In Proceedings of the 4th IEEE International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2007), AAchen, Germany, March 2007

Policy-based Management for Body-Sensor Networks

Sye Loong Keoh[1], Kevin Twidle[1], Nathaniel Pryce[1], Alberto E. Schaeffer-Filho[1], Emil Lupu[1], Naranker Dulay[1], Morris Sloman[1], Steven Heeps[2], Stephen Strowes[2], Joe Sventek[2] and Eleftheria Katsiri[1]

[1]Deparment of Computing, Imperial College London, UK
[2]Department of Computing Science, University of Glasgow, UK

Abstract—**Body sensor networks e.g., for health monitoring, consist of several low-power on-body wireless sensors, higher-level devices such as PDAs and possibly actuators such as drug delivery pumps. It is important that such networks can adapt autonomously to changing conditions such as failures, changes in context e.g., user activity, or changes in the clinical condition of patients. Potential reconfiguration actions include changing the monitoring thresholds on sensors, the analysis algorithms or the configuration of the network itself. This paper presents a policy-based approach for autonomous management of body-sensor networks using the concept of a Self-Managed Cell (SMC). *Ponder2* is an implementation of this approach that permits the specification and enforcement of policies that facilitate management and adaptation of the response to changing conditions. A Tiny Policy Interpreter has also been developed in order to provide programmable decision-making capability for BSN nodes.**

Keywords—**Autonomic management, adaptive sensing, policy-based adaptation, reconfigurable networks.**

I. Introduction

On-body and implantable sensors have the potential to be used in hospitals or homes for monitoring physiological parameters of post-operative and chronically ill patients (e.g., those suffering from diabetes mellitus). These sensors use wireless communications to form body sensor networks (BSNs) [1] and can interact with wearable processing units such as PDAs, mobile phones and the fixed network infrastructure. Health monitoring using BSNs enables early release of patients from hospital and facilitates continuous monitoring of clinical condition in the home or at work. Additionally, healthcare personnel can be automatically alerted to obtain assistance if the patient's condition deteriorates.

Whilst much of the current work focuses on the development of new sensors and processing the data acquired from them [1], we focus on providing adaptation and self-management at both sensor level and for body sensor networks. For example, there is a need to adapt the frequency of measurements on a sensor depending on the activity and clinical condition of the patient. This enables optimising power consumption whilst ensuring that important episodes are not missed. Similarly, the use of variable thresholds for transmitting sensor readings reduces the need for communication and thus power consumption. Typically, sensor configuration may also change depending on the user's context, e.g., location, current activity and medical history. Physiological parameters such as heart rate thresholds then need to be configured and customised accordingly. Policy-based techniques have been used for over a decade in network and systems management in order to define how the system should adapt in response to events such as failures, changes of context or changes in requirements. By specifying the policies (i.e., what actions should be performed in response to an event) declaratively and separately from the implementation of the actions, it is possible to dynamically change the adaptation directives without changing the implementation or interrupting the functioning of the device. Thus, policy-based mechanisms provide feedback control over the system and a constrained form of programmability.

In this paper, we present a policy-based architecture that supports autonomic management for body sensor networks, based on the concept of a Self-Managed Cell (SMC) [2]. A SMC consists of an autonomous set of hardware and software components that represent an administrative domain such as a body area network of physiological sensors and controllers. We introduce *Ponder2* [3], a toolkit that supports the specification and enforcement of policies in the form of event-condition-action rules, the grouping of a SMC's components in domains for management purposes and the dynamic loading of new functionality and new communication protocols. Policies can be defined with respect to the SMC's components, for interactions with the other SMCs and for the management (i.e., loading, removal, activation) of the policies themselves. We also describe the implementation of a policy interpreter for BSN nodes [4].

The paper is organised as follows: Section II describes the SMC architectural pattern. Section III presents the Ponder2 policy interpreter as the core component for adaptation and feedback control. Section IV describes the implementation of a policy interpreter for the BSN nodes. Section V discusses our prototype implementation of Ponder2 and Tiny Policy Interpreter. Sections VI and VII present the related work and conclude the paper with directions for future work.

II. Self-Managed Cells (SMC)

As an example, an SMC represents a body-sensor network consisting of several sensors, i.e., *glucose, blood pressure, heart-rate, ECG monitor* and actuators such as an insulin pump to administer appropriate dosage of insulin in a diabetes Type II patient monitoring system. Context sensors such as accelerometers may be used to sense the patient's activities in order to facilitate the adjustment of drug delivery and monitoring threshold (e.g., for heart-rate).

Figure 1 illustrates the architectural pattern of a SMC that manages a set of heterogeneous components (i.e., managed resources) such as those in a body-sensor network, a room or even a large-scale distributed application. Resource adapters are instantiated to provide a unified view for interaction with the resources as they may use different interfaces or communication protocols. For example, interactions with BSN nodes occur via IEEE 802.15.4 wireless links, while interactions with PDAs, mobile phones or Gumstix[1] typically occur over Wifi or Bluetooth.

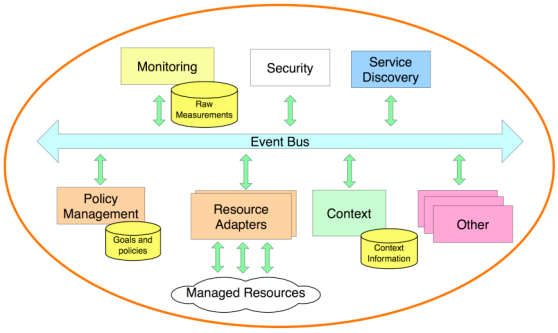

Fig. 1 The SMC architecture pattern

A SMC can load other components and services for detecting context changes, monitoring component behaviours or for security (authentication and access control). However, the *event bus*, the *policy service,* and the *discovery service* work in conjunction with each other and form the core functionality of a SMC that must always be present.

As most pervasive systems are event-driven, the services of an SMC interact using a common publish/subscribe event bus, although we do not constrain all communication to be event-based. The event bus [5] forwards event notifications from services onto any interested parties within the SMC who have subscribed to receive the event. This has the advantage of decoupling the services since an event publisher does not need to know about the recipients, thus permitting the addition of new services to the SMC without disrupting the behaviour of existing services. Secondly, multiple services in the SMC can respond concurrently and independently to the same notification with different actions. Finally, the event bus can be used for both management and application data such as alarms indicating that threshold have been exceeded. In order to lower communication overhead, sensors typically only transmit events when an unusual situation arises rather than transmitting all sensor readings.

The policy service implements a local *feedback control loop* to achieve adaptation and self-management. It caters for two types of policies: *obligation policies* (event-condition-action rules), which define the actions that must be performed in response to events, and *authorisation policies* which specify what actions are permitted on which resources and services. The discovery service is used to discover new components which are capable of becoming members of the SMC, e.g., sensors and other SMCs in the vicinity. It establishes a profile describing the services they offer and generates an event describing the addition of the new device for other SMC components to use as appropriate. The discovery service also manages the SMC's membership as it is necessary to cater for transient failures, which are common in wireless communications, and to detect permanent departure (e.g., device out of range, switched off, or failure).

Complex environments can be realised through the federation and composition of several SMCs. This permits exchanging policies between SMCs and thus programming in a restricted way the behaviour of collaborating or composed SMCs [6].

III. Ponder2 Policy Service

Ponder2 [3] is the policy service for the SMC and has been inspired by the lessons learnt in the development of Ponder [7], a policy language and toolkit developed at Imperial College over a number of years. In contrast to Ponder, which was designed for general network and systems management, Ponder2 has been designed as an entirely extensible framework that can be used at different levels of scale from small embedded devices to complex environments.

Ponder2 combines a general-purpose object management system with a *domain service*, *obligation policy interpreter* and a *command interpreter*. The Domain Service provides a hierarchical structure for managing objects. The Obligation Policy Interpreter handles Event, Condition, Action rules (ECA). The Command Interpreter accepts a set of commands in XML form via several communication interfaces. These commands can be used to interrogate the Domain Service and perform invocations on the managed objects.

Managed objects (also called adapter objects) represent sensors and other SMC devices, services within those devices and remote SMCs. Domains and policies are managed

[1] www.gumstix.com

objects in their own right on which actions can be performed, e.g., adding/removing an object from a domain, enabling or disabling a policy. Obligation policies are also used to decide which adapters should be created when new components appear and in which domain they should be placed. Other policies specified for that domain will then automatically apply to the new component.

As shown in Figure 2, the overall architecture of the policy service comprises the domain structure, the triggering mechanism matching events to obligation policies and the execution invocation engine which is used to make the calls to the objects inside the domain structure. Conceptually the policy service has an event interface through which event notifications are received from the external event bus, an invocation interface through which external invocations are received and an action interface for invocations on external objects.

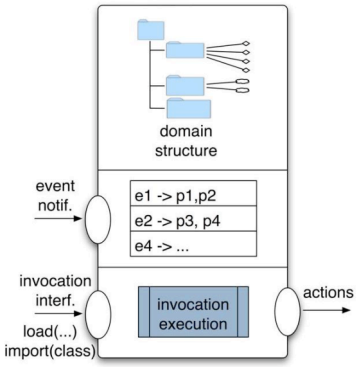

Fig. 2 The policy service architecture

Ponder2 uses XML to encode policies, event specifications and invocation of actions. It interprets XML as a sequence of statements that identify the managed object to be used and parameters or sub-elements within the XML that are to be sent to the managed object. For example, the following snippet identifies the root domain ("/") and sends it an *add* command. The *add* command has its own structure and information saying what is to be added to the domain structure. In this example, the managed object to be added to the root domain will be called *newobject*.

```
<use name="/">
  <add name="newobject">
    ...
  </add>
</use>
```

XML is verbose and not easy to use, so we are developing a higher-level declarative language from which the XML can be generated.

A. Instantiation of Managed Objects

Ponder2 has the ability to load all the code needed on-demand. This enables us to use it across a wide variety of applications and devices with different capabilities by only loading those components that are necessary in each case. By default, a domain hierarchy rooted at / will be created. A built-in *domain factory* is used to create new domain objects within the domain hierarchy. All other managed objects are instantiated through factories that can be loaded on demand. Ponder2 has defined several other *factory objects* such as *policy (authorisation and obligation), basic managed object,* and *event.* This provides the flexibility to dynamically create policies and managed objects for communicating with various sensors and devices with their respective communication protocol such as UDP datagram, Bluetooth [8], IEEE 802.15.4, or Zigbee [9]. For example, the following XML snippet uses a BSN *glucose factory* object to create a managed object for a BSN based glucose sensor and places it into the `/bsn` domain under the name *GlucoseSensor*. This managed object acts as an adapter to the actual sensor and implements the high-level interface in terms of interactions with the sensor via 802.15.4 radio.

```
<use name="/bsn">
   <add name="GlucoseSensor">
      <use name="/factory/BSNglucose">
         <create addr="0:0:0:3" />
      </use>
   </add>
</use>
```

Once instantiated, managed objects receive commands in the form of XML structures. Typically, the main XML element is the `<use name="x">` construct to identify a managed object, with one or more operations or sub-commands represented by child elements. A command to a managed object takes the form:

```
<use name="managedobject" arg1="foo" arg2="bar">
  <op1 op1arg1="doh"/>
  <op2 op2arg1="argh"/>
  ...
</use>
```

where the operations/commands `op1` and `op2` could have child elements of their own.

B. Integration with Event Buses

An *event factory* implements the interface with an external event bus and encapsulates the protocols necessary to communicate with it. Multiple event factories can be used to integrate with different external event buses if required. The *event factory* can be used to create new Event Types that issue a subscription to the external content-based event bus and define the event type name and its arguments. When an

event corresponding to the subscription expression occurs on the event bus, the event factory is notified of its occurrence and raises the corresponding event type in the policy interpreter in order to trigger the policies.

```
<use name="/Event">
  <add name="highGlucoseEvent">
    <use name="/factory/GlasgowEvent">
      <create>
        <and>
         <sub name="level" op="GEQ" value="90" />
         <sub name="type" op="EQ" value="glucose"/>
        </and>
        <arg name="type"/>
        <arg name="level"/>
        <arg name="daytime"/>
      </create>
    </use>
  </add>
</use>
```

In the above example, an Event Type called */event/highGlucoseEvent* that has three arguments, sensor *type,* glucose *level* and *context* is created. The subscription expression to which this event type corresponds is matched by the external event bus against published events on the bus. In this example all events with type *glucose* and level ≥ 90 are matched.

This example assumes that the sensor transmits glucose readings periodically as events to the event bus. The frequency with which this is done is determined by the sensor and may be configurable. However, in order to minimise power consumption linked to the communication of readings, it is desirable that the sensor itself be programmable in terms of basic policies that can determine when actions such as raising external notifications should be done. This has led us to develop a basic policy interpreter for BSN nodes which is detailed in Section IV.

C. Policy Specification

We are primarily concerned with two types of policies: *authorisation policies* define what actions are permitted under given circumstances and *obligation policies* define what actions to carry out when specific events occur if the specified condition is true. An obligation policy specifies the Event Type that will trigger the policy together with arguments expected of that event, optional conditions that must be satisfied and a set of actions to be performed.

The example below shows the XML encoding of an obligation policy that will respond to events of type */event/highGlucoseEvent*. The policy becomes active as soon as it is created. It makes use of three named arguments provided by the event and their value is substituted in the actions and constraint by enclosing the name inside the two characters "!" and ";", e.g., *!level;* and *!daytime;*. The condition can contain simple Boolean statements comparing string and integer values. In the example above, Ponder2 checks whether the glucose level is greater than *125* and the current *time* is later than 20:00. Conditions can contain any combination of *and*, *or*, *not*, *eq*, *ne*, *gt*, *ge*, *lt* or *le*. *And* and *or* take any number of XML sub-elements, *not* takes one, while the others all take two. Note that the arguments for the comparisons have been separated by XML comments to ensure that they are separate XML elements. The action part consists of Ponder2 XML commands, which are only executed after the event has occurred and if the conditions are true. In this example, if the conditions are satisfied, the policy invokes an action on the insulin pump to increase the night dosage by 10%. Other actions defined in a policy could be to send an alarm SMS message to a medical service or tell the patient to perform some actions.

```
<use name="/policy">
  <add name="AdjustDosagePolicy">
    <use name="/factory/policy">
      <create type="obligation"
       event= "/event/highGlucoseEvent"
       active="true">
        <arg name="type"/>
        <arg name="level"/>
        <arg name="daytime"/>
        <condition>
          <and>
            <gt>!level;<!-- -->125</gt>
            <gt>!daytime;<!-- -->20:00</gt>
          </and>
        </condition>
        <action>
          <use name="/actuators/ip">
            <modify ctx="night" value="+10"/>
          </use>
        </action>
      </create>
    </use>
  </add>
</use>
```

IV. TINY POLICY INTERPRETER

Although Ponder2 confers a level of programmability and adaptation to the SMC, it is equally desirable to be able to introduce similar abstractions at the sensor level in order to endow sensors with programmable local decision capability. Consequently, we have implemented a simplified policy interpreter for TinyOS [10] that can be deployed to BSN nodes [4]. It is implemented as a NesC [11] component library that can dynamically add and remove policies specified either as textual scripts or as data structures. Figure 3 shows the component architecture of the Tiny Policy Interpreter. The Policy Script Controller is responsible for loading, adding and removing policies, while the policy interpreter is invoked to execute the actions according to the deployed obligation policies when events occur.

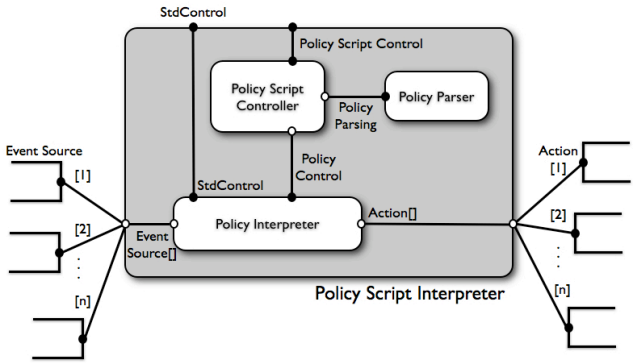

Fig. 3 Configuration of the Tiny Policy Interpreter

Tiny Policies are also specified as event-condition-action rules. Events represent samples from sensors and carry a 32-bit parameter that is the sampled value. Conditions are represented as inclusive ranges: if the value lies within the range, the condition is met. Event sources and actions are represented by NesC interfaces and they are pre-defined as arrays bound to the Tiny Policy Interpreter.

```
policy = event condition "->" action

Event = uint8 "?"

condition = equals_condition
          | in_range_condition
          | less_than_condition
          | greater_than_condition
          | always_condition

equals_condition = uint32

in_range_condition = "["min:uint32 ".." max:uint32"]"

less_than_condition = "<=" max:uint32

greater_than_condition = ">=" min:uint32

always_condition = "always"

action = do_action | set_property_action

do action = uint8 "!" action_arg?

action_arg = "(" uint32 ")"

uint8 = <unsigned 8-bit integer>
```

Fig 4 Policy syntax of Tiny Policy Interpreter

The Tiny Policy Interpreter implements a simple script syntax shown in Figure 4. This syntax is much simpler and compact than the Ponder2 XML policy syntax, which can be translated into the Tiny Policy syntax before deployment to sensors. For example, the policy `1? <= 4 -> 2!` specifies that if event 1 fires with a parameter less than or equal to 4, then perform action 2, while the policy `2? [5..9] -> 1!` means that if event 2 fires with a parameter between 5 and 9 inclusive, then perform action 1. The `always` clause is used to execute an action without needing to evaluate any policy condition, e.g., the policy `1? always -> 3!` means that when event 1 occurs, perform action 3.

A. Actions Library

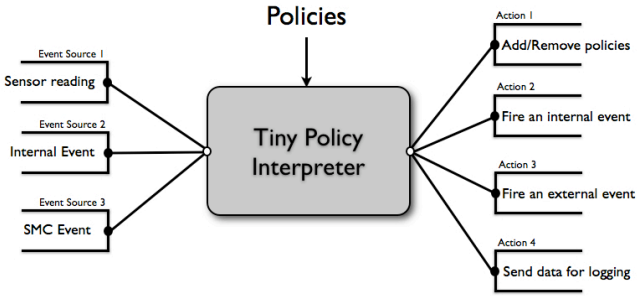

Fig. 5 Event source and action libraries

A set of general-purpose actions for the Tiny Policy Interpreter has also been defined as shown in Figure 5. This includes actions to add and remove policies from the Tiny Policy Interpreter itself. This facilitates adaptation to context changes as policies can be used to decide which set of policies applies according to the user's activity. For example, we can load a policy with a higher heart-rate threshold when the accelerometer has detected higher levels of physical activity. Additional actions include the ability to fire an internal event which triggers another policy within the interpreter in order to allow for the execution of sequential actions, or an external event to the SMC's event bus in order to trigger management actions at the SMC level. Finally, an additional pre-defined action can be used to log measurement values, e.g., to a medical database in the SMC. The event sources for triggering policies within the Tiny Policy Interpreter are either sensor readings, pre-defined internal events or events received from the SMC's event bus, e.g., events generated by other sensors or managed objects in the SMC.

Ponder2 is typically running on a PDA or Gumstix device can load and remove policies from the Tiny Policy Interpreter, thus enabling it to dynamically configure the sensing parameters and behaviour. This provides the flexibility to reconfigure the sensor behaviour without needing to re-program them.

V. Implementation And Evaluation

Ponder2 has been implemented in Java 1.4 and a version has been ported to J2ME. It is deployable to any computing platforms that have a Java Virtual Machine and we have used it in PDAs, Gumstix and mobile phones. We have also implemented a content-based publish/subscribe event noti-

fication system [5] for the SMC. Other event systems such as XMLBlaster [12] and JMS [13] can also be used in conjunction with Ponder2.

In terms of evaluation, we have deployed Ponder2 onto a Gumstix which has a 400 MHz Intel XScale PXA255 processor with 16 MB flash memory and 64 MB SDRAM, running Linux. We observed that the evaluation of policy constraints incurs the most overheads as this involves parsing of the constraints, string comparisons and arithmetic operations. Constraints are stored in the policies in XML format and substantial gains in performance would be possible by pre-compiling them. The time taken to execute a policy without a condition and with an empty action is only 13.57 ms, while it takes 30.05 ms to execute a policy with a simple condition and an action to publish a new event. We also observed that it takes 23.88 ms to execute a policy (with no condition) to invoke an action to issue a command to BSN node via IEEE 802.15.4. In a medical scenario such as diabetes monitoring, many operations take place over time frames of minutes or even hours, so these performance figure are more than adequate in many cases. Some applications such as heart ECG monitoring and analysis, require much faster processing. In these cases, processing can be done directly in the managed objects and sometimes on the BSN sensors themselves.

Each policy is instantiated as a Java object which consumes 3.214 kB. This however includes the policy type (obligation or authorisation), the list of events which may trigger the policy, the actions to be performed and the constraints that need to be evaluated. At the moment we are using XML for internal policy representation, which carries a significant memory overhead. More compact representations could be used for devices with limited memory.

As for the Tiny Policy Interpreter, its current implementation can be installed on BSN nodes [4] and the size of its codebase is 11.61 kB.

VI. Related Work

Work on policy-driven systems has been on-going for over a decade in various application areas. Traditional approaches rooted in network and systems management include PCIM [14], PDL [15], NGOSS Policy [16], Ponder [7] and PMAC [17]. They all make use of event-condition-action rules for adaptation but are aimed at the management of distributed systems and network elements and do not scale down to small devices and sensors.

There are a number of pervasive systems that define frameworks for realising pervasive spaces. Gaia [18] and Aura [19] introduce the notion of *active space* and *smart space* respectively in order to provide a "meta-operating system" to build pervasive applications. These projects focus on spaces of relatively fixed size, e.g., a room or a house and on specific concerns such as context-related applications, user presence and intent and foraging for computational resources. We consider a SMC as an architectural pattern that applies at different levels of scale and we focus on generic adaptation mechanisms through policies.

The Pervasive Information Community Organisation (PICO) [20] is a middleware platform to enable effective communication and collaboration among heterogeneous hardware and software entities in pervasive computing. A community is a grouping of hardware entities and software agents that work together to achieve goals. The notion of community is similar to our SMC, but our focus is to facilitate self-configuration and self-management using policies.

CodeBlue [21, 22] is an ad-hoc sensor network infrastructure for emergency medical care. It integrates low-power, wireless vital sign sensors, PDAs and PC-class systems to provide a combined hardware and software platform for medical sensor networks. CodeBlue also provides protocols for device discovery, publish/subscribe, multi-hop routing and a simple data query interface for medical monitoring. CodeBlue investigates the data rates, node mobility, patterns of packet loss and route maintenance of the wireless sensor network, while the SMC framework focuses on the management of body-sensor networks using policies.

The co-operative artefacts concept [23] is based on embedded domain knowledge, perceptual intelligence and rule-based inference in movable artefacts. Measurements from the sensors are translated into observational knowledge, which is then being evaluated against predefined rules that are defined using Horn logic and domain knowledge. This allows the inference engine to derive the artefact's behaviour in response to the changes in the environment. Smart-Its Context Language (SICL) [24] is a high-level description language for developing context-aware applications on embedded systems. It is integrated with a tuple-space-based communication abstraction that enables inter-object collaboration. The language provides a means of specifying sensor interfaces, inference rules for fusing sensors readings, adaptation rules and basic application behaviour. However, this approach does not support dynamic re-configuration of sensors as re-configuration implies reprogramming of sensors.

VII. Conclusions And Future Work

We have proposed the SMC abstraction as a basic architectural pattern that provides local feedback control and autonomy. Policies in the form of event-condition-action rules, provide a simple and effective encoding of the adapta-

tion strategy required in response to changes of context or in application requirements. The ability to dynamically load, enable and disable the policies together with the ability to use policies in order to manage other policies caters for a wide variety of application needs. Ponder2 has been designed as an extensible framework where all required components can be loaded on demand. This enables us to scale the pattern down to relatively small devices and customise the interpreter for specific application requirements and tasks. The ability to provide a constrained form of programming such as policies is equally important at the individual sensor level. It enables adaptive behaviour of the sensor according to context and thus to also adapt computational requirements and communication and hence power consumption. However, it also provides flexibility to reprogram the sensor with new adaptation strategies without requiring installation of new code.

The requirements we have used have been derived from the need for self-configuration and adaptation in e-Health applications. However, the resulting principles and framework developed are equally applicable to other application areas such as unmanned vehicles, ad-hoc networks, virtual collaborations as well as network and systems management.

The implementation of authorisation policies in both Ponder2 and Tiny Policy Interpreter is currently under way, as we are concerned with trust, security and privacy issues particularly for health-based applications. We are currently investigating security issues for body sensor networks in the CareGrid project [25].

Acknowledgment

We gratefully acknowledge financial support from the UK Engineering and Physical Sciences Research Council (EPSRC) through grants GR/S68040/01, GR/S68033/01 (Amuse Project) and EP/C547586/1 (Biosensornet Project). We also thank Ralf Damaschke for enhancing the Tiny Policy Interpreter during his UROP internship.

References

1. G.-Z. Yang (Ed.) *Body Sensor Networks,* Springer-Verlag, 2006.
2. N. Dulay, E. Lupu, M. Sloman, J. Sventek, N. Badr and S. Heeps. Self-Managed Cells for Ubiquitous Systems. In the *Proceedings of the 3rd International Conference on Mathematical Methods, Models and Architecture for Computer Network Security*, 2005.
3. Ponder2 Documentation Available: www.ponder2.net
4. The BSN Node Specification developed as part of UbiMon project. Available: http://vip.doc.ic.ac.uk/bsn/index.php?m=206
5. S. Strowes, N. Badr, N. Dulay, S. Heeps, E. Lupu, M. Sloman, and J. Sventek..An Event Service Supporting Autonomic Management of Ubiquitous Systems for e-Health, In *Proc. of the 5th Int. Workshop on Distributed Event-based Systems*, Lisbon, Portugal, July 2006.
6. E. Lupu, N. Dulay, M. Sloman, J. Sventek, S. Heeps, S. Strowes, K. Twidle, S.L. Keoh, and A. Schaeffer-Filho. AMUSE: Autonomic Management of Ubiquitous e-Health Systems. *Special Issues of the Journal of Concurrency and Computation: Practice and Experience*, Wiley (In Press).
7. N. Damianou, N. Dulay, E. Lupu, and M. Sloman, In *Proceedings of the International Worksop on Policies for Distributed Systems and Networks (POLICY 2001),* Bristol, UK, Jan 2001.
8. Bluetooth SIG, Inc., Available: http://www.bluetooth.org/
9. Zigbee Alliance, Available: http://www.zigbee.org/
10. TinyOS, Available: http://www.tinyos.net/
11. D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems, In *Proceedings of Programming Language Design and Implementation (PLDI)*, San Diego, June 2003.
12. XMLBlaster, Available: http://www.xmlblaster.org/
13. JMS, Available: http://java.sun.com/products/jms/.
14. B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, Policy Core Information Model Version 1 Specification, Network Working Group, RFC2060, http://www.ietf.org/rfc/rfc3060.txt, 2001.
15. J. Lobo, R. Bhatia, and S. Naqvi, A Policy Description Language. In *Proceedings of the 16th National Conference on Artificial Intelligence,* Orlando, Florida, July 1999.
16. J. Strassner, *Policy-based Network Management*, Morgan Kaufmann, 2004.
17. D. Agrawal, S. Calo, J. Giles, K.W. Lee, and D. Verma, Policy Management for Networked Systems and Applications, In *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management,* Nice, France, May 2005.
18. M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell and K. Nahrstedt, A Middleware Infrastructure for Active Spaces, *IEEE Pervasive Computing,* 1(4):74-83, 2002.
19. D. Garlan, D. P. Siewiorek, A. Smailagic and P. Steenkiste, Aura: Toward Distraction-Free Pervasive Computing, *IEEE Pervasive Computing,* 1(2), 2002, pp. 22 - 31.
20. M. Kumar, B.A. Shirazi, S.K. Das, B.Y. Sung, D. Levine, and M. Singhal, PICO: A Middleware Framework for Pervasive Computing, *IEEE Pervasive Computing,* 2(3):72-79, 2003.
21. D. Malan, T. Fulford-Jones, M. Welsh and S. Moulton. CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care. In *Proc. of the International Workshop on Wearable and Implantable Body Sensor Networks*, April 2004.
22. V. Shnayder, B.-R. Chen, K. Lorincz, T.R.F. Fulford and M. Welsh. Sensor Networks for Medical Care. *Havard University Technical Report TR-08-05,* April 2005.
23. M. Strohbach, H-W. Gellersen, G. Kortuem and C. Kray. Cooperative Artifacts; Assessing Real World Situations with Embedded Technology. In *Proc. of the 6th Intl. Conference on Ubiquitous Computing.* Nottingham, UK, September 7-10, 2004
24. F. Siegemund. A Context-Aware Communication Platform for Smart Objects. In *Proc. of the 2nd International Conference on Pervasive Computing (Pervasive 2004), Vienna, Austria, April 2004.*
25. EPSRC CareGrid Project. Available: http://www.caregrid.org/

Address of the corresponding author:

Author:	Sye Loong Keoh
Institute:	Department of Computing, Imperial College London
Street:	South Kensington Campus,
City:	London, SW7 2AZ
Country:	United Kingdom
Email:	sye.keoh@imperial.ac.uk