the SimPoly application with seven obligation policies and eight authorization policies. It takes as little as $62\mu s$ to process an obligation and $81\mu s$ to process an authorization policy. We measured the latency of raising an external event on a sensor node. The whole process includes processing an authorization policy and then an obligation policy. The latency is $140\mu s$. We also measured the latency caused by policy managements. It takes $375\mu s$ to load an authorization policy. Thus, it takes in total $437\mu s$ to process a loading-policy request and load the policy.

VI. CONCLUSIONS

In this paper, we have presented the Finger novel policy system for supporting both obligation and authorization policies for distributed sensor networks. It enables policy-based dynamic adaptation to changes in context or application requirements without interrupting the current network operation. The policy system occupies minimal resources, only 4.99K bytes of ROM and 0.53K bytes of RAM. Measurements on the prototype application shows that enforcing a policy incurs as little as $62\mu s$ to process an obligation and $81\mu s$ to process an authorization policy. We stress that although Finger has been implemented on BSNs, it is extensible and can be deployed to many other sensor node platforms including Mica2, Telos and TMote. Currently, we have an initial implementation of the authentication module which still needs to be integrated with authorisation.

ACKNOWLEDGEMENTS

We gratefully acknowledge support from the UK Engineering and Physical Sciences Research Council (EPSRC) through grant EP/C547586/1 (Biosensornet Project).

REFERENCES

[1] G.-Z. Y. (Ed.), "Body Sensor Network," Springer-Verlag, 2006.
[2] TinyOS Community Forum, http://www.tinyos.net/.
[3] 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.
[4] J. W. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," *Proc. ACM SenSys*, 2004.
[5] S. S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," *Proc. IEEE ICDCS*, 2005.
[6] J. Strassner, *Policy-based Network Management*: Morgan Kaufma, 2004.
[7] Ponder2, http://wwww.ponder2.net/.
[8] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," *Proc. IEEE POLICY*, 2001.
[9] 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.
[10] The BSN Specification, http://ubimon.doc.ic.ac.uk/bsn/.
[11] The Finger Policy System for Body Sensor Networks, http://www.doc.ic.ac.uk/~yzhu/projects/finger/.