An Event Service Supporting Autonomic Management of Ubiquitous Systems for e-Health

Stephen Strowes*, Nagwa Badr*, Naranker Dulay†, Steven Heeps*, Emil Lupu†, Morris Sloman† and Joe Sventek*
*Department of Computing Science, University of Glasgow
{sds,nagwa,heeps,joe}@dcs.gla.ac.uk
†Department of Computing, Imperial College London
{n.dulay,e.c.lupu,m.sloman}@doc.ic.ac.uk
http://www.dcs.gla.ac.uk/amuse/

Abstract— **An event system suitable for very simple devices corresponding to a body area network for monitoring patients is presented. Event systems can be used both for self-management of the components as well as indicating alarms relating to patient health state. Traditional event systems emphasise scalability and complex event dissemination for internet based systems, whereas we are considering ubiquitous systems with wireless communication and mobile nodes which may join or leave the system over time intervals of minutes. Issues such as persistent delivery are also important. We describe the design, prototype implementation, and performance characteristics of an event system architecture targeted at this application domain.**

I. Introduction

Monitoring chronically ill patients as they go about their normal activity enables early release from hospitals and improves the patients' quality of life. Analysis and data mining of the monitored information can be used to predict potential problems (such as a possible heart attack for a specific patient being monitored) and to generate a warning to the patient or medical staff; the information can also be used by medical researchers to understand body changes that take place prior to a specific problem. On-body and environmental sensors may also be used in the home for monitoring elderly patients to determine problem situations or deterioration of well-being over time [1]. However, configuration of the multiple sensors and software components that form an adaptive body-area network or a home monitoring network is not currently feasible for non-technical patients or medical staff.

Existing network and systems management frameworks do not cater for ubiquitous environments, although specific techniques for monitoring and event correlation, service discovery, quality of service and policy-based management can be used to some degree. Current frameworks are aimed at large-scale corporate environments, telecommunications networks and internet service providers. For self-management in ubiquitous systems to become a reality, it is necessary to define and implement architectures which can scale down to small lightweight structures with local decision making capabilities. The management functionality must be automatically integrated and adapted to the specific application requirements without human intervention. Autonomous, self-managed cells must be composable to form larger cells but also need to collaborate and integrate with each other in peer-to-peer relationships as well as across multiple levels of abstraction relating to hierarchical service relationships.

We are developing autonomic management techniques for self-configuring and self-managing such systems [2]. Such an autonomic monitoring system is termed a self-managed cell (SMC). At the heart of an SMC is an event bus, over which all management communication between devices or services is carried. The bus can also be used to carry application events relating to alarms indicating sensor thresholds have been exceeded e.g. for heart rate, blood pressure, blood oxygen level, body temperature. Actuator devices such as heart defibrillators, insulin and other drug pumps are being developed that could be triggered by these events. However, we do not consider that all communication within an SMC is routed via the event bus. We assume there may be remote invocations or monitored data, such as from a heart ECG monitor that could be sent to a remote station for viewing and analysis.

The SMC also has to cater for a variety of different communication protocols such as WiFi and ZigBee, as well as GPRS or 3G for remote reporting.

In this paper, we present relevant background and related work (Section II), outline the requirements and architecture of the event bus for an SMC (Section III), describe a prototype implementation and initial performance results (Sections IV and V), and discuss intended future work (Section VI).

II. Background and Related Work

As with most management systems, an SMC must concern itself with the five traditional aspects of system management (fault, configuration, accounting, performance, and security) to varying degrees [3]. Figure 1 shows that an SMC constructed as a body-area network consists of a collection of wireless sensors. These sensors can both send and receive data. Each sensor can also receive control commands from management components, such as the Policy Service, to change thresholds or monitoring strategy. Many management systems perform control actions as a result of receiving events that an error threshold has been exceeded, a new service has been requested or a component has failed.

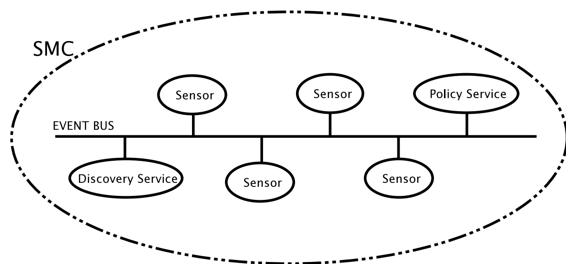


Fig. 1. High level view of an SMC).

Given the wireless nature of the networks to be constructed and the potentially sensitive nature of the data to be carried by them, mechanisms must be employed to guarantee delivery of events between components.

The core of an SMC consists of three components largely independent of each other, each fulfilling part of the functionality required. These, the policy service, the discovery service and the event bus, are discussed below.

A. Policy-based Management

Policies provide the means of specifying the adaptation strategy for autonomic management [4]. Authorisation policies specify what resources the components assigned to a role can access, and obligation policies (event-condition-action rules) specify how components/services react to events and interact with other components/services. When a device is discovered and granted membership of an SMC, the appropriate policies, based on device type, are deployed to it. This is triggered by a discovery event. Policies can be added, removed, enabled and disabled to change the behaviour of cell components without reprogramming them. Policies also govern the behaviour of the discovery service and the policy service itself, enabling these to be tailored to specific situations.

B. Device Discovery

An SMC includes a discovery service, which implements a protocol to search for new devices to integrate into the cell, and maintains connectivity to those devices while they are within range. The discovery service is responsible for managing group membership. It handles the detection and admission of new services to the SMC when they enter communication range (employing authentication specific to the application) and the removal of services which have left the SMC (through being physically removed or battery failure). The protocol is designed to mask transient disconnections between components, e.g. a nurse leaves the room for a short period of time before returning.

The discovery protocol does not use the event bus for monitoring group membership. Instead, the discovery protocol works with the event bus to separate the concern of group membership from the concern of passing events between services. However, the discovery service informs the SMC of the arrival or departure of devices via “New Member” and “Purge Member” events, respectively.

C. Event Bus, Behaviour and Semantics

The event bus is required to forward events from services in an SMC onto any interested parties within the SMC which have subscribed to receive the event. Other services, such as the discovery service, use the event bus to generate management or application specific events, but as stated previously, not all communication is via the event bus.

It is essential that the communication of management events satisfy *at most once* semantics - i.e. all events are delivered to each interested component exactly once as long as the component remains a member of the SMC. Since there may be causal relationships between pairs of events from the same sending component, the event bus must also guarantee that all events from a particular sender are delivered to each interested receiver in the order sent. Note that this does not say anything about delivery order between events from *different* sending components, as this would require a model of causality for the entire SMC.

It is not expected that the event bus will have to deal with high volumes of events since it is devoted to management traffic related to a small set of sensors over a patient’s body. Indeed, given that the target platform for the event bus is to be a PDA or similar device, we have to constrain the memory footprint and computational load that the event bus requires.

D. Related Work

Many existing content-based publish/subscribe systems are designed to allow the service to scale to many more subscribers than we need within an SMC, often employing some sort of method of distributing servers to spread the expected workload (for example, Elvin [5], Siena [6], JMS [7]). Of these, some have processing requirements too high for the intended platform of a PDA (JMS, which requires J2EE), and some carry potential licensing issues in the future (Elvin is being marketed as a saleable product by Mantara Software). Testing has confirmed that the Siena codebase is capable of being compiled to run under both a restricted J2ME CDC Personal Profile virtual machine, as well as Blackdown’s JVM version 1.3.1, which has been ported to run on various iPAQ models under Familiar Linux. Some systems, such as iBus//Mobile [8] and Elvin, are designed to offer delivery of events to mobile agents, but don’t offer an event forwarding service on mobile devices. None of the systems support the required delivery semantics outlined in Section II-C.

In light of these restrictions, Siena was chosen as the pub/sub mechanism for rapid prototyping of an event bus which provided the required semantics. Siena is open source, and so is easy to modify. This allowed us to build an event bus quickly. We have since followed this with the development of a dedicated pub/sub mechanism written in C to replace Siena.

III. Event Bus Architecture

The event bus relies on a number of distinct software components to offer the functionality we require: a content-based publish/subscribe mechanism to match events against subscriptions; proxy objects to mask heterogeneity of the

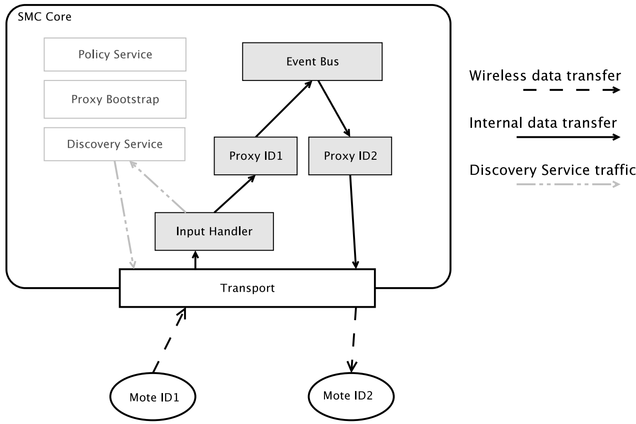


Fig. 2. High level view of objects within an SMC, and illustration of event flow through the core (minus acknowledgement packets).

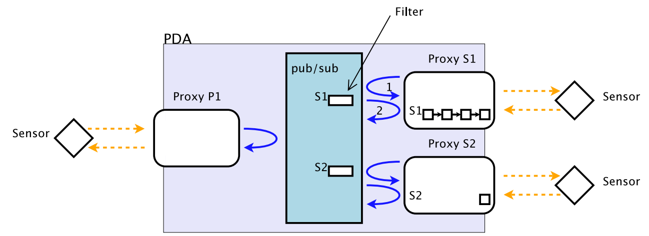


Fig. 3. High-level block diagram of basic interactions between components outwith the event bus, and the event bus itself; solid-lined arrows here indicate synchronous procedure-call semantics, and the direction of the 'jump' of that call; dashed-line arrows indicate asynchronous calls.

various transport mechanisms used by both publishers and subscribers; a bootstrap mechanism to instantiate the proxies as required; and a generic transport layer to carry packets across the network transport being used. By masking network details within the transport layer, we retain flexibility across different types of network transports while testing. In this section we provide an overview of these components. The relationships between these components is shown in Figure 2, with an example of the movement of an event between sensors.

A. The Publish/Subscribe Mechanism

In order to produce a system quickly for experimentation, we used Siena with an appropriate interface to allow translation of Siena subscription/notification types to or from our own. Additional code, in the form of proxies, surrounding the publish/subscribe mechanism is then responsible for both providing the semantics we require of the event bus as described in Section II-C), and for presenting an interface to the event bus for external services to use. Thus, we do not expect our semantics to be implemented by the basic publish/subscribe mechanism itself, and so present a clear separation of concerns.

The "EventBus" interface which we place around the publish/subscribe mechanism has allowed us to replace Siena with a more lightweight mechanism, thus reducing dependencies on other codebases given the unique nature of our target application.

B. Proxies

To ease the development of the architecture for various types of service or device, and to provide the data delivery semantics we require, each core component of the SMC (including the event bus) communicates with member services via a dedicated proxy. The proxy deals with data translation to and from a format the device will understand, and also ensures delivery of events and subscriptions in both directions. Each service granted membership of the SMC is represented by a proxy object, which provides a standard interface to that service.

On receiving notification of a new member from the discovery service, the event bus creates a proxy for that new member. This mechanism is covered in Section III-C. The proxy immediately subscribes to incoming "Purge Member" events generated by the discovery service, such that it can destroy itself, and any outbound data awaiting delivery, when required (i.e. when the service permanently leaves the SMC).

A proxy is modelled as an abstract class containing generic code applicable to all SMC services, completed by a concrete class containing implementation details specific to the device/service type. With this design, we can build complex proxies for simple sensors (capable of performing translation between the device protocol and higher level event types) or simple proxies for complex sensors (resembling a mere forwarding mechanism between the services). Note that while Figure 2 indicates the use of one transport layer, a proxy would be able to generate its own transport layer to facilitate communication over a different network transport; for example, a proxy might be kept in place to facilitate communication with a diagnostic device, connected to the SMC via an Ethernet connection.

All calls between services in this model are synchronous; events are always acknowledged when passing from publisher (sensor) to event bus, and from the event bus to each subscriber, so that events cannot be lost in transit. This model is shown in Fig. 3. While the core event bus semantics require that the event bus acknowledge receipt of events and subscriptions, it is the design choice of the proxy as to whether it should forward this acknowledgement to the device itself (for example, a temperature sensor may periodically transmit data and not require any acknowledgement prior to the next reading).

Subscribers register to receive events which match the content descriptions to which they have subscribed (Fig. 3, Arrow 1). In the case of simpler devices, the proxy itself might carry enough knowledge to register for appropriate events on behalf of the device upon its creation, after the device is granted SMC membership. Otherwise the device/service might register itself via its proxy.

As part of the subscription process, a filter is placed in the publish/subscribe server, representing this subscription, and

the ID of the proxy registered. This information is used first to determine whether an event is applicable to a given subscriber, and to subsequently push matching events to the subscriber (Fig. 3, Arrow 2).

Outgoing events are delivered to a subscriber's proxy by the event bus, where they are queued prior to processing and sending to the device, thus maintaining the ordering constraint on events. This also allows for events unacknowledged by the device to be resent by the proxy.

Incoming data from devices are also sent to the proxy, to perform pre-processing of that data into fully fledged data objects before forwarding to other internal services (for example, the temperature sensor mentioned above may periodically send a series of bytes representing a temperature reading, which the proxy converts into an object representing an event carrying that temperature, which can be sent to the event bus to be forwarded to recipients).

The publisher of an event notifies the core of the SMC of a new event without any knowledge of the number of subscribers listening for that event or the existence of proxies themselves. The proxy then deals with internal communication of the event to the event bus, acknowledging to the publisher events which have been accepted.

C. Proxy Bootstrap Mechanism

By specifying that all communication between the event bus and the SMC services takes place via a proxy, there must be a mechanism for creating a proxy when a new service joins the SMC.

The most straightforward method of achieving this is to register a service responsible for the creation of proxies with the publish/subscribe server which will react to "New Member" events generated by the discovery service; these events must carry enough information for the proxy-creation process to be able to generate the appropriate proxy type for the new service. The bootstrap mechanism must therefore be initialised on the creation of the event bus.

D. Transport Layer

Components within the core of the SMC use a generic transport layer to communicate with each other, which decouples higher level components from the actual network layer beneath. This is modelled as an abstract class (forming the generic interface required), extended to complete the details of the actual network transport to be used.

This transport layer presents `recv()` and `send()` calls to objects which make use of it. Respectively, the layer returns and accepts arrays of bytes, which can be bundled into further packet structures if required for transmission across the underlying transport mechanism. Much of the complexity of the underlying transport can be hidden within the constructor of a concrete transport class.

The choice of using byte arrays as input and output of the transport layer not only simplifies the functionality of the layer, but avoids unnecessary class hierarchies in other parts of the codebase. Further, handling data transfer in this manner removes the reliance on Java's serialisation process, allowing for coding SMC services external to the core of the SMC (e.g. sensors), in languages other than Java. Thus, we do not enforce the use of Java as part of the SMC, and do not preclude the possibility of the core SMC services being rewritten in C or C++ in the future.

IV. Prototype Implementation

Initial work involved building the architecture we describe in Section III with Siena performing matching of events to subscriptions. We have since replaced this mechanism with our own C-based matching mechanism, interfacing with the existing Java codebase via JNI [9]. Our own matching mechanism is based on the basic Siena fast forwarding algorithm [10].

Initial development has taken place largely on desktop systems sharing the same local area network, passing UDP datagram packets between machines. This allows for packets to be sent between hosts without the need to set up TCP connections and without guarantee of delivery, and so can be seen to mimic the wireless environment over which our SMC will run.

The current prototype uses a transport layer which makes use of datagram sockets. Sockets are opened within the "Transport" constructor, and subsequent `send()` and `recv()` calls are wrappers around send and receive calls over these sockets.

In this prototype, a 48 bit ID for each service is generated from the transport layer's unicast socket and the port number that socket is attached to – by simply opening a socket and not binding to a specific port, the operating system is free to choose the port number for the socket request, and so the prototype is not hardwired to use a specific port for unicast traffic. Broadcast traffic, generated by the discovery service, is delivered on an arbitrarily chosen port number known by services, to allow new services to listen for nearby discovery services.

This development environment has been migrated to an iPAQ hx4700 PDA running Familiar Linux with Blackdown Java 1.3.1, communicating with a laptop (1.2GHz Pentium 3 with 256MB RAM) via an IP connection over a USB cable; this allows for the same UDP Transport layer to be used in testing the suitability of the software for a more restricted environment.

Support for 802.11b under Linux on this PDA is not yet available, so development is progressing on a wireless implementation using the built-in Bluetooth [11] capabilities of the device; Bluetooth dongles will allow the use of other devices, and allow testing of devices moving in and out of range of the SMC. The testing of these environments should allow for an easy migration to Zigbee [12] hardware in the future.

Currently, prototype versions of the event bus, discovery service, and policy service have all been trialled largely independently of each other. Work is underway to integrate the various core components of the SMC to enable further development, testing, and experimentation. Testing of the

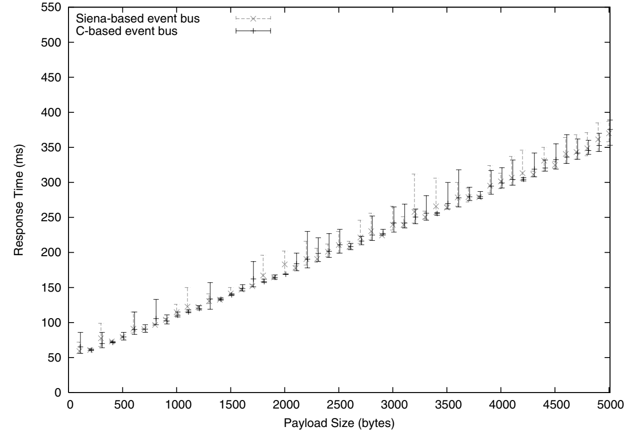
proxy architecture has consisted of building test sensors in Java and C, allowing the proxies to translate/acknowledge data as required.

V. Initial Performance Results

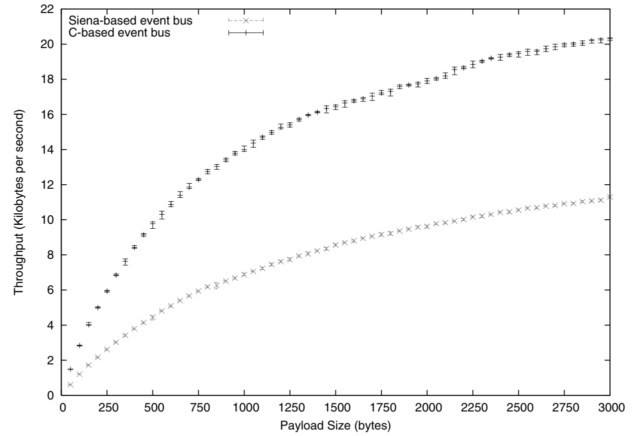
The performance of the event bus is key to the success of the SMC architecture, given the constrained environment in which it is intended to run. Using the testing environment of the PDA and the laptop as described in Section IV, and the two event buses we have developed, we tested the elapsed response time of the event bus against message size (Figure 4(a)) and the throughput of the event bus against message size (Figure 4(b)).

The response time of the event bus is dependent on the latency of the link, scheduling decisions made by the Linux kernel at both ends of the link, the time taken to transfer data on a socket to the JVM, and the behaviour of the JVM itself. The latency on the link is 1.5ms on average (0.6ms minimum, 2.3ms maximum taken over the link for 1 minute), so most of the latency observed in Figure 4(a) is dependent on the behaviour of the operating system at each host, and also of the JVM at each host. The average rise in response time over the course of the experiment is generated by copying of packet data, which we have attempted to minimise in the C-based publish/subscribe mechanism.

As mentioned in Section II-C, the event bus is targeted solely at the needs of management control and limited application event traffic; we do not expect the throughput requirements for such traffic to be onerous. Considering that the link can sustain a throughput of approximately 575KB/s when simply transferring data from one host to another, the results in Figure 4(b) indicate a considerably lower data throughput via both our event buses. However, by building a minimal publish/subscribe matching mechanism in C designed for our system, we do improve the throughput which the architecture can sustain. This gain in throughput may be attributed to the much simpler codebase not requiring the same data translations Siena required, including translation to or from our own data types. Obviously, the payload throughput we observe does not take into account application headers, datagram packet headers, the overhead of dealing with each packet through the OS to the JVM and back, and copying data, and hence some loss of data throughput can be attributed to these factors. There is certainly scope for improving the performance and throughput this software can provide as development continues.



(a) Variation in end-to-end delay against data sizes.



(b) Variation in throughput against data sizes.

Fig. 4. Observed behaviour of the event bus running on the PDA at varying data sizes.

VI. Future Work

The wireless nature of the devices we expect an SMC to use in an e-Health environment are driving development toward wireless technologies. Currently, we are developing a prototype using Bluetooth. Soon, we will test the SMC architecture using devices which communicate via the ZigBee wireless protocol, using a number of scenarios to test various aspects of the system (such as maximum timeouts for the discovery service to allow silence from a device until a "Purge Member" event is launched). In a similar vein, we will explore the mechanism for queueing and repeating attempts to deliver events to services which are unavailable, but have not yet been declared to have left the SMC. Further investigation into event bus performance (variation in delays incurred depending on message size or number of recipients, for example), and possible improvements will also be investigated.

Further, it is possible that we would see power-saving benefits from quenching techniques such as those demonstrated in the Elvin publish/subscribe system. We also intend to replace the content-based publish/subscribe mechanism with a type-based publish/subscribe [13] mechanism, to remove the reliance on arbitrary tags as event identifiers.

Development of the existing event bus/discovery service/policy service architecture will continue, while also expanding our array of useful test scenarios to help verify the validity of the system.

We also intend to look into using the JamVM virtual machine with the output of the GNU Classpath project to minimise the footprint of the Java virtual machine. We will consider the performance of this JVM against the Blackdown

JVM.

Acknowledgement

The authors wish to thank the UK Engineering and Physical Sciences Research Council for their support of this research through grants GR/S68040/01 and GR/S68033/01.

References

[1] Care in the Community, A virtual research centre of the DTI Next Wave Technologies and Markets Programme, http://www.dticareinthecommunity.com/, accessed 6 March, 2006.

[2] J. S. Sventek, N. Badr, N. Dulay, S. Heeps, E. Lupu, and M. Sloman, "Self-managed cells and their federation." in *CAiSE Workshops (2)*, 2005, pp. 97–107.

[3] M. Sloman, Ed., *Network and Distributed Systems Management*. Addison Wesley, May 1994, ISBN: 0201627450.

[4] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*. London, UK: Springer-Verlag, 2001, pp. 18–38.

[5] G. Fitzpatrick, T. Mansfield, S. Kaplan, D. Arnold, T. Phelps, and B. Segall, "Augmenting the workaday world with elvin," in *Proceedings of the Sixth European conference on Computer supported cooperative work*. Norwell, MA, USA: Kluwer Academic Publishers, 1999, pp. 431–450.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, Aug. 2001. [Online]. Available: http://serl.cs.colorado.edu/~carzanig/papers/

[7] S. Grant, M. P. Kovacs, M. Kunnumpurath, S. Maffeis, K. S. Morrison, G. S. Raj, and J. McGovern, *Professional JMS*, 1st ed., P. Giotta, Ed. Wrox Press, March 2001, ISBN: 1861004931.

[8] Softwired, "iBus//Mobile homepage," http://www.softwired-inc.com/products/mobile/mobile.html, accessed 17 January, 2006.

[9] S. Liang, *The Java Native Interface: Programming Guide and Reference*. Addison Wesley, July 1999, iSBN: 0201325772.

[10] A. Carzaniga and A. L. Wolf, "Forwarding in a content-based network," in *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, Aug. 2003, pp. 163–174.

[11] Bluetooth SIG, Inc., "The official bluetooth membership site," https://www.bluetooth.org/, accessed 20 January 2006.

[12] "Zigbee alliance," http://www.zigbee.org/, accessed 20 January 2006.

[13] P. Eugster, R. Guerraoui, and J. Sventek, "Type-Based Publish/Subscribe," Swiss Federal Institute of Technology, Lausanne (EPFL), Tech. Rep., 2000.