APM

POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE CB3 0RD UNITED KINGDOM
+44 1223 515010 • Fax +44 1223 359779 • Email: apm@ansa.co.uk • URL: http://www.ansa.co.uk

Object Lab

Object Monitor

Richard Hayton & Scarlet Schwiderski

Abstract
The benefits of event monitoring systems have been recognised for a wide range of applications: centralised and distributed applications, running on different computing platforms, in different execution environments. CORBA's and Microsoft COM's event services, EventKona, and Java Beans events all support certain features of event monitoring systems, but no general architecture has been designed. The goal of the **Object Monitor** project is to define a general event monitoring architecture based on the Java Beans approach for events.

1938 **Approved** 21 January 1997

Distribution:
Supersedes:
Superseded by:

Copyright © 1996 APM Limited

TABLE OF CONTENTS

1 INTRODUCTION **1**

2 GENERALISED EVENT MODEL **2**

2 .1 Registration and Notification 2

2 .2 Example 2

2 .3 Event Classification 3

2 .4 Registration Formats 3

2 .5 Optimisation of Heterogeneous Registration Formats 4

3 THE JAVA BEANS API **5**

3 .1 Events in the Java Beans Architecture 5

4 OBJECT MONITOR PROPOSAL **6**

5 SUMMARY **8**

6 REFERENCES **9**

1 INTRODUCTION

Many existing and emerging computing systems are active. Components of *active systems* communicate by signalling *events* and by examining the events generated by other components. Components at all levels of a distributed system may make use of event technology, from local software components (for example a GUI) to complex systems (for example an Airline Booking system). The alternative to using event technology in such systems is to use *polling*, i.e. to poll the state of certain objects in a database in regular intervals and monitor them in order to detect certain situations. However, polling does not scale, especially if events occur infrequently.

Abstracting "asynchronous notifications" as events can have many advantages for both the programmer and the end user. The notion of events in GUI design has been long accepted, but events are equally useful in any situation when an occurrence must be signalled to one or more interested parties. For example, in banking an event may be raised when an account's overdraft limit has been exceeded, in order to notify the customer. Or, in air traffic control an event may be raised when two planes get below a certain distance, in order to signal a potential security hazard. Further, on the lower/systems level a mobile application may be signalled events whenever it's network connectivity changes or battery power runs low. By abstracting "occurrences" as instances of events, we are able to re-use technology in order to deal with event multicast, failure recovery, archiving, security, priority and other non-functional issues.

From the user's point of view, the use of events can lead to more active and configurable applications. Typically, users may request to take some action when new mail arrives, or to be notified as soon as a colleague logged on to their computer. More complex requests can be constructed as a *composition* of basic events. For example "When Fred is not on the phone and Mary is back from lunch, set up a conference call between Fred, Mary, and me."

Recently, there has been an increased interest in active systems and in event notification in particular. Various event frameworks have been proposed, such as CORBA's event service [1], Microsoft's COM [2], EventKona [3], and Java Beans [4]. However, a general lack is that these models focus on particular application areas, and do not interwork well. For example, the CORBA model concentrates on the semantics of event streams and does not consider the issues of event classification. The Java Beans approach has strong event classification, but poor support for generic tools such as multiplexing or remote notification.

In the following section, we examine the basic concepts of event-based systems, and derive a generic framework to allow different implementations of event services to inter-work. In Section 3, the Java Beans event system is evaluated and some limitations are highlighted. In Section 4, we propose to implement a subset of the event framework described in Section 2 as an extension to the Java Beans model. This will allow programmers to use *generic adapters* to control the non-functional aspects of event notification. In particular, the distribution, performance and failure trade-offs may be tuned. Additionally, we propose to implement a general purpose *composite event recognisor* in order to aid application development, and in order to provide support for declarative composite event specification used in interactive applications.

2 GENERALISED EVENT MODEL

2.1 Registration and Notification

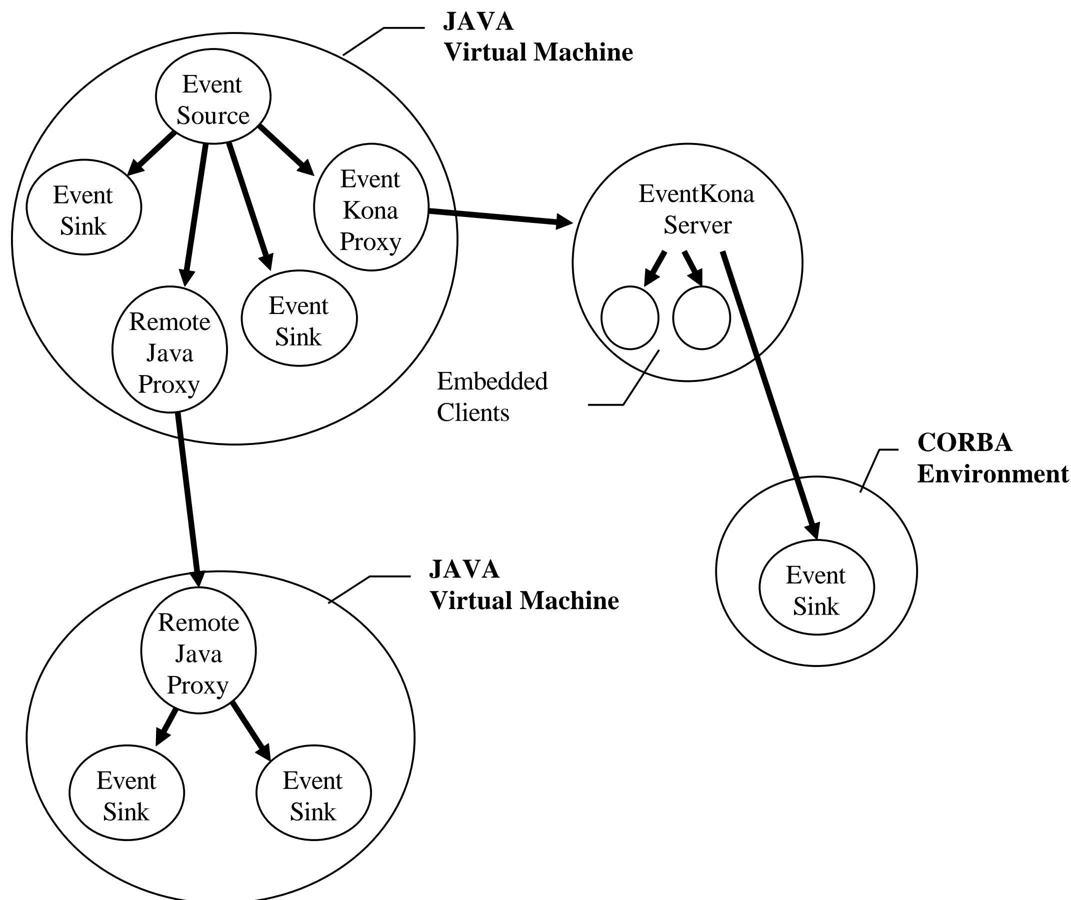
The basic concept behind event notification is that one component indicates when something (namely an event) has occurred, and that interested parties are made aware of it. The event may be generated *implicitly*, for example by invoking a method on an object, or *explicitly*, for example by raising an alarm. A naive approach to event evaluation is to enter all events in a central repository, and for interested parties to dredge this repository. This approach does not scale in general, although it may be appropriate in some circumstances.

The key point of our approach is that clients interested in events *register* that interest explicitly. This registration may be kept with the object responsible for generating the event, or with some other object that will itself be informed of the event. When an event occurs, the *event generator* has a list of all registered interested parties and notifies them.

We might want to make events available to a wider audience (i.e. in a distributed system) by creating a per-domain *event server*. An event server would accept remote requests and act as a proxy between the true event generator and the *event sink*. Clearly, there may be an arbitrary connection of objects capable of manipulating events. For example, a *composite event service* is a particular kind of proxy that looks for patterns of events rather than for individual events. A client may connect to a composite event server and request to be notified whenever an **A** event occurs followed by a **B** event. The composite event server would in turn register interest in events **A** and **B** and would "wait" for a suitable sequence to occur before signalling an event of its own. This approach is highly flexible, as it does not enforce a particular implementation.

2.2 Example

The following diagram gives an example of event propagation across a number of machines and event implementations. The event is generated by "Event Source" which understands a local Java protocol used to propagate the event to clients within the same virtual machine. One of these clients is a proxy for remote Java objects and another is a proxy for the EventKona event service. These proxies propagate the event to their clients on local or remote machines, where they are processed further.



2.3 Event Classification

In a large distributed system, there may be many components capable of generating events. In order for a client to be informed when specific events occur, we must be able to *classify* the events so that the client can describe what he is interested in. For this reason, events are of specific *event types* and are characterised by specific *event parameters*. For example, an event may have a type "account's overdraft limit exceeded" and be characterised by parameters such as "account name", "account number", and "amount".

As distributed systems are typically large and complex, it is unreasonable to enforce a single class hierarchy for all events. This would suggest a universal implementation, and would restrict the speed and ease of application development. Instead events should be classified within some *context*. Typically this context will be defined by the application, or by the object that generates the events. Further, the context may be defined as the scope of some event service, such as EventKona.

In order for a client to register interest in an event, the client must be aware of a service that understands the context in which the event is classified. This may be the object that generates the event (if there is only one such object), the event service for the domain in which the object resides, or some *value-adding service* such as an EventKona server.

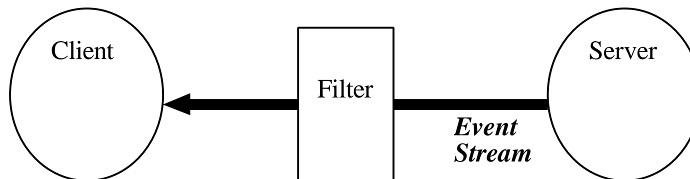
2.4 Registration Formats

When a client registers interest in events generated by an object or signalled by a service, it will wish to have some degree of control over which events it receives. Such a specification may take many different forms. For example, a client may be able to supply an arbitrary *acceptance expression* in terms of the event type and event parameters, which is evaluated for each event to determine if it is relevant. Alternatively, the acceptance expression may be more restricted, for example, allowing to specify the event type or a wildcard template only.

Whilst the former system has the benefit of expressive power, it also requires all acceptance expressions to be evaluated for each event. If only a restricted form of specification is allowed, the event generator will have to perform less processing, and may be able to use the same information to match multiple client requests. Taking an open approach, different objects or value-adding services will provide different flavours of registration.

2.5 Optimisation of Heterogeneous Registration Formats

Although it is necessary for different applications and services to choose different forms of acceptance expression, it is possible to hide much of this heterogeneity. Notionally, the client wishes to filter a stream of all events originating from the server.



In implementation terms, this is facilitated by providing an acceptance expression as a *pre-filter* and by constructing a *post-filter* to perform the rest of the filtering. For services that allow very general acceptance expressions, the pre-filter is large and the post-filter is small. For services with less general acceptance expressions, the reverse is true.

The form of acceptance expression used by the client must be chosen in such a way that it can be split into two parts, namely the pre- and post-filters. We propose that a wildcard template coupled with a general algebraic expression is a suitable choice for the majority of applications. This allows decomposition at the event class/template level, and in addition the algebraic expression can be split into 'hard' and 'easy' parts.

For example, a client interested in bogus **Withdrawal(account,amount,location)** events from their account might specify this as **Withdrawal(R.Hayton,x,l) : x>100 AND BadLocation(l)**. This can be split in the following ways:

Pre-Filter	**Post-Filter**
Withdrawal	Withdrawal(R.Hayton,x,l) : x>100 AND BadLocation(l)
Withdrawal(R.Hayton,x,l)	x>100 AND BadLocation(l)
Withdrawal(R.Hayton,x,l) : x>100	BadLocation(l)
Withdrawal(R.Hayton,x,l) : x>100 AND BadLocation(l) (download code for BadLocation)	*none*

3 THE JAVA BEANS API

The Java Beans API [4] defines a software component model for Java. A Java Bean is a reusable software component. Java Beans can be composed together into applications by end users. One feature of Java Beans is the support for events as a simple communication tool between Java Beans.

3.1 Events in the Java Beans Architecture

Java Beans supports the communication of events between an Event Source and one or more Event Listeners (that is, Event Sinks). An event is an object of a subclass of java.util.EventObject; that is, an event has a type and an arbitrary number of parameters. Event Listeners are strongly typed with respect to the event classes they may receive, and can register and de-register an interest in events of these classes dynamically by calling ADD and REMOVE methods at the corresponding Event Sources. Event Listeners implement actions for signalled event occurrences, depending on the event type.

Advantages:

- close to the proposed Event model
- support for typed events and event parameters
- dynamic registration of interest in events of specific event types
- the basic behaviour of event delivery can be enhanced by introducing *Event Adapters* (an Event Adapter class may be defined and interposed between an Event Source and an Event Listener)

Disadvantages:

- adapters are typed according to the event class, and cannot be reused for different event classes
- the approach is a programming convention, not a set of tools or abstractions
- only synchronous delivery of events to Event Listeners (does not scale)
- filtering events (i.e. by defining a registration format) not considered
- distribution aspects not considered, keeping in mind that naive distribution is expensive
- monitoring high-level composite events not considered

The last three points can, of course, be implemented explicitly by means of the provided programming tools, but there is no generic support.

4 OBJECT MONITOR PROPOSAL

We propose to build the Object Monitor Architecture on top of Java Beans events. In this way, we can exploit existing technology which is widely available. The idea is to extend the Java Beans approach with respect to the disadvantages listed above.

The following tasks have been identified:

- **Support for generic event adapters:**

  Special tools for event handling will support the end user in implementing an event monitoring system, therefore hiding unnecessary implementation detail. In the current Java Beans model, adapters are strongly typed and cannot be reused for different event classes. Many *non-functional* aspects of event management, such as filtering, buffering or remote distribution should be provided as generic, reusable components.

  Effort 1 pm

  Deliverable document

- **Generic support for distributed event monitoring:**

  Many applications are distributed, that is, Event Source and Event Listener reside on remote computing nodes. In this case, Remote Java Proxies (see Figure 1) have to be interposed at both sites, in order to support the event signalling and registration process. A naive implementation of distributed event notification would involve one RPC for each notification to each client. Clearly other approaches can be used, such as multicasting event instances or batch processing events. The correct approach will depend on the frequency of events, the number and distribution of Event Listeners and the importance of rapid delivery and preventing event loss. The intention is to provide an extensible set of protocols to allow programmers to make informed decisions about the correct trade-offs for their particular applications.

  Supporting distributed event monitoring requires the implementation of **asynchronous delivery:**

  By definition, an event is an “asynchronous notification”, a happening which is to be observed by one or more interested parties. Therefore, the asynchronous delivery of events is more natural than the synchronous delivery, which blocks the program execution at the event generating site until the Event Listener has reacted accordingly.

  Effort 2 pm

  Deliverable software

- **Generic support for event filtering:**

  Especially when looking at distributed systems, it is necessary to consider the efficiency of event signalling. Often an Event Listener is interested in events of a specific event type which fulfil certain constraints on their parameters. For example, a rainfall event may only be of interest if the amount exceeds a certain threshold. When registering an interest in events of a specific event type, the Event Listener must be able to specify such constraints with respect to the used registration format. Hence, irrelevant events do not cause an overhead on network traffic.

  Effort 2 pm

  Deliverable software

- **Generic support for high-level/composite events:**

Some events are high-level in nature, that is, map down to a number of low-level events. For example, an event “flood alert at Norfolk cost” may map down to different low-level “flood alert” events, relating to different weather stations at the Norfolk cost. Such *composite events* cannot be supported generically with the current approach. Supporting the specification and detection of composite events is a worthwhile task. In a distributed system, this task is complicated by the fact that low-level events are disseminated by different Event Sources located at different computing nodes.

Effort 2 pm

Deliverable software

Implementing the tasks in the given order will ensure a stepwise enhancement of the event monitoring system. The implementation of each task has to obey security and efficiency concerns.

5 SUMMARY

This document described the basic features of event monitoring systems and presented a proposal for a specific event monitoring system, the Object Lab's **Object Monitor**. This system is to be based on Java Beans events, which proposes a clean architecture for the generation, registration, and notification of events. The enhancements of Java Beans events for Object Monitor are concerned with the generic support of the generation, registration, and notification processes, with distributed event monitoring, event filtering, and the support of composite events. The document proposed a time frame for the realisation of the different tasks and defined the deliverables.

6 REFERENCES

[1] Object Management Group (Framingham, MA), *CORBA Services: Common Object Services Specification v.1.0*, March 1995

[2] Microsoft Corporation and Digital Equipment Corporation , *The Component Object Model Specification: Draft Version 0.9*, October 1995, ftp.omg.org:/pub/docs/1995/95-10-15.ps

[3] WebLogic, *Developers Guide for EventKona/T3*, http://www.weblogic.com/

[4] JavaSoft, *Java Beans*, http://java.sun.com/beans