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*— Healthcare and economic drivers have increased the desire to deploy systems for continuous monitoring of patients, both in hospital and outpatient settings. Such systems must operate autonomically, and must meet a number of operational constraints. Event systems used in traditional monitoring systems have not been designed with these constraints in mind. We describe the design, implementation, and performance characteristics of an event system 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 [10]. 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. Their architecture is based on functional decomposition where the various functions are integrated in centralised network operations centres by human administrators. 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 [1]. The systems must be able to add or remove components, cater for failed components and error-prone sensors, automatically detect and adapt to a user's current activity and communication capability as well as catering for interaction with health visitors or other medical staff who attend patients or visit elderly people. 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. 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 forms of system management (fault, configuration, accounting, performance, and security) to varying degrees [2]. Figure 2 shows that an SMC constructed as a body-area network consists of a collection of wireless sensors, which can both send and receive data; each sensor can also receive control commands from management components, such as the Policy Service. Most management systems utilise an event bus/service to convey control traffic between the components of the system, primarily due to the need for run-time extensibility of the component topology [2]. Application-specific traffic between components is expected to be carried using other communication paradigms; the event bus can be used to convey application data between components, but only if the reliability and delay semantics of the bus match the data delivery semantics required by the components.

Given the wireless nature of the networks to be constructed and the potentially sensitive nature of the data to be dealt

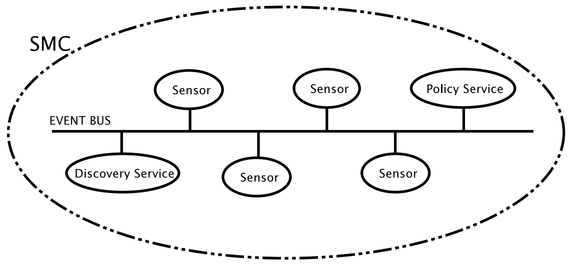


Fig. 1. High level view of an SMC.

with, mechanisms must be employed to guarantee delivery of management messages 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 event bus, the discovery service and the policy service, are discussed in this section.

A. Policy-based Management

Policies provide the means of specifying the adaptation strategy for autonomic management [3]. 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 granted membership of an SMC, the appropriate polices are deployed to it. 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 maintain 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.

The discovery protocol does not use the event bus for monitoring group membership. Instead, the discovery protocol works outwith the event service 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 services via “New Member” and “Purge Member” events, respectively.

C. Event Bus Behaviour and Semantics

The event bus is required to forward event notifications from services in an SMC onto any interested parties within the SMC, with other critical functions, such as device discovery, fulfilled by different services; these services may either use the event bus to communicate with each other, or another communication paradigm if more appropriate.

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 that component is still 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; indeed, if the target platform type for the event bus is to be a PDA, we have to constrain the memory footprint and computational load required for the event bus. Thus, the event bus must be lightweight, but not so lightweight that this conflicts with required functionality.

D. Related Work

Within the constrained environment of a PDA, care needs to be taken when choosing an existing publish/subscribe technology to use. Many such systems were designed for highly scalable, high performance applications, and are neither suited to our application type nor available processing power.

A number of content-based publish/subscribe services for the routing functionality of our event bus were considered. Many existing publish/subscribe projects 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 [4], Siena [5], JMS [6]). Of these, some have running 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). Another system, iBus//Mobile [7], aims to target delivery of events to mobile agents, but does not offer the event forwarding service on mobile devices.

In light of these restrictions, Siena has been chosen as the publish/subscribe technology used within our event bus, at least for the purposes of prototyping. Testing has confirmed that the Siena codebase is capable of being compiled to run under both a restricted J2ME CDC Personal Profile virtual machine, and also Blackdown’s JVM version 1.3.1, which has been ported to run on various iPAQ models under Familiar Linux.

III. Event Bus Architecture

The event bus relies on a number of distinct software components to offer the functionality we require: an interface to the Siena codebase; proxy objects to services; a bootstrap mechanism to build proxies for the event bus to use; and a

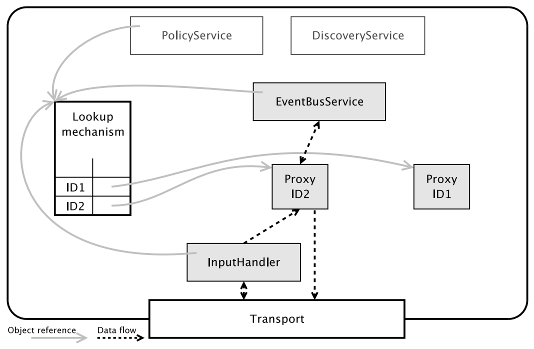


Fig. 2. High level view of objects within an SMC.

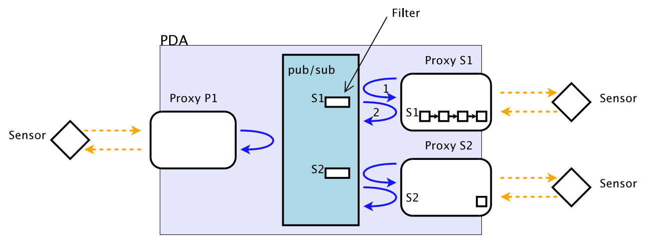


Fig. 3. High-level block diagram of basic interactions between components outwith the event bus, and the event bus itself; solid-lined arrows here indicate synchronous procedure-call semantics, and the direction of the 'jump' of that call; dashed-line arrows indicate asynchronous calls.

generic transport layer to carry packets. In our design we also 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.

A. *The Publish/Subscribe Server*

For the purpose of prototyping the event bus we chose to speed up development by embedding Siena code within our codebase, as discussed in Section II-D. Additional code surrounding the publish/subscribe mechanism is then responsible for providing both the semantics we require of the event bus as described in Section II-C and also to provide an appropriate interface to the functionality it will offer.

Beyond prototyping, we intend to replace the Siena codebase with custom code built for purpose, in a bid to minimise the resources required to run the event bus, and to reduce dependencies on other codebases given the unique nature of our target application.

B. *Proxies*

To ease the development of the architecture to varying types of services or devices, and to assist in embedding Siena code, the core components of the SMC (including the event bus) communicate to services via that service's own proxy.

All entities granted membership of the SMC are represented by a proxy object, which provides a standard interface to that entity. This proxy is responsible for dealing with data exchange between SMC members, and also for providing the guaranteed delivery semantics we require for data transfer.

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 being communicated with. 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 in place to facilitate communication with a diagnostic device, connected to the SMC via an ethernet connection.

On receiving a 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 when required (ie: when the service is removed from the SMC).

Proxies deal with outbound data queues, hence providing the guarantee of data delivery we require; thus, we do not expect this functionality of the publish/subscribe mechanism itself. Clearly if data is queued at a proxy, that proxy can destroy itself and remove any waiting data on receipt of a "Purge Member" event. Incoming data from services into the SMC are also sent to the proxy, to perform pre-processing of that data into fully fledged objects before forwarding to other internal services (for example, constructing an event to be sent to the event bus from a series of bytes representing the output from a temperature sensor).

All calls between services in this model are synchronous; event notifications are always acknowledged when passing from publisher to router, and from router to each subscriber, so that events are not lost in transit. This model is shown in Fig. 3.

The publisher of an event notifies its proxy of a new event without any knowledge of the number of subscribers listening for that event. The proxy deals with internal communication of the event to the event bus, acknowledging to the service events which have been accepted. While calls are synchronous, simpler devices might transmit data to proxy without requiring any return information of the proxy (for example, a temperature monitor which regularly generates events).

Subscribers register to receive notifications of the types in which they are interested (Fig. 3, Arrow 1). In the case of more simple devices, the proxy itself might carry enough knowledge to register for appropriate events on behalf of the device upon its creation when the device is granted SMC membership; otherwise the device/service might register by itself via its proxy.

On subscribing, 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

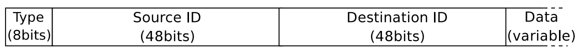


Fig. 4. Transport layer packet format.

a notification is applicable to a given subscriber, and to subsequently push matching notifications to a subscriber (Fig. 3, Arrow 2).

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 SMC use a generic transport layer to communicate with each other, which de-couples higher level components from the actual network layer beneath. This is modelled as an abstract class (forming the generic interface required), extended by concrete classes carrying the details of the actual network transport to be used.

This transport layer presents two calls to objects which make use of it: `recv()` and `send()`. Respectively, the layer returns and accepts bytes arrays, 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 development has taken place largely on desktop systems sharing the same local area network, passing datagram packets between machines. This allows for packets to be sent from host to host 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 to mimic connectionless transmission over a wireless medium. Sockets are opened within the Transport constructor, and subsequent send and recv calls are wrappers around send and receive calls over these sockets.

Packets tend to be of the format outlined in Figure 4, and packets of this form are defined which carry events or subscription requests. Event and subscription objects can be reduced to byte arrays for inclusion into these packets.

In this prototype, the 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 UDP packets to be transmitted between machines, for testing of 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 [8] 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 [9] 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.

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, we tested the elapsed response time of the event bus against message size (Figure 5(a)) and the throughput of the event bus against message size (Figure 5(b)).

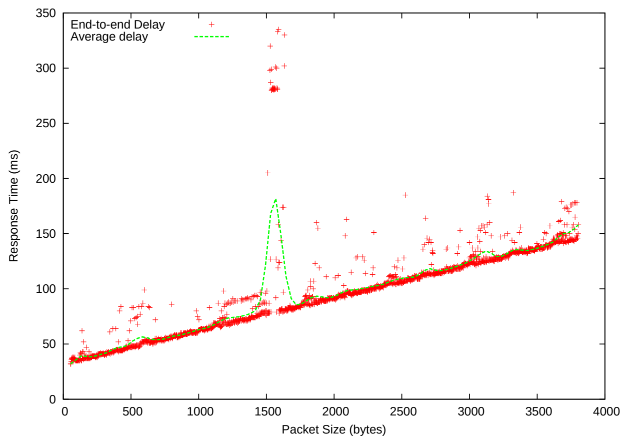
The response time of the event bus is dependent on the latency of the link, scheduling decisions made by the linux kernel at both ends of the link, the time taken to transfer data on a socket to the JVM, and the behaviour of the JVM itself. The latency on the link is 1.5ms (on average, 0.6ms minimum, 2.3ms maximum taken over the link for 1 minute), so most of the latency observed in Figure 5(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 unnecessary copying of packet data, which we will eliminate in future versions of the software.

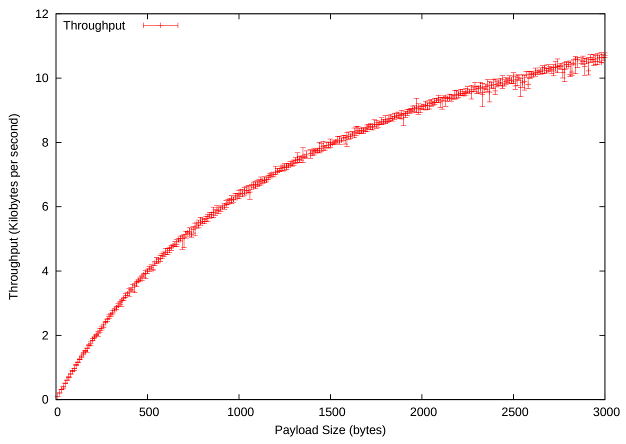
The spike in the results at packet sizes around 1500 bytes in this graph deserves explanation, as it was evident in all tests: it appears that the garbage collector on the PDA's JVM runs at this point during all tests, and attempts to influence the behaviour of the collector by using `System.gc()` at key points in the code failed to force the JVM scheduler to run the collector sooner. No events were lost, there was simply an additional latency on those packets. Running the test against just those packet sizes confirms that, under normal conditions, they achieve a latency similar to surrounding packet sizes.

As indicated earlier, the event bus is targeted solely at the needs for management control traffic; we do not expect the throughput requirements for such traffic to be onerous, but if applications decide to use the event bus for communicating data traffic, throughput considerations will become more important. The throughput results in Figure 5(b) are small, and the figures shown relate only to the volume of data encapsulated within an event (the payload). Thus, the actual throughput of data 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, copying data, and translating event types to and from a form which the Siena code can handle. The raw throughput observed on the link when simply transferring data from one host to another is approximately 575KB/s. There is clearly significant scope for improving the performance and throughput this software can provide as development continues.

During these tests, the JVM consumed approximately 10% of the CPU load on the PDA for smaller payloads, and upward of 30% of the CPU load for larger payloads.



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



(b) Variation in throughput against packet sizes.

Fig. 5. Observed behaviour of the event bus running on the PDA at varying packet 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 [10] 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 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. To improve performance, we are considering a reimplementation of the project in C or C++ beneficial, though a hybrid approach using C/C++ components interfacing with the existing Java code via JNI [11] might prove enough to enhance performance.

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] J. Sventek, N. Badr, N. Dulay, S. Heeps, E. Lupu, and M. Sloman, "Self-Managed Cells and their Federation," in *Workshop Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*.

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

[3] 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.

[4] 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.

[5] 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/

[6] 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.

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

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

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

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

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