CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (in press)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1194

AMUSE: autonomic management of ubiquitous e-Health systems

E. Lupu[1,*,†], N. Dulay[1], M. Sloman[1], J.Sventek[2], S. Heeps[2], S. Strowes[2], K. Twidle[1], S.-L. Keoh[1] and A. Schaeffer-Filho[1]

[1]*Department of Computing, Imperial College London, South Kensington Campus, London SW7 2AZ, U.K.*
[2]*Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow G12 8RZ, U.K.*

SUMMARY

Future e-Health systems will consist of low-power on-body wireless sensors attached to mobile users that interact with an ubiquitous computing environment to monitor the health and well being of patients in hospitals or at home. Patients or health practitioners have very little technical computing expertise so these systems need to be self-configuring and self-managing with little or no user input. More importantly, they should adapt autonomously to changes resulting from user activity, device failure, and the addition or loss of services. We propose the Self-Managed Cell (SMC) as an architectural pattern for all such types of ubiquitous computing applications and use an e-Health application in which on-body sensors are used to monitor a patient living in their home as an exemplar. We describe the services comprising the SMC and discuss cross-SMC interactions as well as the composition of SMCs into larger structures. Copyright © 2007 John Wiley & Sons, Ltd.

Received 1 June 2006; Revised 15 November 2006; Accepted 10 February 2007

KEY WORDS: autonomic computing; pervasive systems; self-configuration; policy-based management

1. INTRODUCTION

Numerous sensors have been developed for monitoring physiological parameters including pulse, heart-rate, body temperature and oxygen saturation, as well as behavioural parameters such as posture and gait [1,2]. Sensors are either wearable or implanted and communicate wirelessly between

*Correspondence to: E. Lupu, Department of Computing, Imperial College London, South Kensington Campus, London SW7 2AZ, U.K.
†E-mail: e.c.lupu@imperial.ac.uk

Contract/grant sponsor: U.K. Engineering and Physical Sciences Research Council; contract/grant numbers: GR/568040/01 and GR/568033/01
Contract/grant sponsor: CEC TrustCom project; contract/grant number: IST-2002-2.3.1.9.1945

themselves and with more powerful wearable processing devices such as mobile phones, PDAs or diagnostic units, which can further interact with a fixed network infrastructure at home, in the hospital or in the street. There is considerable research on the design of new body sensors and measurement techniques, miniaturization of existing sensors and the design of actuator devices such as drug pumps, bio-electrical and bio-mechanical devices [1,2]. There are numerous healthcare applications for these devices including post-operative care (both in the hospital and at home), monitoring of conditions with episodic manifestations such as cardiac arrhythmia, management of chronic conditions such as diabetes mellitus, drug regime monitoring and assistance to elderly patients. The benefits to patients include early release from hospital and improved quality of life, constant monitoring of their clinical condition and well-being, as well as automated alerts and assistance from healthcare personnel when needed. The benefits for healthcare providers include a better service offered to patients, better understanding of the patient's condition, reduced usage of hospital resources and better medical evidence data for the clinical condition and its treatment. Widespread, continuous monitoring of chronic conditions such as cardiac problems will enable medical researchers to accurately determine the conditions that lead to problems. However, whilst the sensors and devices for e-Health are a reality today, the configuration and management of the multiple sensors and software components necessary for these applications still requires considerable technical computing expertise.

Achieving the *autonomic computing* goal [3] of systems that are self-configuring, self-healing, self-optimizing and self-protecting is necessary for ubiquitous e-Health applications. However the challenge is greater than in traditional distributed systems because computational resources on sensors and mobile devices are scarce, the systems are heterogeneous and there is a constant need to adapt to change. A typical scenario for healthcare monitoring is based around a body-area network comprised of multiple sensors and actuators and one or more devices of higher computational capability such as a PDA/mobile phone or diagnostic devices. This body-area network may interact with a variety of other devices depending on its environment. In the home it may interact with servers for storage of medical data and more advanced diagnosis, with home control systems that adapt the home environment to the patients' needs or with devices of healthcare personnel during home visits. In a hospital or GP clinic the body network may interact with other medical devices and may permit those devices to reconfigure its behaviour. In the street it may interact with contextual services or access remote services via cellular networks, for example to request emergency assistance. Across all of these environments the body-area network needs to behave autonomously whilst continuously *adapting* its behaviour according to the patient's clinical condition, the patient's context and interactions with other devices. In short, there is a continuous need for *self-management*.

Traditional network and system management offers a number of techniques for management and adaptation including monitoring, event dissemination and correlation, fault diagnosis and policy-based control, which have often been applied in enterprise networks. However, in enterprise networks, these techniques are functionally integrated providing their results directly to a human systems administrator. This structuring is not suitable for pervasive environments where these systems must integrate locally, providing local feed-back control and adaptation without user intervention.

This paper presents an alternative approach based on structuring the system into *Self-Managed Cells* (SMCs). Each cell is autonomous and must facilitate easy addition or removal of components, cater for failed components and error prone sensors and automatically adapt to the user's current activity, environment, communication capability as well as interactions with other SMCs. Each cell will therefore need to implement a local *feed-back control loop,* and we leverage our previous experience with policy-based techniques in order to provide a flexible mechanism for driving adaptation decisions. An example of a SMC may be the set of sensors, actuators and other devices which form the body-area network for a patient, although more complex devices which manage internal resources may be SMCs in their own right. However, the set of systems available in a smart home or in a hospital also form a SMC and should exhibit autonomous behaviour. A hospital ward, operating theatre and the hospital itself should exhibit similar characteristics. It is therefore desirable to consider the SMC as an *architectural pattern* that can be tailored on instantiation and that can be applied at different levels of scale from body-area networks to large distributed systems. Although the SMC examples used in this paper are aimed at e-Health systems, similar arguments and structures can be defined for intelligent buildings, unmanned autonomous vehicles, intelligent transportation systems and many other applications.

Section 2 describes the SMC architectural pattern and its main components, which are then presented in detail in Section 3. Section 4 discusses aspects relating to interactions across SMCs and composition of SMCs. Sections 5 and 6 present the related work and discuss the current status and outstanding issues.

2. THE SMC

A SMC manages a set of heterogeneous components such as those in a body-area network, a room or even a large-scale distributed application. Different interaction and transport protocols may be necessary in order to interact with each component. For example, interactions with sensors in our prototype occur via IEEE 802.15.4 wireless links whereas interactions with more complex devices such as PDAs, mobile phones or gumstix[‡] typically occur over Bluetooth or WiFi. The SMC must have a unified view for interacting with these components for management purposes and in particular provide a uniform interface for the invocation of management actions. Therefore, adapter objects are instantiated for interacting with each component upon their discovery.

The SMC defines an architectural pattern that applies at different levels of scale from body-area networks to larger distributed and enterprise systems. To this extent it must comprise services that whilst providing the same interface may have different implementations in different SMC instantiations. Since SMCs may need to scale up to larger systems, the set of services that constitute the SMC also needs to reflect the management requirements of these systems and needs to be dynamically extensible. As most management systems are event-driven, we assume that SMCs consist of a set of services that interact using a common publish/subscribe event bus as shown in Figure 1. Although it is not necessary that all interactions be event-based, the use of an event bus confers several advantages. Firstly, it de-couples the services because a sender does not need to know the recipients of an event, thus permitting the addition of new services to the SMC without disrupting the behaviour of existing services. For example, a context service that gathers environment data may be added to mobile SMCs or an auditing service may be added to SMCs that require records to be kept of interactions that have occurred. Similarly, security services that perform anomaly detection, and support authentication and confidentiality as well as optimization services which try to optimize performance according to a utility

[‡]http://www.gumstix.com.

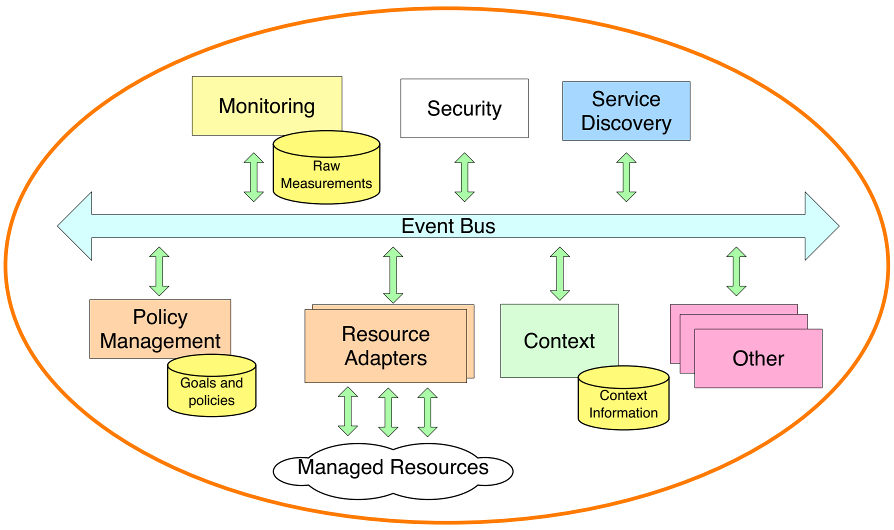

Figure 1. The SMC architectural pattern.

function, could be added in more complex SMCs. Secondly, an event bus allows multiple services to respond concurrently and independently to the same notifications with different actions. For example, when a new sensor is discovered a policy service may initiate its configuration whilst a diagnostic service would take into account the additional input received from that sensor. Finally, the event bus can be used for both management and application data such as alarms indicating that thresholds have been exceeded, e.g. for heart-rate or oxygen saturation. More generally, an event-bus architecture is well suited to adaptive ubiquitous systems which are essentially event-driven as changes of state in resources need to be notified asynchronously to several, potentially unknown, recipients. An event may indicate the discovery of a new component, component failure, change in context or medical condition, e.g. ECG anomaly detected. We have developed a simple publish–subscribe event system supporting at-most-once persistent event delivery in which the service attempts to deliver the event until it knows that the subscriber is no longer a member of the SMC. Interactions between management components are typically event-based in order to benefit from the extensibility they support. However, we do not insist that all interactions take place via the event bus and in particular interactions between application components can be based upon other communication paradigms such as simple point-to-point messages or remote invocations.

Figure 1 represents the SMC architectural pattern with an extensible set of services communicating through the event bus as well as the management and control adapters to the managed resources. Although the set of services may change depending on the context in which the SMC is instantiated (e.g. body-area network, home control system, hospital) a number of services constitute the core functionality of the SMC and must always be present. These include the *event bus*, a *discovery service* and a *policy service*. In our current prototype these services are implemented in Java and run on either a PDA or a gumstix device.

The *discovery service* is used to discover nearby components that are capable of becoming members of the SMC, for example intelligent sensors, and other SMCs when they come into communication range. It interrogates the new devices to establish a profile describing the services they offer and then generates an event describing the addition of the new device for other SMC components to use as appropriate. It also maintains a list of known devices as we have to cater for mobile wireless components which may wander in and out of communications range and distinguish such occurrences from permanent departures from the cell.

The SMC's adaptation strategy for self-management is achieved through a *policy service* that implements a basic feed-back control loop. As shown in Figure 2 changes of state in managed objects are disseminated in the form of events through the event bus. The policy service performs reconfiguration actions and caters for two types of policies: *obligation policies* (event–condition–action rules) that define which configuration actions must be performed in response to events and *authorization policies* that specify which actions are permitted on which resources or devices. Policies can be added, removed, enabled and disabled to change the behaviour of cell components without applying code modifications and may also be used to enable or disable other policies. For example, the following policies could be specified for a body-area network of sensors monitoring the recovery of a patient with a cardiac condition:

```
1. on hr(level) do
            if level > 100 then
                  /os.setfreq(10min); /os.setMinVal(80)
2. on context(activity) do
            if activity == "running" then
                  /policies/normal.disable(); /policies/active.enable()
3. auth+ /patient → /os.{setfreq, setMinVal, stop, start}
4. auth+ /patient → /policies.{load, delete, enable, disable}
```

Policy 1 is triggered by a heart rate (hr) event as measured by a heart rate sensor and sets the frequency for monitoring oxygen saturation (os) as well as new thresholds for the generation of events from these measurements. When the heart rate is above 100 the oxygen saturation should be checked every 10 minutes and an alarm should be generated if the value is below 80. Policy 2 assumes the existence of a context sensor notifying the SMC of the patient's current activity. When the patient is running the heart rate may increase naturally so policies applying to the normal mode of operation should be disabled and policies specific to strenuous physical activity should be enabled. Policies 3 and 4 are the required authorizations to permit management of the oxygen saturation monitor and of the policies themselves.

Based on the lessons learnt from our previous work on policy specification [4,5] we have developed a new light-weight policy service appropriate for limited-resource devices. Both the discovery service and the policy service are described in more detail in Section 3. The implementation of the policy service is also available in open source form at: http://www.ponder2.net.

The policies defined above are specified upon the instantiation of the SMC at a time when an oxygen saturation sensor may not exist. Policies are specified in terms of *roles* which act as placeholders

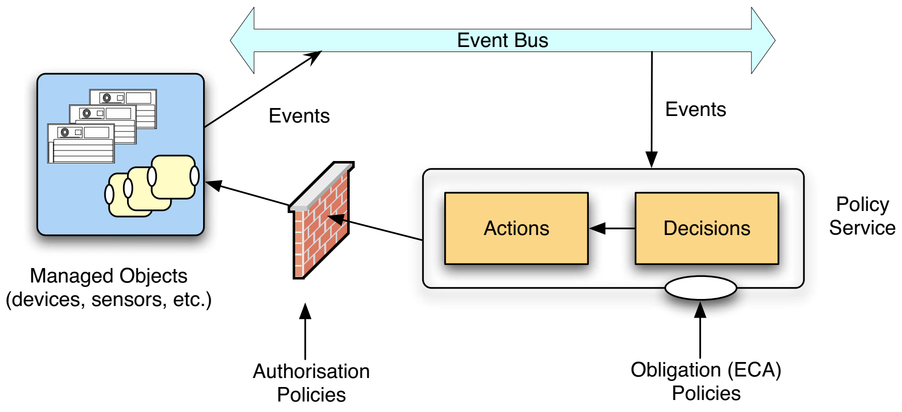

Figure 2. Policy-based feed-back loop.

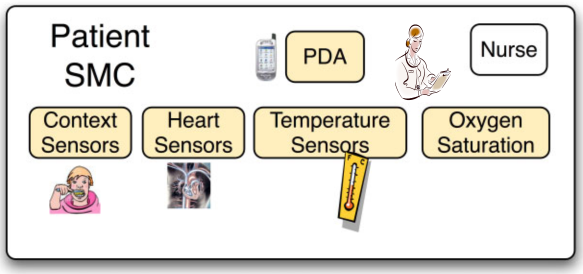

Figure 3. Patient SMC roles.

for components within the SMC. Roles are associated with interfaces, which define the methods that components must provide and events that those components can raise or that can be sent to them. This allows policies to be written in terms of the actions and events on those interfaces. When a new device, a sensor or another SMC, is discovered, it can be dynamically assigned to a role and policies defined for that role would apply to that device. Several devices may be assigned to the same role and any policy actions would then be performed on all the devices associated with that role. We use the term *mission* for the set of policies relating to a role which is loaded onto a remote SMC that is capable of interpreting them, as explained in Section 4. Figure 3 indicates typical roles for a patient SMC. Sensors such as heart and temperature sensors monitor the physiological condition of the patient while context sensors report on the patient's current activity. This is necessary in order to avoid mistaking an increased heart rate due to physical activity for the symptoms of an impending heart attack. The nurse role will allow interactions with a nurse's SMC as explained in Section 4.

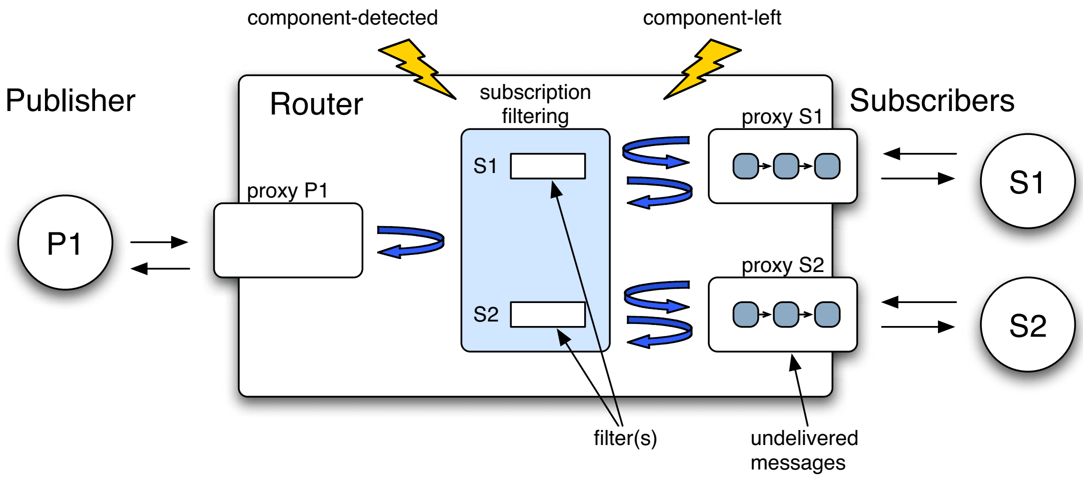

Figure 4. Event service architecture.

3. THE SMC ARCHITECTURE SERVICES

3.1. The discovery service

The discovery service is responsible for detecting new devices or other SMCs when they come into communication range. It is responsible for maintaining the membership of the SMC and informs the other services when devices have joined or permanently left the SMC by generating the *component-detected* and *component-left* events as shown in Figure 4. The discovery service is designed to mask transient disconnections from the SMC. Because the discovery service determines when a component has joined the SMC, it also carries out any admission control (vetting) for accepting the device in the SMC based on the device's profile and any authentication information available. By making the discovery service policy-driven, it can easily be adapted to different applications.

We have implemented a simple device discovery service for our e-Health testbed which can run on a PDA, a gumstix or a mobile phone. Although a number of protocols for service discovery already exist [6] we needed an implementation that can scale down to the body sensor nodes (BSNs)[§] that we are using and possessed the flexibility to customize the discovery service through policies. The discovery service broadcasts its *identity message* (id; type[; extra]) at frequency R. This enables the SMC to advertise itself to both devices and other SMCs, enables current SMC members to determine

[§]These BSNs were developed in the DTI Ubimon project (see http://www.ubimon.org). They have very low-power 16-bit processors, 64 KB RAM, 256 KB Flash memory, six analog channels for sensors and communicate using IEEE 802.15.4 radio. These sensors may need to survive for long periods of time without battery replacement.

that they are still within reach of the SMC and avoids having the discovery service in listening mode at all times to detect advertisements from new devices. A new device responds to the identity message with a unicast device identity message. The discovery service can then query the device to obtain a device profile, performs vetting procedures if required, informs the device whether it has been accepted for membership and if so generates a *component-detected* event which results in the device being registered and classified in the policy interpreter's domain structure (see Section 3.3) with a device specific adapter being created for that device (see Figure 4).

Each existing member device unicasts its identity message to the discovery service at the frequency D. If the discovery service misses n_D successive messages from a particular device, it concludes that the device has left the SMC permanently, and generates a corresponding *component-left* event. This event will trigger the removal of any notifications addressed to that device from the event service and the removal of any adapters and role references corresponding to that device in the policy service.

Once a device joins a SMC, it will not respond to discovery service broadcasts from other SMCs for as long as its membership in the SMC lasts. Membership can be terminated either by the device itself, the SMC (e.g. by performing a reset of the device) or if the device misses n_R successive identity broadcasts from the discovery service of the SMC to which it is bound. Healthcare scenarios often also require a more permanent form of membership. A health monitoring sensor should not decide that it has left the SMC because there is a problem with the discovery service and then join the SMC of the person sitting next to the patient on the bus. One approach is to use pairing through physical contact or explicit actions (e.g. as with simple Bluetooth devices) in order to set up more permanent associations. We are investigating other more secure techniques in the Caregrid project [7].

3.2. The event bus

Events are a critical aspect of the SMC as they trigger policies that adapt the SMC's behaviour. However, as stated previously, not all interactions need to occur via the event bus. The event bus has been implemented as an at-most-once, persistent publish/subscribe delivery service, using a router to distribute events to subscribers [8]. The router supports content-based subscriptions. Subscribers register to receive notifications of event occurrences and specify a filter, which is matched against the events received by the router. All events that match the filter are forwarded to that subscriber. In the current implementation, publishers do not need to register with the event router, thereby allowing simple sensors to send notifications directly without additional registration overhead. However, this has disadvantages as it would be desirable to inform publishers with no current subscriptions that they do not need to send notifications, thereby enabling them to save power. This feature, also known as *quenching* (see, e.g., Elvin [9]) is planned for future revisions of the implementation.

The event bus (Figure 4) must guarantee reliable delivery of events since the events are used to trigger adaptation and re-configuration actions and this is also required by medical applications. Furthermore, it must guarantee that messages from the same publisher will be delivered to the subscriber in the same order as they have been received by the router. This is required as events from the same publisher may be causally related. To achieve this, all messages (including subscription messages) are acknowledged when received by the router or the subscribers. The event router maintains proxy objects for all publishers and subscribers connected to the bus. These proxies fulfil two functions: first they buffer events that have not yet been received by subscribers, and second they adapt to the specific communication protocols used by publishers and subscribers. For example, body sensor nodes communicate via basic messaging over 802.15.4 whereas gumstixs and PDAs may communicate using datagram protocols over Bluetooth. Buffering of events is necessary in order to mask transient communication failures.

As shown in Figure 4 when the event bus receives notification that a new component has been added to the SMC, it instantiates a proxy for that device. The type of proxy to be instantiated is determined according to the device profile as established previously by the discovery service. Event occurrences are then notified to the router. Successful delivery of an event to the router causes that event to be delivered to the proxies whose filters match the event. Each proxy maintains a first in first out (FIFO) queue of events and attempts to deliver the event at the head of the queue periodically until it is successful or it learns that the device is no longer a member of the SMC. If the router receives a component-left event (from the discovery service), it removes that subscriber's filters and deletes the proxy for that subscriber; the destructor for the proxy purges any events in its FIFO queue.

3.3. The policy service

We have had considerable experience with the use of policies as a means of specifying adaptive behaviour in network management and other applications. The use of interpreted policies means they can be easily changed without shutting down or recoding components. The policy service maintains adapter objects for each of the components on which management actions can be performed. This includes the sensors and other devices present within the SMC, services within those devices and remote SMCs. These adapter objects (also called managed objects) are grouped in a domain structure that implements a hierarchical namespace, i.e. similar to a file system. However, unlike in a file system, domains may overlap and a managed object may belong to several domains. Domains, policies and roles are managed objects in their own right on which actions can be performed, for example, adding/removing an object from a domain, enabling or disabling a policy. Consequently, events can trigger obligation policies (ECA rules) that can enable or disable other policies and change domains and domain membership [10]. In essence, domains are a means of classifying and grouping the managed objects in a hierarchy and permit them to be addressed using simple path expressions.

We are concerned primarily with two types of policies: *authorization policies* that define which actions are permitted under given circumstances and *obligation policies* that define which actions should be performed in response to an event occurring if specific conditions are fulfilled (event–condition–action rules). Authorization policies should be enforced on the target components they are protecting as these must make the decision whether to permit or deny access. For example, a policy of the form

auth+ /sensors/temperature →/pda.reportTemp

would be needed to permit temperature sensors to perform the reportTemp operation on the pda. Obligation policies are implemented by the policy service or loaded into a remote policy service as part of a mission. For example, the following policy specifies that the oxygen saturation sensor should be activated when the heart rate is above 100:

on heartrate(hr) **do if** hr > 100 **then** /sensors/os.activate()

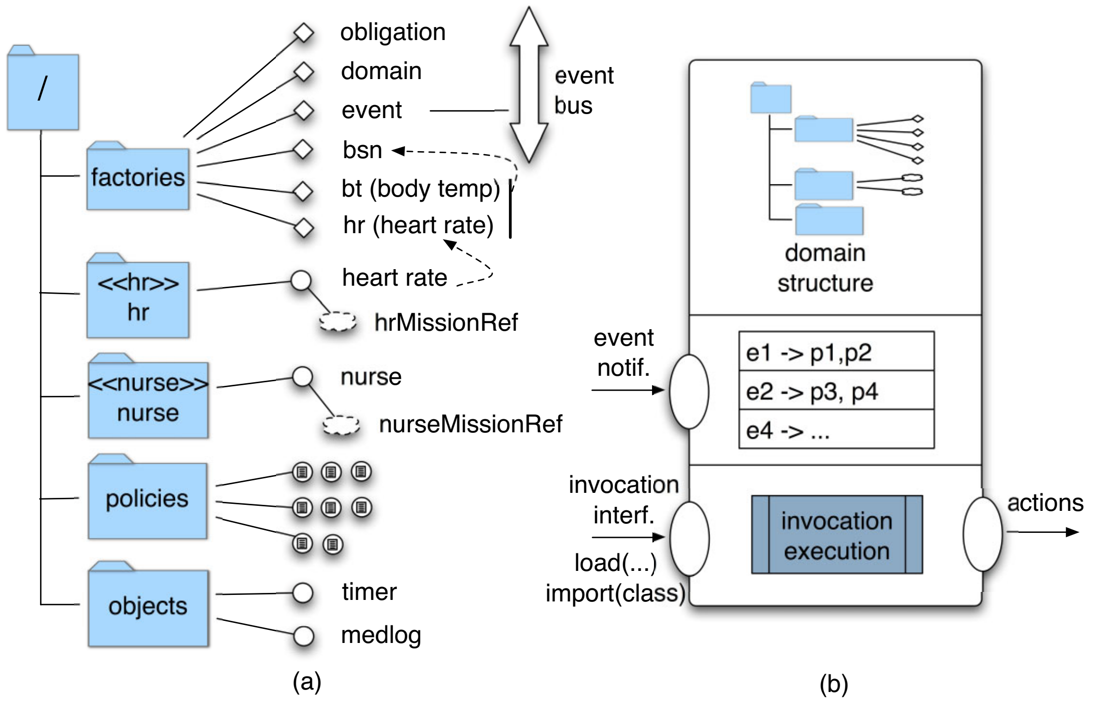

Figure 5. Policy service: (a) example domain structure and (b) overall architecture.

When an obligation policy is created in the policy service, an event subscription for that event is sent to the event bus. Upon receiving a notification, the policy service evaluates all the obligation policies triggered by that event.

The policy service has been implemented with particular focus on flexibility and 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 which are necessary in each case. When started, the policy service has a reference to its *root* domain and only recognizes the import command that can load new classes. Typically, the classes loaded are factories that permit the creation of new objects in domains and the first class to be loaded is the factory for the domain objects themselves (Figure 5). This enables the policy service to create new domain objects to form a hierarchy of domains under the *root* domain. In addition, factory objects are then loaded in order to communicate with the event bus, create policies and create adapters for the various sensors and devices in the SMC. The *event factory* is specific to the event bus and encapsulates the protocols necessary to communicate with it. However, multiple event factory objects can be created, allowing the policy service to connect to different event buses with different underlying protocols, for example XMLBlaster. Similarly, new types of policies, for example delegation, filtering, etc., can be defined by providing and dynamically loading the corresponding factory. The BSN factory object (shown in Figure 5(a)) encapsulates the code for interacting with a BSN sensor node using IEEE 802.15.4 radio. Specific factories can then be defined for each of the different types of BSN sensors in use, for example the hr sensor for monitoring heart rate which uses the basic BSN adapter for interaction with BSN nodes. The same principle has been used in other application areas to use Java RMI and SOAP to interact with remote services. When a *component-detected* event appears on the event bus, policies determine which factory is used to create the adapter object and in which domain the object will be placed. Other policies specified for that domain will then automatically apply to the new component, for example,

```
on component_detected(id, profile, addr) do
        if profile == "heart rate" then
                r = /fact/hr.create(profile, addr); /sensors.add(r)
```

If the component is another SMC a similar policy will subsequently select the appropriate *mission* and load it in the remote SMC. This operation returns a reference to the remote mission, which is placed within the adapter object for the remote SMC as explained below. Figure 5(a) also shows the other elements of a typical domain structure within a Ponder2 policy service [11]. These include the *policies* domain in which by default all obligation policies are stored and domains corresponding to the SMC roles such as the *nurse* domain.

As shown in Figure 5(b) the overall architecture of the policy service comprises the domain structure, the table matching obligation policies to events 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, an invocation interface through which external invocations are received (e.g. to load a mission) and an action interface through which calls are made to external objects.

4. INTERACTIONS BETWEEN SMCs

The components described in the previous section define a basic SMC that can discover and manage simple sensors and devices. To scale the SMC architectural pattern to larger systems there is a need to cater for cases in which managed resources are themselves SMCs and to provide techniques for composing SMCs. Even in a body-area network for health monitoring, some sensors may permit a constrained form of programming in terms of policies or more complex diagnostic devices may be SMCs in their own right managing their own resources. Two basic types of interactions are of interest to us: *composition* and *peer-to-peer* interactions.

Composition interactions occur when a managed resource or device within an SMC is itself an SMC with its own resources and devices. This implies that the device can itself be programmed by the containing SMC in terms of policies that it must enforce. Moreover, the device may expose to its containing SMC a management interface for re-configuration. For example, a diagnostic device may be part of a body-area network SMC and will allow that SMC to load new decision algorithms and new policies into it. Composition also implies that the contained SMC ceases to advertise itself independently but will rely on the containing SMC to bind it with other devices and SMCs with which it needs to interact. Only the containing SMC will have access to its management interface.

Peer-to-peer interactions occur, for example, when a nurse or other health worker visits the patient at home. The nurse would typically have their own PDA and medical devices that need to interact with the body-area network monitoring the health of the patient. In particular, the nurse SMC might need to be notified of events occurring within the patient SMC, and may need to load policies for execution by the patient, for example for defining new thresholds or alert behaviour. Similarly, when discovering the presence of the nurse SMC, the patient SMC may need to load policies onto the nurse SMC to trigger re-calibration of the patient sensors if needed. This avoids the requirement for the nurse PDA to store calibration procedures for all possible patient sensors. Peer-to-peer interactions are not necessarily between 'peers' at the same level of abstraction. For example, a medical monitoring service may use a wireless communication service as well as a storage area network for storing large quantities of monitored data in a layered interaction style.

A *mission* defines the requirements of one SMC for interacting with another. It is a group of policies which define the duties of the remote SMC in terms of the obligation policies it must enforce. These obligation policies are written in terms of the *mission interfaces* for each SMC. Mission interfaces specify the following information.

- *Events*—these are events that are available to the loaded mission and can trigger the mission's policies. Typical examples of such events are local timers or events from the SMC hardware, as well events from remote SMCs.
- *Notifications*—these are *events* that the mission can raise within either the local SMC in which it has been loaded or the remote SMC.
- *Local actions*—that may be invoked by the mission's policies in the SMC in which it has been loaded. These may be actions on local resources such as hardware sensors or actuators.
- *Remote actions*—that may be invoked by the mission's policies either when the mission is running locally or in a remote SMC.

The events and actions specified in the mission interfaces define a scope for specifying mission policies. The concept of a mission and mission interface is common to both composition and peer-to-peer interactions, although a mission interface for peer-to-peer interactions is likely to be more limited in the range of actions and events defined compared to that available to a containing SMC. In both cases, authorization policies are needed to specify which SMCs are permitted to load a mission, invoke remote actions and subscribe to receive events.

When an SMC discovers a new SMC with which it wants to collaborate, it will first assign that SMC to the corresponding role in its structure and will instantiate the required missions at the remote SMC. A reference to the remote mission will be kept as part of that SMCs role. For example, as shown in Figure 6, when the nurse SMC discovers the patient's body-area network SMC it instantiates a *patient mission* on the patient's SMC if permitted by the latter. Similarly, upon discovery of the nurse SMC the patient may instantiate at the nurse a mission defining the policies it expects the nurse to fulfil, if permitted. Note that it is not necessary that both missions are initiated. In many cases of composition only the external SMC will instantiate a mission in the managed SMC but not *vice versa*.

The *patient mission* loaded by the nurse into the patient relies on the following interfaces (Figure 7): the nurse SMC must expose actions for storing monitoring data and displaying an ECG as well as the ability to be notified that the ECG is currently in progress. The patient generates the events notifying the following: (1) that the mission has been loaded; (2) the heart rate (hr) value; and (3) endECG() to notify that the recording has finished. It also must provide the actions needed to read the recording logs, raise a timer event and read the ECG data.

The mission specification is parameterized by the nurse and patient interfaces (nurse and patient) as shown in the following mission specification:

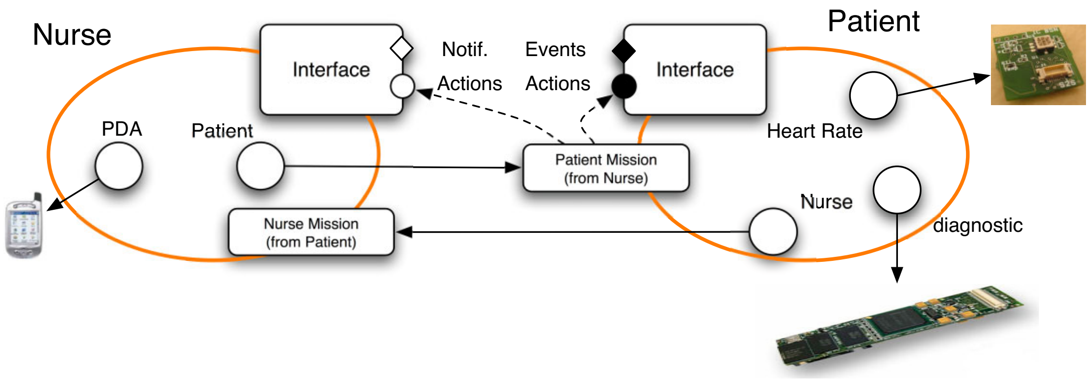

Figure 6. Missions across SMCs.

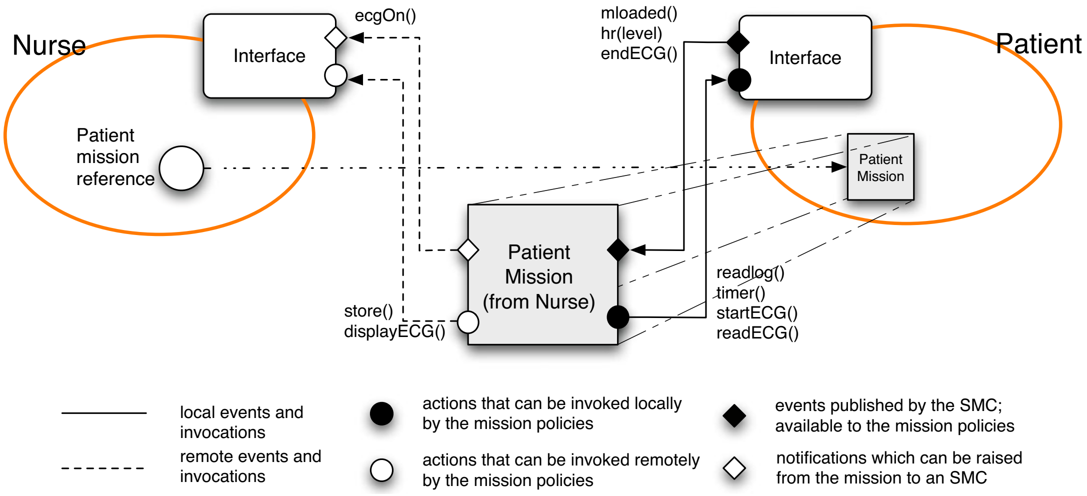
Nurse
Interface
ecgOn()
mloaded()
hr(level)
endECG()
Interface
Patient
Patient mission reference
Patient Mission
Patient Mission (from Nurse)
readlog()
timer()
startECG()
readECG()
store()
displayECG()
local events and invocations
remote events and invocations
actions that can be invoked locally by the mission policies
actions that can be invoked remotely by the mission policies
events published by the SMC; available to the mission policies
notifications which can be raised from the mission to an SMC

Figure 7. Mission invocations.

```
mission patientT(nurse, patient, ECGLevel, ECGTime ) do
  1. on patient.mloaded() do
                nurse.store(patient.readlog())
  2. on patient.hr(level) do
          if level > ECGLevel then
                patient.startECG()
                patient.timer(ECGTime, endECG())
                nurse.ecgOn()
  3. on patient.endECG() do
                nurse.display(patient.readECG())
```

The patient's SMC generates a mloaded() event after the mission has been successfully loaded. This triggers Policy 1 to read the patient's data logs and then to store them in the nurse SMC. Policy 2 is triggered by heart rate events generated by the patient's SMC. It specifies that if the heart rate becomes greater than ECGLevel an ECG should be started on the patient SMC for the duration ECGTime. The *timer* action requests the patient SMC to generate the endECG event after ECGTime has elapsed. The ecgOn event informs the nurse that the ECG reading has started. Note that ECGTime and ECGLevel are parameters that are provided to the mission upon instantiation as they may be specific to the patient being visited. Policy 3 specifies that when the ECG is finished it must be read from the patient's SMC and be displayed on the nurse's SMC. The mission is instantiated at the patient's SMC by the nurse who provides the references to the SMC instances along with the values for ECGTimer and ECGLevel. Thus the nurse SMC would typically have an obligation policy to trigger the instantiation:

```
on newPatient(p) do
      ref = p.loadMission(/patients.interface, p.interface, 82, 40); /roles[p].add(ref)
```

Authorization policies are needed in both SMC's to permit the actions to be invoked. In the nurse's SMC the following authorizations are needed:

```
auth+ /patient → /nurse.store
auth+ /patient → /nurse.displayECG
```

In the patient's SMC the following authorization policy is needed:

```
auth+ /nurse → /patient.loadMission
```

It is assumed that authorization policies are not needed for local actions performed as part of the mission. Actions to load, remove, enable and disable missions are provided in the management interface of all SMCs. The ability to load policies into a remote SMC facilitates the customization of interactions between SMCs and permits the dynamic programming of the behaviour of a contained SMC. However, this implies that the receiving SMC is able to verify the received policies in terms of their format, semantics and conflicts with other policies before accepting them. The design and implementation of this verification process remains to be investigated.

5. RELATED WORK

IBM has been the prime mover towards autonomic computing [3] and HP is also addressing similar issues in on-demand Utility Data Centres [12]. However, most of the industrial work focuses on large clusters and Web servers whereas we are concentrating on pervasive computing which is potentially more dynamic owing to the mobility of components. The Universal Plug and Play (UPnP) Architecture supports resource discovery and configuration of consumer devices (TV, video recorder, air conditioning etc.), which communicate via wireless connections within a home or office [13]. It concentrates on device configuration rather than the configuration of software within nodes and does not support the adaptability required for healing, optimizing or protecting.

There are a number of pervasive systems projects that define frameworks for realizing pervasive spaces [14,15]. In general, these projects tend to focus on spaces of relatively fixed size such as a room or a house and tend to focus on specific concerns such as context-related applications, user presence and intent or foraging for computational resources. Although they recognize the federation and composition of spaces as a main concern, little work has been done in that direction. In contrast, we consider an architectural pattern at different scales and focus on generic adaptation mechanisms (i.e. through policies). Cross-SMC interactions are a basic feature for our model, although further elaboration is needed.

There are many discovery services for both fixed and *ad hoc* networks including UPnP [13], SLP [16], Jini [17], SDP [18], Zeroconf [19], Konark [20], DEAPspace [21]—for a recent survey and taxonomy of service discovery see Zhu *et al.* [6]. These protocols vary substantially in their infrastructure (directory based versus directory less), language dependence and transport protocol dependence. Some of them provide membership management whilst others focus on providing device reconfiguration functionality. We needed a very simple protocol that could be easily implemented on simple sensors such as BSN nodes or even simpler and that could be policy driven so that different actions can be taken when devices are discovered based on the policy specification. We therefore implemented the protocol described in this paper. However, other protocols could be used in different applications in conjunction with the policy service instead.

There are many publish/subscribe event services such as Elvin [9], XMLBlaster [22], Gryphon [23,24], JMS [25] and Sienna [26]; unfortunately, none of these routers are designed to run on small devices such as BSNs and PDAs. After experimenting with several, including Elvin, XMLBlaster and Sienna, we have implemented our own service for use with the body-area network of BSN. However, in different applications that are less restricted in power consumption and computational capabilities we have also used the policy service in conjunction with XMLBlaster, Siena and others. As explained in Section 3.3, the policy service can use different event services simultaneously through different adaptors and act as a gateway between them.

Work on policy-driven systems has been going on for over a decade in various application areas. Traditional approaches rooted in network and systems management include PCIM [27], PDL [28], NGOSS Policy [29], Ponder [4] and PMAC [30]. They have a common base in the 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 implementations on small devices and sensors. Other approaches have been aimed at interactions between distributed agents and include LGI [31], KAoS [32] and Rei [33], although they all have a slightly different focus. LGI policies use a simple Prolog notation to specify the actions that agents must undertake upon the receipt or sending of messages. It assumes that policies are interpreted by trusted controllers at each agent's site. KaOS is a collection of component-based agent services developed to support mobile agents and subsequently extended for Grid computing and Web Services environments. Policy specification takes an ontology-based approach and policies are represented using the DAML notation [34]. Rei, like KAoS, follows an ontology approach to policy representation although, as in LGI, policies can be specified in a simpler Prolog-based notation. The enforcement is based on a decision engine that uses deductive reasoning to infer the rights and obligations of objects in the managed system in response to requests that specify the current state of the system. Finally, a number of approaches such as XACML [35] focus on authorization policies alone. Our approach is based on our experience with the design and implementation of the Ponder system but is intended to cater for simple pervasive computing devices as well as larger distributed systems.

6. CURRENT STATUS AND FUTURE WORK

We have built implementations of the core SMC services including the event service, the discovery service and the policy service [11] which run on both PDAs and gumstix devices. We have also implemented the client side of the service discovery protocol and adapter objects for interacting with BSN nodes that can currently host a variety of sensors for temperature, acceleration, monitoring heart rate and oxygen saturation. A simple version of the policy service also runs on BSNs. This has enabled us to build concrete demonstrators in which sensors can be discovered and configured by the SMC dynamically and policies can be changed dynamically and applied automatically to the devices. Although the underlying XML-based policy service and protocols for federating and exchanging policies between SMCs already exists, the higher-level language model supporting mission specifications and remote instantiation of mission specifications is in the process of being implemented. We intend to demonstrate SMC federation between 'smart-devices'—themselves SMCs—and body-area networks as well as between a home network and a body-area network.

Cross SMC interactions also require further work. Although the concept of *mission* enables us to deploy sets of policies (missions) to remote SMCs, further work is needed to define more complex SMC-based structures which recursively compose. This will be necessary in order to apply the SMC paradigm to larger configurations such as sensor networks or autonomous fleets of unmanned vehicles undertaking complex missions such as search and rescue. Further work is also needed on the composition of SMCs and interactions between SMCs at different layers of abstraction. A SMC should be able to expose an abstract interface realized by aggregating the functionality of its resources and needs to fulfil abstract goals that it must refine and achieve in terms of its resources and interactions with other SMCs. This requires techniques such as planning and policy refinement. Although we have previously worked on policy refinement [36], these techniques require significant computational resources and work is needed to scale these approaches down to PDA-like devices.

In order to evaluate the applicability of the SMC as an architectural pattern in different contexts we are in the process of developing solutions based on the SMC architecture for both unmanned autonomous vehicles [37] and large-scale distributed systems in the form of Virtual Organizations [38]. In the latter case the policy-service has been used in a large collaborative project developing a framework for Virtual Organizations, which provides service composition based on Web Services in conjunction with different discovery services and using different invocation paradigms, for example SOAP, WSRF/WSDM, etc. Policies are used within this context in order to change configurations on the virtualization points of services and to specify how the Virtual Organization should react to changes in membership, violation of service level agreement parameters and changes in the reputation and reliability of participants.

The work presented in this paper focuses on the adaptive systems infrastructure, the use and the implementation of the SMC architectural pattern. Patient trials to determine user acceptance of healthcare monitoring, what sort of feedback to users is required, performance in real environments, the issues related to interaction between social workers and healthcare personnel in the use of sensor-based home monitoring, as well as the business processes related to the commercial infrastructure, are being addressed in the DTI funded Saphe project [39].

There is a wide range of security and trust issues to be addressed, particularly for health-based applications. Traditional aspects of authentication and access control are challenging because devices have limited capabilities for cryptographic operations or for the evaluation of complex access control policies. The issuing and verification of credentials is difficult to achieve computationally but SMCs still need to interact with other devices and SMCs with which they share no previous knowledge or key material. In addition, in health-based scenarios there is a constant need to adapt the trade-off between protection, patient privacy and access to services depending on current context. For example, in an emergency situation having access to the monitored data is of paramount importance but the SMC may not have access to a networked infrastructure to decide that the person attending is a genuine health-care professional. Similarly, medical information is exchanged between SMC devices across wireless links that can be trusted to different extents. Issues of trust are therefore particularly important as security decisions have to be taken with very limited user intervention or none at all, based on limited information and according to current context. Information relating to trust, recommendations, reputation and experience of previous interactions with the other party are likely to contribute to the decision. Although we have started working on these issues [40,41] further work is ongoing within the CareGrid project [7].

7. CONCLUSIONS

We have proposed the SMC structure as a basic architectural pattern that aims to provide local feed-back control and autonomy and advocated the realization of more complex systems through the composition and peer-to-peer interactions between SMCs. Our particular focus is on health care monitoring as this provides a wide range of benefits to the public at large whilst presenting numerous challenges in terms of both SMC functioning and interactions. However, we are also working towards applying the SMC architecture in other scenarios and thus demonstrating its wider applicability.

The use of the event bus as the primary means of exchanging management information de-couples architectural components and provides the basis for extending the functionality of the SMC by adding additional services (e.g. context, accounting, etc.) as well as scaling the SMC to larger numbers of managed resources by using alternative implementations of the SMCs services. For example, we are experimenting with different event buses such as XML Blaster and Siena for larger distributed systems and our own for body-area networks.

Policies, in particular in the form of event–condition–action rules, provide a simple and effective encoding of the adaptation strategy required in response to changes of context or changes in requirements. The ability to dynamically load, enable and disable the policies together with the ability to use policies in order to manage policies caters for a wide variety of application needs.

However, this also raises a number of issues to be addressed: (i) How to verify dynamically loaded policies to determine their acceptability? (ii) How to structure large sets of policies in terms of meaningful higher level abstractions? Whilst the concept of missions introduced in this paper attempts to provide an initial answer to the latter question, it needs to be further refined to cater for more complex configurations. Although a spectrum of techniques can be used for the verification of dynamically loaded policies ranging from simple authorization to conflict analysis and sandboxing, further work is needed to determine how these techniques can be used in conjunction with one another and in which circumstances. Event–condition–action rules can express most desired behaviours and the temptation is often strong to use the obligation policies as a general programming paradigm. However, this leads to programs that are awkward to write and difficult to maintain as it is difficult to keep track of the propagation of information between functions through events. Policies should be used for encoding the 'strategy' for adaptation, i.e. which adaptation actions to perform, not the actions themselves.

ACKNOWLEDGEMENTS

The authors wish to thank the U.K. Engineering and Physical Sciences Research Council for their support of this research through grants GR/S68040/01 and GR/S68033/01. We would also like to acknowledge financial support under the CEC TrustCoM project IST-2002-2.3.1.9.1945.

REFERENCES

1. Yang G-Z (ed.). *Body Sensor Networks*. Springer: London, 2006.
2. *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2006)*, MIT, Boston, MA, April 2006. IEEE Computer Society Press: Los Alamitos, CA, 2006.
3. Kephart JO, Chess DM. The vision of autonomic computing. *IEEE Computer* 2003; **36**(1):41–50.
4. Damianou N, Dulay N, Lupu E, Sloman M. The Ponder specification language. *Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2001)*, Bristol, U.K., January 2001 (*Lecture Notes in Computer Science*, vol. 1995). Springer: Berlin, 2001; 18–39.
5. Damianou N, Dulay N, Lupu E, Sloman M, Tonouchi T. Tools for domain-based policy management of distributed systems. *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Florence, Italy, April, 2002; 203–217.
6. Zhu F, Mutka MW, Ni LM. Service discovery in pervasive computing environments. *IEEE Pervasive Computing* 2005; **4**(4):81–90.
7. EPSRC CareGrid Project. http://www.doc.ic.ac.uk/%7End/projects/CareGrid.html [30 May 2006].
8. Strowes S, Badr N, Dulay N, Heeps S, Lupu E, Sloman M, Sventek J. An event service supporting autonomic management of ubiquitous systems for e-Health. *Proceedings of the 26th International Conference on Distibuted Computing Systems Workshops (ICDCSW'06): 5th International Workshop on Distributed Event-Based Systems (DEBS)*, Lisbon, Portugal, July 2006; 22–27.
9. Elvin. http://www.mantara.com/ [30 May 2006].
10. Lymberopoulos L, Lupu E, Sloman M. An adaptive policy-based framework for network services management. *Journal of Network and Systems Management (Special Issue on Policy-Based Management)* 2003; **11**(3):277–303.
11. Ponder2 Policy System. http://www.ponder2.net [6 November 2006].
12. HP Utility Data Center: Enabling Enhanced Datacenter Agility, May 2003. http://www.hp.com/large/globalsolutions/ae/pdfs/udc_enabling.pdf [25 May 2005].
13. Universal Plug and Play Device Architecture. http://www.upnp.org/resources/documents.asp [30 May 2006].
14. Roman M, Hess C, Cerqueira R, Ranganathan A, Campbell R, Nahrstedt K. A middleware infrastructure for active spaces. *IEEE Pervasive Computing* 2002; **1**(2):22–31.
15. Garlan D, Siewiorek DP, Smailagic A, Steenkiste P. Project Aura: Toward distraction free pervasive computing. *IEEE Pervasive Computing* 2002; **1**(2):22–31.
16. Guttman E, Perkins C, Veizades J, Day M. Service location protocol, version 2. *Request for Comments 2608*, June 1999.
17. Sun Microsystems. Jini Architectural Overview. http://www.sun.com/jini [30 May 2006].

18. Bluetooth SIG, Specification of the Bluetooth System. Volume I: Core Specification, February 2001. http://www.bluetooth.com/dev/spec [30 May 2006].
19. IETF Zeroconf Working group. Zero Configuration Net-working. http://www.zeroconf.org/ [30 May 2006].
20. Helal S, Desai N, Verma V, Lee C. Konark—A Service discovery and delivery protocol for *ad-hoc* networks. *Proceedings of the 3rd IEEE Conference on Wireless Communication Networks (WCNC)*, vol. 3, New Orleans, LA, March 2003; 2107–2113.
21. Nidd M. Service discovery in DEAPspace. *IEEE Personal Communications* 2001; **8**(4):39–45.
22. xmlBlaster.org. http://www.xmlblaster.org/ [30 May 2006].
23. Zhao Y, Sturman D, Bhola S. Subscription propagation in highly-available publish/subscribe middleware. *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, Toronto, Canada, October 2004; 274–293.
24. Bhola S, Strom S, Bagchi S, Zhao Y, Auerbach J. Exactly-once delivery in a content-based publish–subscribe system. *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, Bethesda, MD, June 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 7–16.
25. Sun Microsystems. Java Message Service. http://java.sun.com/products/jms/ [30 May 2006].
26. Carzaniga A, Rosenblum D, Wolf AL. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems* 2001; **19**(3):332–383.
27. Moore B, Ellesson E., Strassner J, Westerinen A. Policy core information model—Version 1 Specification. *Request for Comments 3060*, Network Working Group, 2001. Available at: http://www.ietf.org/rfc/rfc3060.txt.
28. Lobo J, Bhatia R, Naqvi S. A policy description language. *Proceedings of the 16th National Conference on Artificial Intelligence*, Orlando, FL, July 1999; 291–298.
29. Strassner J. *Policy-based Network Management: Solutions for the Next Generation*. Morgan Kaufmann: San Francisco, CA, 2004.
30. Agrawal D, Calo S, Giles J, Lee K-W, Verma D. Policy management for networked systems and applications. *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management*, Nice, France, May 2005. IEEE Computer Society Press: Los Alamitos, CA, 2005; 455–468.
31. Minsky NH, Pal P. Law-governed regularities in object systems—Part 2. A Concrete Implementation. *Theory and Practice of Object Systems* 1997; **3**(2):87–101.
32. Uszok A, Bradshaw J, Jeffers R, Suri N, Hayes P, Breedy M, Bunch L, Johnson M, Kulkarni S, Lott J. KAoS policy and domain services: Toward a description-logic approach to policy representation. Deconfliction and enforcement. *Proceedings of the 4th IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2003)*. Lake Como, Italy, June 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 93–96.
33. Kagal L. Rei: A policy language for the Me-Centric project. *Technical Report HPL-2002-270*, HP Laboratories, Palo Alto, CA, 2002; 1–22.
34. DAML. The DARPA Agent Markup Language. http://www.daml.org [30 May 2006].
35. OASIS. XACML 2.0 Core: eXtensible access control markup language (XACML), Version 2.0. http://www.oasis.org [30 May 2006].
36. Bandara A, Lupu E, Russo A, Dulay N, Sloman M, Flegas P, Charalambides M, Pavlou G. Policy refinement for DiffServ quality of service management. *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management (IM 2005)*, Nice, France, May 2005; 469–482.
37. Asmare E, Dulay N, Kim H, Lupu E, Sloman M. Management architecture and mission specification for unmanned autonomous vehicles. *Proceedings of the 2006 Conference on Systems Engineering for Autonomous Systems Defence Technology Centre*, Edinburgh, U.K., July 2006.
38. EU Project TrustCoM, IST-2002-2.3.1.9-1945. http://www.eu-trustcom.com [30 May 2006].
39. SAPHE: Smart and Aware Pervasive Healthcare Environment. http://ubimon.doc.ic.ac.uk/saphe/index.php?m=338 [6 November 2006].
40. Keoh S-L, Lupu E, Sloman M. PEACE: A policy-based establishment of ad-hoc communities. *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ, December 2004; 386–395.
41. Grandison T, Sloman M. Trust management tools for Internet applications. *Proceedings of the 1st International Conference on Trust Management* (*Lecture Notes in Computer Science*, vol. 2692). Springer: Berlin, 2003; 91–107.