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 Monitor

A Framework for Event-Monitoring Systems

David Franklin

Abstract

This report describes Object Monitor, an Object Lab deliverable that demonstrates distributed computing in action.

Object Monitor is an event monitoring framework which greatly facilitates the development of distributed event monitoring applications. It enables applications to register an interest in primitive events and also patterns of primitive events (composite events) via a simple and expressive event language. Applications can monitor many event patterns concurrently.

Object Monitor was designed with the aim of creating a flexible, open and efficient framework, enabling applications to directly use its functionality where appropriate but to be able to extend and override this where required. As such it is compatible with, but extends, the Java Beans event model and comprises a number of re-usable and extensible Java classes and interfaces.

2056.00.01	**Approved** **Technical Report**	08 October 1997

Distribution:
Supersedes:
Superseded by:

Table of Contents

1 Overview

This report describes Object Monitor, an Object Lab deliverable that demonstrates distributed computing in action.

1.1 What Is Object Monitor

Object Monitor is an event monitoring framework which greatly facilitates the development of distributed event monitoring applications. Its principal features are:

- It enables applications to monitor primitive events and also patterns of primitive events (also known as 'composite' events).
- It provides a simple and expressive composite event language which enables applications to express their interest in an event or in a specific combination of events.
- It enables applications to monitor a number of event patterns concurrently.
- It enables event sources and listeners to reside on different hosts.

Thus, distributed composite event monitoring applications can be easily developed with the Object Monitor framework.

1.2 Architectural Goals

Object Monitor was designed with the aim of creating a flexible, open and efficient framework, enabling applications to directly use its functionality where appropriate but to be able to extend and override this functionality where required.

As such it is compatible with, but extends, the Java Beans event model and comprises a number of re-usable and extensible Java classes and interfaces.

1.3 Structure of this Report

Section 2 introduces the basic concepts behind event-monitoring systems in general, and explores some of the issues that are introduced by distributed event-monitoring.

Section 3 describes Object Monitor's composite event language, giving some examples of its use. This language allows applications to register an interest in patterns of primitive events and is further explored in Section 4 when describing a demonstrator which shows how the Object Monitor framework can be used to monitor composite events within the network management domain.

Section 5 presents the event model of Object Monitor, showing how it relates to the Java Beans event model. Sections 6 and 7 describe the architecture of Object Monitor, explaining how it can be used as a framework to develop both centralised and distributed event monitoring systems by describing the implementation of the network management demonstrator in further detail.

In Section 8, Object Monitor is compared with some commercial event frameworks.

1.4 Deliverables

The Object Monitor deliverables comprises the following:

- This report.
- Java source and class files for the framework components, including the RMI transport implementation.
- Java source files for the single and multiple-JVM demonstration applications.
- Java documentation files.

2 Event Monitoring

2.1 Event-Driven Systems

Many existing and emerging systems handle asynchronous communications using events - they listen to events and they generate events. Components at all levels may exploit event technology, from simple GUI components through to complex distributed systems.

From the end user's perspective, the use of events can lead to more flexible applications. Some typical examples are:

- In banking, an event may be generated when an account's overdraft limit is exceeded, in order to notify the account holder.
- In air traffic control, an event may be generated when two planes get too close to one another, in order to signal a potential conflict that needs to be resolved.
- In the office, a rule-based event-driven system could be configured to perform some action when new mail arrives, or to set up a conference call as soon as the participants are back from lunch.

Event monitoring is crucial for many aspects of large, distributed systems - the alternative of polling simply does not scale. The major benefit of event-driven systems is loose coupling between event sources and event sinks. This enables event sources to be exploited in different applications without the event sources needing to be aware of any details of the event sinks. As such, event monitoring presents a simple and flexible programming model. However a difficulty arises when trying to correlate events and infer some consequence from them. What is needed is a means by which applications can define high level interpretations for particular combinations of lower level primitive events - applications need support for expressing their interests in terms of patterns of primitive events i.e. composite event patterns.

2.2 Events

An event is a happening of interest, which occurs instantaneously at a specific time. System behaviour can be monitored in terms of a set of primitive events, representing the lowest level at which system activity is observable. However, large systems can generate many diverse events and it therefore becomes useful to be able to filter and correlate primitive events to generate composite events. This permits the level of abstraction to be raised to a convenient level. A composite event is specified as some combination of primitive events.

2.3 Event Classification

Each event source generates events of a particular type and maintains a list of event listeners that are interested in those events. For an event listener to be notified of the occurrence of an event, it must register its interest with the event source that generates the event. The alternative of broadcasting all events to all potentially interested parties does not scale well for distributed systems. Furthermore, registration has additional benefits:

- Event monitoring need only take place when an event listener is interested in the occurrence of an event.
- An event listener can reduce the number of uninteresting events that it receives by supplying an 'acceptance expression' to the event source to distinguish interesting events from uninteresting events.

2.4 Distributed Event-Based Systems

In a distributed system there are some additional problems to contend with which make it difficult to determine a consistent global view of a system's state:

- The absence of a global clock means that care must be taken when comparing timestamps on events which come from different sources.
- Variable and unbounded network delays cause difficulties in knowing whether an event has not occurred.

For example, consider a composite event pattern which is specified as:

event A, followed by event B without an intervening event C

In non-distributed systems this raises no additional problems. However, in distributed systems there is the possibility that having

received an A event followed by a B event, there is a C event in transit whose timestamp lies between those of A and B thus invalidating the match. This problem is depicted in Figure 1 and raises the issue of how long should an event monitoring system wait for (potentially) delayed events?

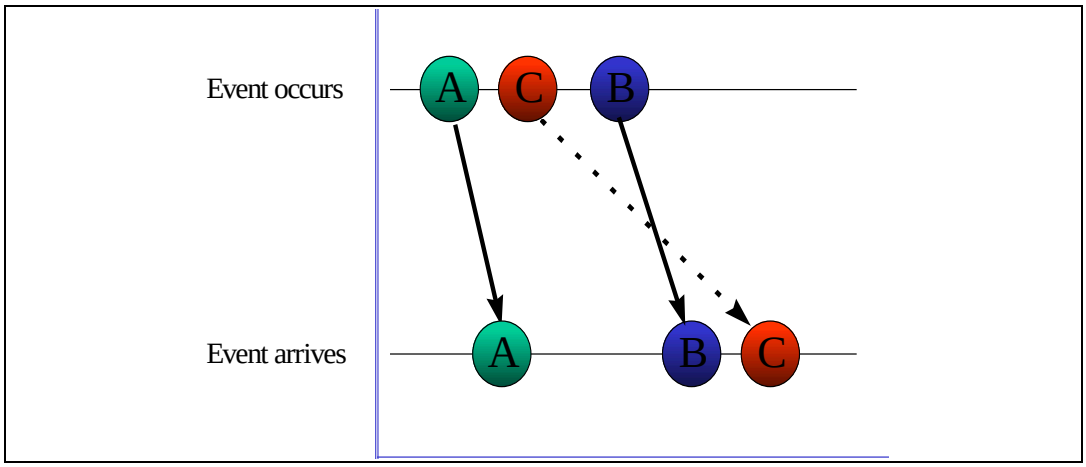

Figure 1 Occurrence c.f. Detection of Events in the presence of Variable Delays

Some event frameworks [1] define a detection window which specifies how long a possibly relevant event will be retained. Object Monitor takes a more flexible approach, defining 'heartbeat' events which provide timing information for each event source. Each event source can generate heartbeat events at regular intervals thereby providing information about the most recent activity associated with a given event source. Heartbeats enable the Object Monitor framework to distinguish the possibilities that:

- Either, a source has generated an event that may be of interest - in which case the framework must wait until it receives the event.
- Or, a source has not generated an event that is of interest - in which case the framework has all the information that is relevant to making the decision.

In the example above, the source which emits C events will also emit C heartbeat events. If a C heartbeat is received whose timestamp exceeds that of the B event, then there can be no C events that are relevant to this particular pattern and hence there is no need to wait any further. This approach is both safe and correct. It also allows applications to trade-off waiting time against processing demands - a faster heartbeat will allow an application to determine that it can stop waiting sooner but at the cost of processing more heartbeat events.

3 Composite Event Language

Object Monitor uses a declarative, easy to understand language for specifying composite events. It allows multiple concurrent evaluations to be active and was designed with distribution in mind.

3.1 Event Combination Operators

A composite event expression is constructed from primitive event expressions using the following event combination operators:

- `-` the 'without' operator: `X-Y` matches when an `X` event occurs without a `Y` event having occurred first.
- `;` the 'sequence' operator: `X;Y` matches when an `X` event is followed by `Y` event.
- `|` the 'inclusive or' operator: `X|Y` matches whenever an `X` event or a `Y` event occurs.
- `$` the 'whenever' operator: `$X;Y` means whenever an `X` event occurs a separate evaluation for `Y` events is started.

A composite event expression is evaluated in terms of:

- the time at which the evaluation is to begin, and
- an initial set of variable bindings (i.e. an environment).

An evaluation returns a set of tuples of the form: `(occurrenceTime, environment)` where `occurrenceTime` is the time at which the composite event triggers, and `environment` is a set of `(variableName, value)` pairs.

3.2 Side Expressions

Frequently it is important to be able to express an algebraic relationship between event parameters. This requirement is met by the use of side expressions, for example:

```
X(a,b){a>b}
```

matches an `X` event whose second parameter is greater than its first parameter.

3.3 Side Assignments

Object Monitor also allows arbitrary assignments to be made, for example:

```
X(a,b)[p=a+b];X(p,q)
```

matches a sequence of `X` events where the first parameter of the second event is equal to the sum of the first two parameters of the first event.

This is further extended by the `@` operator which represents 'now':

```
X(a,b)[t1=@];X(a,b)[t2=@]{(t2-t1)<10}
```

matches two identical `X` events which occur within 10 time units[1] of each other.

[1] These are currently ms.

4 Demonstrator

This section describes the functionality of the event monitoring demonstrators that form part of the Object Monitor deliverables.

4.1 Application Domain

The domain chosen for the demonstrators described in this report is that of network management. A simplified version of a real situation was chosen. It comprises a simple network of four nodes which are interconnected in a loop as depicted in Figure 2. Each node is connected to two neighbouring nodes by a cable which contains separate input and output connectors.

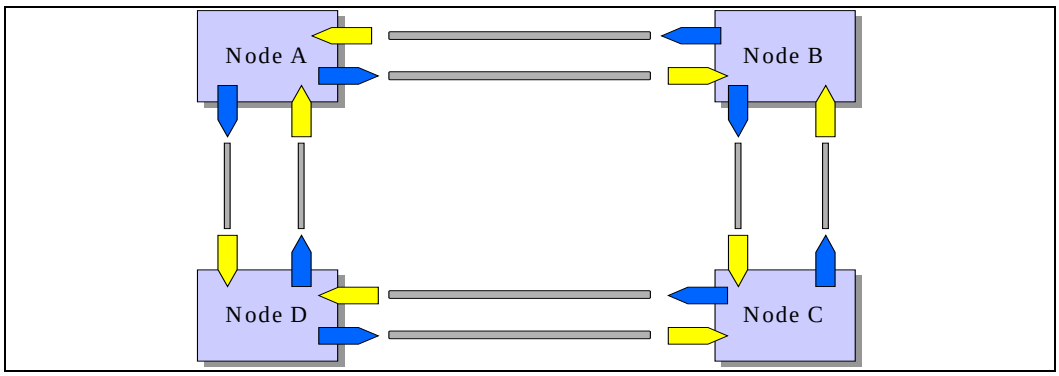

Figure 2 Network Management Scenario

4.2 Direct Graphical Manipulation

The network can be manipulated by the user to exhibit the following behaviour:

- Nodes can have internal faults which can subsequently be cleared.
- Cables can be broken and can subsequently be fixed.
- Connectors can become dislodged and can subsequently be reconnected.

The demonstrators model this situation using a GUI which has the following primary visual components:

- **Model**: a mouse-sensitive model of the network, allowing users to inject and clear device faults, break and restore cables, and dislodge and reconnect cables.
- **Pattern**: a means of specifying composite event patterns, allowing users to register interest in particular combinations of primitive events.
- **Outcome**: a means of notifying users about the composite events which have been triggered as a result of the actions that have been performed on the model by the user.

4.3 Event Patterns

The only active devices in the network are the nodes - the cables and connectors are passive. A node can detect that some internal fault has been raised or cleared, and detect that it has lost or regained its input signal. These are the raw material out of which events can be generated and these events are all that is available to an external entity to enable it to monitor the network's behaviour.

4.3.1 Node Failure

Some classes of fault can be inferred directly. For example, the failure of nodes can be monitored by an event pattern of the form:

```
N(device)= $F(device, code){code==0}
```
[2]

This event pattern matches whenever some device generates an `F` event whose code parameter is 0. The device parameter is assigned the ID of the device which exhibits the fault.

4.3.2 Security Violation on a Specific Node

A more specific node failure, such as a security violation on device 1 could be monitored by an event pattern of the form:

```
S()=$F(device, code){device==1&&code==1}
```

[2] N.B. These composite event patterns are simplified and hardwired for explanatory purposes.

4.3.3 Broken Cable

Because cables are passive (i.e. cannot generate events themselves), the only way that a broken cable can be detected is to infer it from the primitive events that are generated from nodes in such a situation i.e. when two neighbouring nodes report that they have lost their input signal:

```
B(devA,devB)=$(F(devA,code){code==2};F(devB,code)
    {!(devA==devB)3})
```

The matching composite event, `B(devA, devB)`, returns the devices which have become disconnected by the broken cable.

4.3.4 Loose Connector

As with a broken cable, a loose connector has to be inferred from a pattern of primitive events. The following pattern matches whenever any node reports that it has lost its input 5 times within 10 time units:

```
L(d)=$F(d)[t=@];F(d);F(d);F(d);F(d);F(d){@<t+10}
```

[3] This simplistic test may not be sufficient in practice to represent the notion that two nodes are neighbours.

5 Event Model

The Object Monitor event model is an extension of the Java Beans event model [2]. Before discussing these extensions therefore, a brief overview of the Java Beans event model is presented. This is followed by a discussion of the rationale behind the extensions that Object Monitor has made to the Java Beans event model.

5.1 The Java Beans Event Model

The goal of Java Beans is to define a software component model for Java, enabling Java components to be developed by independent software vendors and subsequently composed together into applications by end users.

The Java Beans event model, depicted in Figure 3, is based on the concepts of 'event listeners' and 'event sources'. An event listener is an object that has registered interest in receiving particular events from an event source. Each event source generates events of a particular type and maintains a set of listeners that wish to be notified when such events occur. Event sources provide an interface which allows listeners to add themselves and remove themselves from this set of listeners. Because Java does not support method pointers, callbacks are handled by pre-defined interfaces.

The Java Beans event model is strongly typed. For example, an event source that generates events of class `X` will have methods of the form:

```
addXListener(XListener listener)

removeXListener(XListener listener)
```

to enable event listeners to register and de-register their interest in `X` events. Listeners to `X` events must implement the `XListener` interface which has a `X`-specific methods, such as:

```
xFoo(X event)

xBar(X event)
```

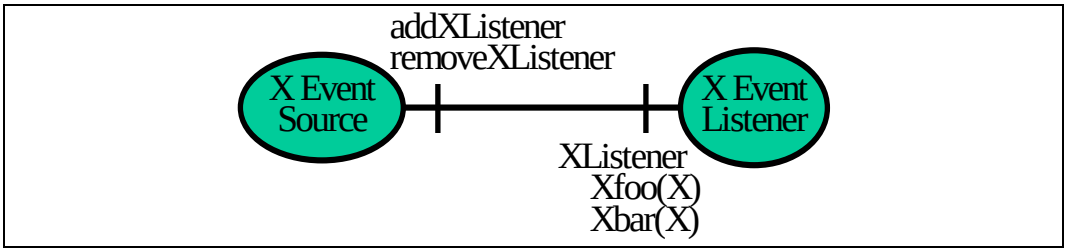

Figure 3 The Java Beans Event Model

5.2 The Object Monitor Event Model

As a consequence of its strong typing, the Java Beans event model does not directly accommodate using generic components for the non-functional aspects (such as filtering, buffering, distribution) of event management. For example, to insert a filter between an event source and an event listener, would require an event-specific filter class to be implemented. This is obviously untenable for an event framework which is to deal with an arbitrary set of event types.

Object Monitor therefore extends the Java Beans event model by introducing generic events between event sources and event listeners. This makes it possible to exploit the Java Reflection API to develop the generic components which perform event filtering, buffering, and distribution.

In addition to this, Object Monitor also:

- Enables listeners to monitor not only primitive events but also patterns of primitive events - i.e. composite events.
- Allows clients to be isolated from the details and actual sources of primitive events.

Object Monitor enables event listeners to register an interest in particular combinations of primitive events. These combinations are expressed in a form that is convenient to the event listener rather than being determined by the details of actual event sources. If and when a matching sequence of events occurs, the event listener is notified, via a callback, of the composite event and the aspects of it that they have expressed interest in.

5.3 Generic Events

Each source generates one or more distinct types of events. To employ a generic event monitoring framework, these events must be converted

into a common form so that all events can be processed generically i.e. without regard to the specific type of each event. A typical arrangement of sources is shown in Figure 4. This depicts two sources, `Src-A` generating `A` events and `Src-Z` generating `Z` events. `A` events and `Z` events must be converted into a common form (generic events) before they can be passed into the Object Monitor event framework. A generic event is essentially a means of placing a unifying wrapper around an object - the real identity and contents of the wrapped object are accessible using the Java Reflection API. The 'stubs' shown in Figure 4 perform the wrapping that is required to convert each specific event into a generic event.

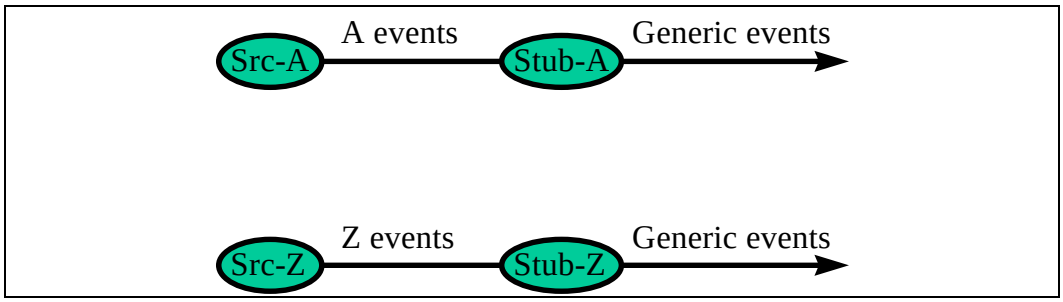

Figure 4 Typed Source Events must be Converted to Generic Events

5.4 Decoupling Listeners and Sources

5.4.1 Event Description

For event listeners to describe the events that they are interested in, it is necessary to be able to classify events. Events therefore are typed and can be further characterised by parameters. However, in large distributed systems, many components may be capable of generating many events, and events may be complex. This has the consequence that a source event may contain parameters which are of no interest to a particular listener.

To decouple a listener's view of an event from the source's view of that event, Object Monitor allows listeners to define a local view of source events. For example, a source may define an event type `Foo` with parameters `x, y, z` whereas a listener may only be interested in parameters `x` and `z` and wants a more convenient way of referring to the event. Object Monitor allows a listener to define a mapping between the source name of an event, e.g. `Foo`, and a listener's alias for that event, e.g. `A`, and also a mapping between the source event parameters and the parameters that are of interest to the listener. For example, a listener may use the form `A(b, c)` to refer to the parameters `z` and `x` respectively of source events of type `Foo`.

Thus, Object Monitor provides a means by which different applications can define different contexts in which events are classified. This is achieved by application-supplied mappings between source events and listener views of those events.

5.4.2 Event Sources

In the same way that it is useful to decouple a listener's view of events from a source's view of those events, it is also useful to enable listeners to describe the sorts of events that they are interested in without requiring them to have any direct knowledge of the location of the actual event sources. Object Monitor thus provides a trading mechanism to further decouple listeners and sources. This is another illustration of the way in which Object Monitor acts as an extensible framework - Object Monitor specifies an `import` interface for the trader but the implementation (i.e. sophistication) of this interface is entirely application-specific.

6 Architecture

6.1 Overview

The overall architecture of Object Monitor is shown in Figure 5. Event sources export themselves to a Trader. Event listeners register an interest in specific patterns of primitive events. The Object Monitor framework uses the Mapper and Trader to locate and register with the appropriate event sources that contribute to this pattern. These event sources then notify the Object Monitor framework when an event of interest occurs. The framework processes these events to notify the event listeners when matches to the (composite) patterns of interest have occurred. The event listeners can be thought of as a set of composite event listeners, the framework as a set of composite event talkers and primitive event listeners, and the event sources as a set of primitive event talkers. Thus, from the perspective of an event listener, the Object Monitor framework appears as a composite event talker.

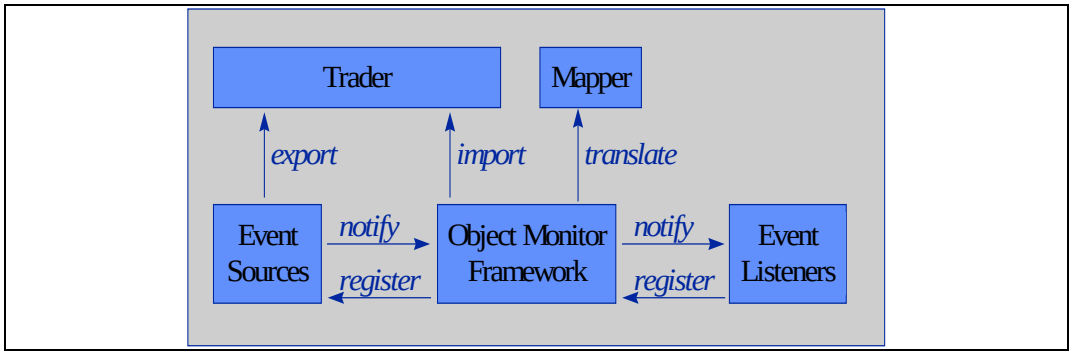

Figure 5 Event Sources, Event Listeners and the Object Monitor Framework

6.2 Layers

An event monitoring application developed using the Object Monitor framework can be arranged into the following distinct layers:

- Application layer. This contains the components that must be supplied by an application - the primitive event sources, the

composite event listeners, and implementations of the trader and mapper interfaces which allows the sources and the sinks to be decoupled.

- Composite Event layer. This is the core of the Object Monitor framework. It contains the generic event filtering and event correlation components which handle the event registrations, de-registrations and notification, and an interpreter for the composite event language.

- Transport layer. For non-distributed applications, this layer is empty thus imposing no overhead. For distributed applications, this layer can contain either the Object Monitor transport components or, where applications have specific transport requirements, application-supplied transport components.

The Composite Event and Transport layers constitute the Object Monitor framework. They comprise Java classes and interfaces which can either be used directly by an application, or extended according to application-specific needs in keeping with the architectural goals of making the Object Monitor framework flexible, open, and efficient.

6.3 Bridges

Most of the components within the Composite Event and Transport layers act as a kind of 'bridge' - a bridge is a component which implements both an event 'talker' interface and an event 'listener' interface:

- An event talker interface is an event source - it notifies its listeners of any events that are of interest to them.

- An event listener interface registers with an event talker interface to receive notifications of events that it is interested in. The listener can choose to listen to all events, or to some subset of events by specifying an acceptance expression which enables the source to filter out uninteresting events. Each listener can specify its own acceptance expression independently of any other listeners.

Each event talker and event listener interface deals with events of a given type. A bridge component may implement talker-listener interfaces which deal in terms of either the same types of event or different types of event. Talker-listener bridges which deal with the same type of event are a means of inserting some transparent functionality - for instance, a transport service for moving events of a given type from one host to another. Talker-listener bridges which deal with different types of event are the means by which the generic

functions required for event monitoring are performed. These bridges form the fundamental building blocks for the Object Monitor framework.

6.4 Chains of Talker-Listener Bridges

An application developed with the Object Monitor framework is essentially a chain of talker-listener bridge components which transforms generic events originating from diverse event sources into composite events which are sent to diverse composite event listeners. This is shown in Figure 6.

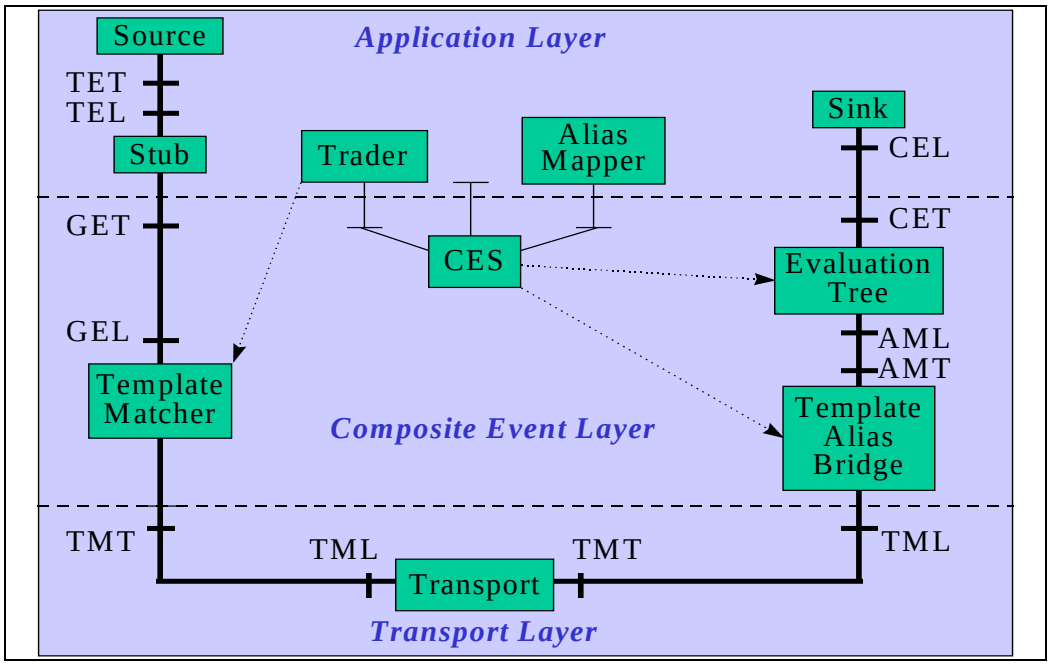

Figure 6 A Chain of Talker-Listener Bridges

The abbreviations used in this diagram are as follows (refer to 6.4.1 for an explanation of these event types):

TET	Typed Event Talker
TEL	Typed Event Listener
GET	Generic Event Talker
GEL	Generic Event Listener
TMT	Template Match Talker
TML	Template Match Listener
AMT	Alias Match Talker
AML	Alias Match Listener
CET	Composite Event Talker
CEL	Composite Event Listener
CES	Composite Event System

The overall effect of these bridges is to engineer the illusion of composite event talkers for composite event listeners in the spirit of the Java Beans event model.

6.4.1 Event Types

Object Monitor defines the following fundamental types of event:

- **Generic** events. These are the inputs to the event monitoring framework. An application must convert its typed events into generic events.

- **TemplateMatch** events. These represent an instance of a generic event which matches an acceptance expression which forms part of a composite event expression.

- **AliasMatch** events. These represent an event listener's view of a primitive event - essentially it extracts the parameters that a listener is interested in from a TemplateMatch event.

- **Composite** events. These represent a match to some composite event specification. For example, when the composite event expression: `F(x,y)=X(d,x);X(d,y)` matches, the composite event `F(x,y)` is generated with the parameters `x` and `y` from the AliasMatch events `X(d,x)` and `X(d,y)`.

6.4.2 Event Transformation

Generic events are transformed into Composite events by a chain of talker-listener bridge components:

- The `TemplateMatcher` bridge listens to Generic events (implements the `GenericEventListener` interface) and talks `TemplateMatch` events (implements the `TemplateMatchTalker` interface).

- The `TemplateAlias` bridge listens to `TemplateMatch` events (implements the `TemplateMatchListener` interface) and talks `AliasMatch` events (implements the `AliasMatchTalker` interface).

- The evaluation tree listens to `AliasMatch` events and talks Composite events.

Each application must implement the `GenericEventTalker` interface for each source that it wishes to introduce to this chain. It must also

implement the `CompositeEventListener` interface to receive notification of matches to composite event specifications.

Additional bridge components that listen to and talk the same type of event can be introduced into this chain at arbitrary points. This can be used for example to multiplex and demultiplex events from and to multiple sources and listeners or, as indicated in Figure 6, perform event transport where event sources and event listeners are on different hosts.

6.4.3 Class Collaborations

Figure 6 also introduces some additional infrastructure which is responsible for locating appropriate components, maintaining the decoupling between sources and listeners, and constructing the actual chain of bridges:

- `Trader`. Each application is responsible for providing some entity which implements the `Trader` interface. This entity is responsible for locating a suitable `TemplateMatchTalker` interface for a given (opaque) source descriptor.

- `AliasMapper`. Each application is responsible for providing some entity which implements the `AliasMapper` interface. This entity is responsible for mapping between a listener's view of the event world ('aliased events') and a source's view of the event world. It performs two related tasks: (i) it returns an opaque source descriptor for a particular aliased event, and (ii) it specifies the mapping between the positional parameters of an aliased event (listener event world) and the actual attributes of the corresponding real event (source event world).

- `CompositeEventSystem` (CES). Each application must construct an instance of this class, configuring it with the application-specific `Trader` and `AliasMapper` interface implementations. These interfaces enable the `CompositeEventSystem` to locate and register with suitable event sources when it constructs an evaluation tree for a composite event specification. The root node of the evaluation tree is connected to a composite event listener and the leaf nodes of the evaluation tree are connected to appropriate `AliasMatchTalker` interfaces.

An application expresses interest in a pattern of primitive events by specifying a composite event, using the composite event language described in Section 3. For each composite event specification, the `CompositeEventSystem` constructs a chain of talker-listener bridges (or augments the existing chain) and returns a `CompositeEventTalker` interface to the application. The application

then registers an appropriate `CompositeEventListener` interface with this to receive notifications when the composite pattern is matched.

6.5 Application Layer

The application layer contains the following components:

- On the source side are the Generic event talkers. Generally, sources generate primitive typed events. These events must be converted, via some application-supplied 'stub', into sources which implement the `GenericEventTalker` interface, providing timestamped Generic events which are suitable for passing into the Object Monitor framework.

- On the listener side are the Composite event listeners. These receive notification when a match is found for a composite event specification that the application has expressed an interest in.

- A `Trader` and `AliasMapper` implementation. The sophistication of these components is entirely application-dependent.

6.6 Composite Event Layer

The composite event layer contains the following components:

- `TemplateMatchers`. These listen to `GenericEventTalkers`, test the Generic events that they receive against `Templates` that have been supplied by their listeners, and notify them of any matches. A `Template` is an acceptance expression - it specifies a set of matching events in terms of the names and values of a subset of an event's attributes.

- `TemplateAliasBridges`. These deliver `TemplateMatch` events to all the `AliasMatchListener` objects that have requested them. To do this, they register with an appropriate `TemplateMatcher` for a particular event `Template`.

- An evaluation tree for each active composite event specification. This is implemented as a tree of nodes in which leaf nodes implement the `AliasMatchListener` interface and the root node implements the `CompositeEventTalker` interface.

- The `CompositeEventSystem`. This accepts a composite event expression and uses the `Trader` and `AliasMapper` objects to locate and connect appropriate bridge components.

6.7 Transport Layer

The transport layer is cleanly separated from and transparent to the composite event and application layers. It can be tailored to meet application-specific requirements without affecting other parts of the system.

6.7.1 Non-Distributed

In non-distributed applications this layer will be empty - the `TemplateMatchTalker` interface of a `TemplateMatcher` can be connected directly to the `TemplateMatchListener` interface of a

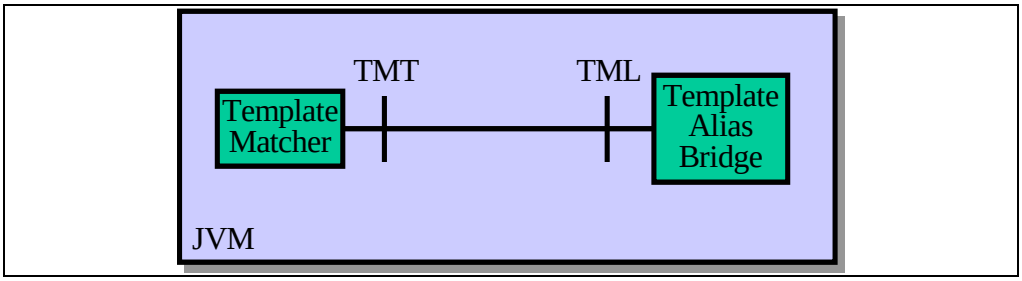

Figure 7 Non-Distributed Transport

`TemplateAliasBridge`. This is shown in Figure 7.

6.7.2 Distributed

In distributed applications the `TemplateMatchTalker` interface of a `TemplateMatcher` cannot be directly connected to the `TemplateMatchListener` interface of a `TemplateAliasBridge` because these components reside in different JVMs - see Figure 8. To address this, some engineering infrastructure must be inserted between the `TemplateMatcher` and the `TemplateAliasBridge` so that the transport of `TemplateMatch` events can be handled transparently to the composite event layer above.

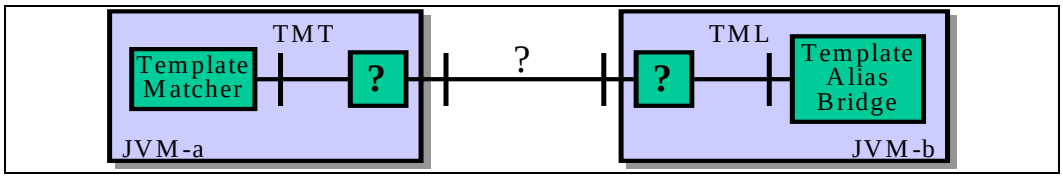

Figure 8 Distributed Transport

Handling the distribution issues in this way preserves the logical `TemplateMatchTalker-TemplateMatchListener` connection and gives a large degree of freedom to applications over the choice of transport engineering mechanism. For example, where applications

have specific distribution requirements, such as using multicast, or using CORBA IIOP for example, this can be cleanly accommodated by an application-specific transport layer without impacting the rest of the system.

6.7.3 Java RMI Implementation

The default transport infrastructure that has been adopted for Object Monitor uses Java RMI [3]. However, as stated above, providing that the logical `TemplateMatchTalker-TemplateMatchListener` connection is preserved, this can easily be replaced to suit application-specific requirements.

RMI places some additional restrictions on interfaces:

- A remote interface must extend the `java.rmi.Remote` interface.
- Each method in the remote interface must declare `java.rmi.RemoteException` in its `throws` clause.
- All arguments to, and return values from, a remote method must implement the `java.io.Serializable` interface.

It is important that these restrictions have minimal impact on the application and composite event layers. However, the obligation to implement the `java.io.Serializable` interface cannot be circumvented.

The Java RMI transport layer for Object Monitor thus essentially consists of the following additional interfaces and classes:

- A remote `TemplateMatchTalker` interface: `rmiTemplateMatchTalker (rTMT)`
- A remote `TemplateMatchListener` interface: `rmiTemplateMatchListener` (`rTML`).
- A source-side proxy which implements the interfaces: `TemplateMatchListener` and `rmiTemplateMatchTalker`.
- A listener-side proxy which implements the interfaces: `rmiTemplateMatchListener` and `TemplateMatchTalker`.

The remote proxy classes are used as shown in Figure 9.

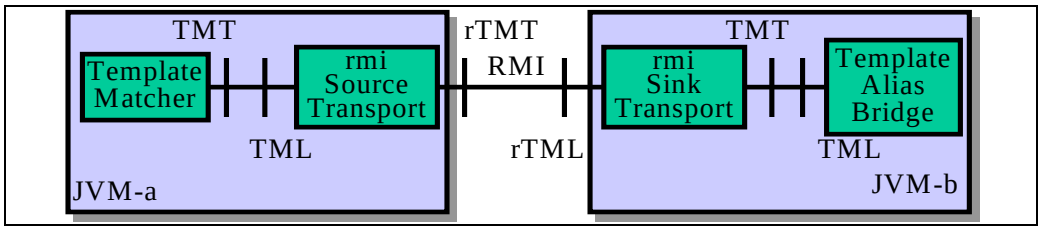

Figure 9 Remote Proxy Classes for TMT and TML Interfaces

RMI-specific implementations of the `Trader` and `AliasMapper` interfaces also need to be provided to enable the `rmiTemplateMatchTalker` and `rmiTemplateMatchListener` interfaces to be connected without the `CES` being aware that they are in fact remote.

The `CompositeEventSystem` obtains an opaque object source descriptor from the `AliasMapper` for a given event source. It then passes this descriptor to a `Trader` to obtain a `TemplateMatchTalker` interface. The `CompositeEventSystem` is totally unaware of the mechanism by which an application is distributed - it simply sees a `TemplateMatchListener` interface which it registers with a `TemplateMatchTalker` interface. The means by which this transparency is achieved is described in Sections 6.7.3.1 and 6.7.3.2.

6.7.3.1 Export

Each remote JVM in which event sources run creates an instance of `rmiSourceTransport` - see Figure 10. For each source that it wishes to export (a `GenericEventTalker`), an instance of `TemplateMatcher` is created. This is registered with the `rmiSourceTransport` object which in turn exports the remote event source to the RMI Registry.

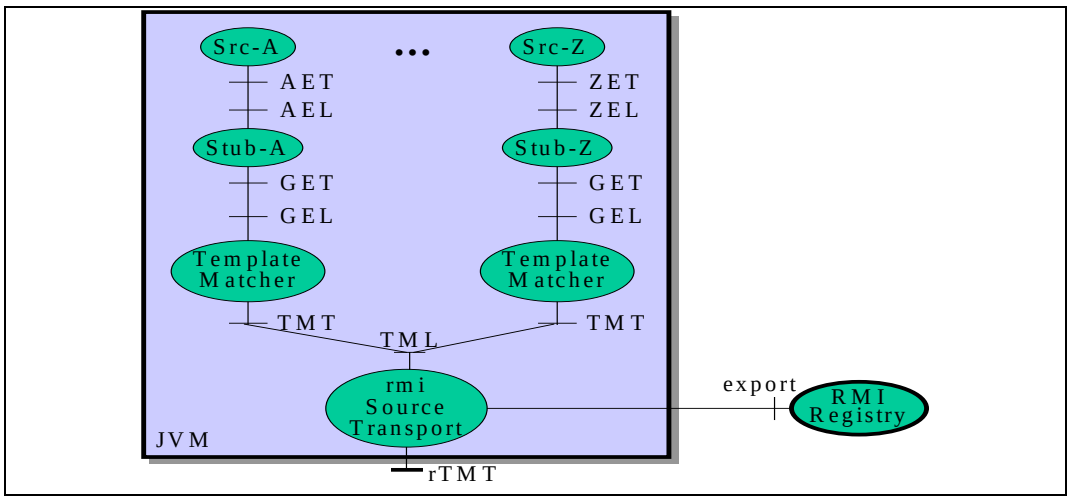

Figure 10 Source-side Transport

6.7.3.2 Import

When the `CompositeEventSystem` subsequently tries to locate this source - see Figure 11, the `Trader` (which runs in the same JVM as the `CompositeEventSystem`) imports this `rmiTemplateMatchTalker` interface from the RMI Registry, creates a local `rmiSinkTransport` object, binds the `rmiSinkTransport` object to the `rmiTemplateMatchTalker` interface, and returns the `rmiSinkTransport` object to the `CompositeEventSystem`. The `CompositeEventSystem` now has a handle to an object which offers a local `TemplateMatchTalker` interface and can therefore connect the `TemplateMatchListener` interface of the `TemplateAliasBridge` object to the `TemplateMatchTalker` interface of the `rmiSinkTransport` object. Transparent to the `CompositeEventSystem`, the chain is completed when the `rmiSinkTransport` object registers with the `rmiSourceTransport` object in response to registration requests from the `TemplateAliasBridge` object.

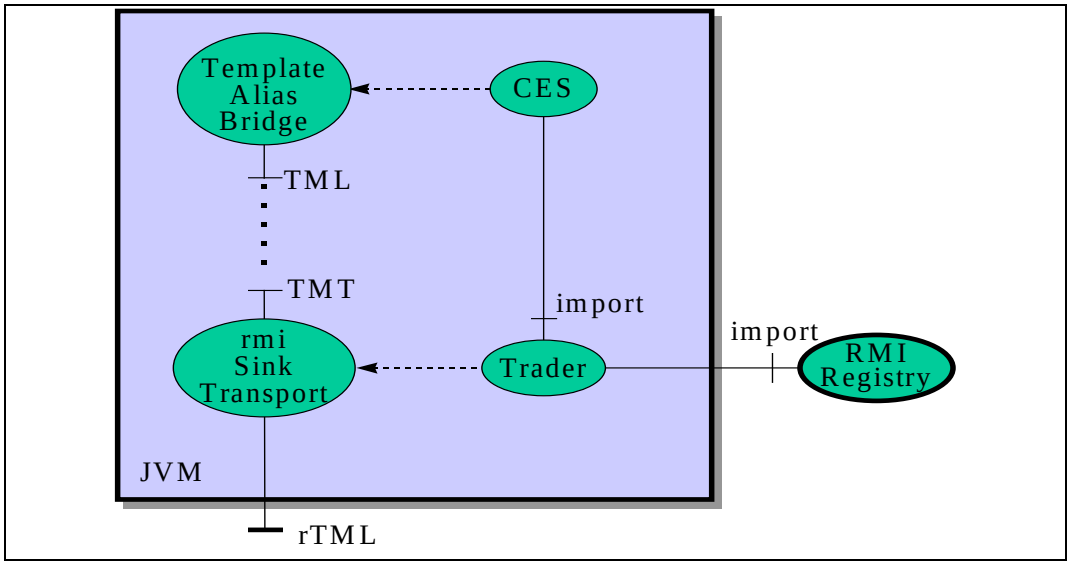

Figure 11 Sink-side Transport

7 Demonstrator Implementation

This section illustrates how easily non-distributed and distributed applications can be developed using the Object Monitor framework. It describes how the demonstrator that is described in Section 4 was implemented.

7.1.1 Non-Distributed Variant

In the non-distributed variant of the network management demonstrator, the Model, Pattern and Outcome widgets exist in a single window within a single JVM - see Figure 12.

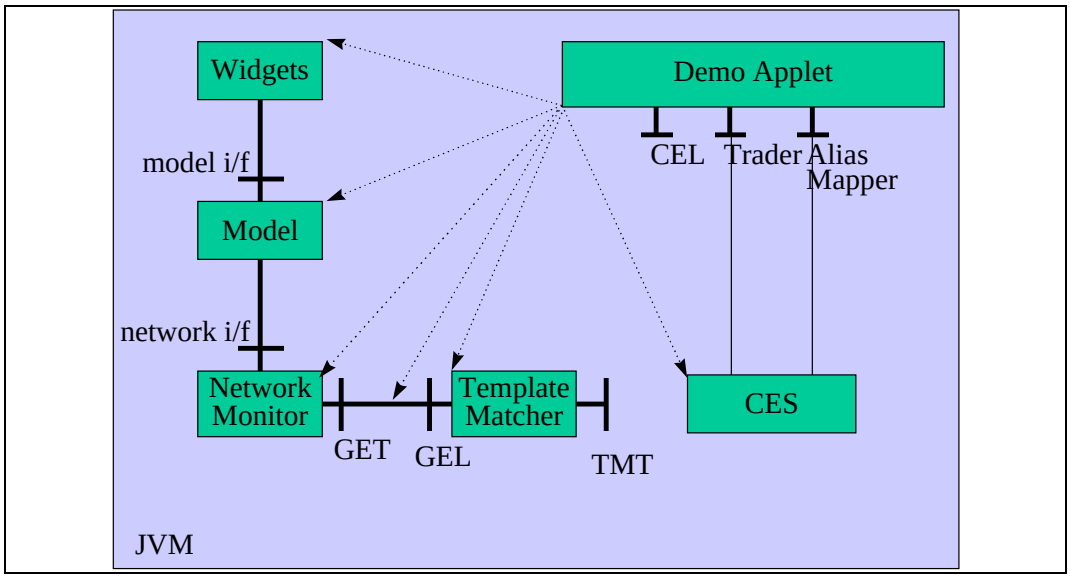

Figure 12 Single JVM Demonstration

The GUI widgets representing the devices, cables and connectors are connected to their model counterparts. These in turn invoke network management methods on the `NetworkMonitor`. The `NetworkMonitor` implements the `GET` interface that is needed by the `TemplateMatcher`. It thus generates `GenericEvents` as a result of the method invocations from the model, and sends them to the `TemplateMatcher`. The `DemoApplet` is responsible for creating the GUI structure and for connecting the `NetworkMonitor` to the

TemplateMatcher. It also 'exports' the TemplateMatchTalker interface to the Trader so that the CompositeEventSystem can locate an appropriate TemplateMatchTalker interface for the TemplateAliasBridge TemplateMatchListener interface.

7.1.2 Distributed Variant

In the distributed variant, the Model widgets exists in a separate JVM from the Pattern and Outcome widgets, thereby enabling the network to be monitored remotely. These JVMs may reside on different hosts - see Figure 13.

The JVM hosting the event source configures the TemplateMatchTalker interface of the TemplateMatcher in much the same way as for the non-distributed variant except that the TemplateMatcher is connected to a rmiSourceTransport whose rmiTemplateMatchTalker interface is then exported to the Trader.

The JVM hosting the event listener requests an appropriate CompositeEventTalker from the CompositeEventSystem in response to a pattern specified via the Pattern widgets. The CompositeEventSystem acquires a local TemplateMatchTalker interface from the rmiTrader which is in fact connected to the remote TemplateMatchTalker interface via the rmiSinkTransport and rmiSourceTransport bridges.

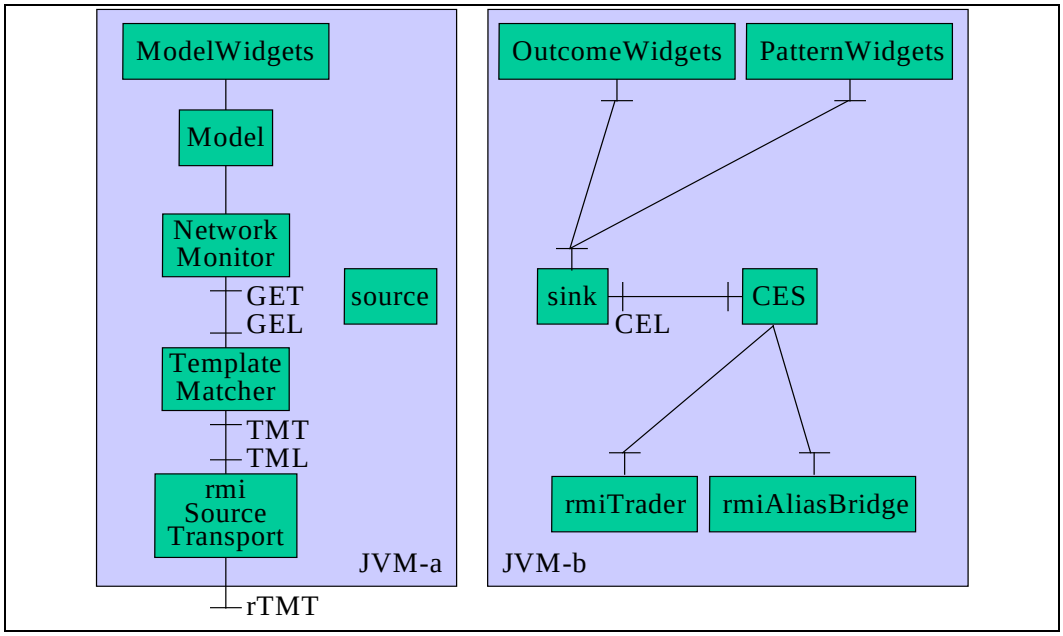

Figure 13 Multiple JVM Demonstration

8 Comparison With Other Event Frameworks

Recently, there has been considerable interest in event monitoring systems, as a consequence of which several event frameworks have been proposed. This section gives a brief overview of some of them in relation to Object Monitor: the CORBA Event Service, the CORBA Notification Service, Iona's OrbixTalk, and WebLogic's T3 Architecture.

8.1 CORBA Event Service

The CORBA Event Service [4] is concerned only with the dissemination of events between multiple event suppliers (event sources) and multiple event consumers (event listeners) via a named event channel. It supports two distinct communication models: consumer push and supplier pull. It provides no support for event filtering and correlation.

8.1.1 CORBA Notification Service

The OMG is well aware of the limitations of the Event Service and has issued an RFP for a Notification Service [4] which will extend the Event Service by defining 'standardised interfaces and mechanisms for enabling event consumers to express interest in events that are filtered by type and content, and for enabling event suppliers to provide filterable events to the service.'.

8.2 OrbixTalk

OrbixTalk [5] is an Orbix-specific extension of CORBA that allows shared information to be organised in a hierarchical structure of topics. An application registers interest in listening to one or more topic talkers.

OrbixTalk thus provides some support for event filtering, though the granularity is quite coarse, but provides no direct support for event correlation.

This [6] appears to be very similar to OrbixTalk in that the event service is oriented around a hierarchically organised topic tree. As a consequence it suffers from similar limitations to those of OrbixTalk.

9 Glossary

AML	AliasMatchListener
AMT	AliasMatchTalker
CES	CompositeEventSystem
CEL	CompositeEventListener
CET	CompositeEventTalker
CORBA	Common Object Request Broker Architecture
GEL	GenericEventListener
GET	GenericEventTalker
JVM	Java Virtual Machine
OMG	Object Management Group
RFP	Request For Proposal
RMI	Remote Method Invocation
rTML	remoteTemplateMatchListener
rTMT	remoteTemplateMatchTalker
TEL	TypedEventListener
TET	TypedEventTalker
TML	TemplateMatchListener
TMT	TemplateMatchTalker

10 References

[1] GEM: a generalized event monitoring language for distributed systems, Distrib. Syst. Engng 4 (1997) 96-108.

[2] http://www.javasoft.com/beans/spec.html, Java Beans API Specification.

[3] http://www.javasoft.com:80/products/jdk/1.1/docs/guide/rmi Java Remote Method Invocation Specification.

[4] http://www.omg.org/, OMG CORBA Event Service, Notification Service.

[5] http://www.iona.com/Products/Services/index.html, IONA Technologies, OrbixEvents, OrbixTalk.

[6] http://www.weblogic.com/, WebLogic's T3 Architecture.

[7] R.J. Hayton, OASIS An Open Architecture for Secure Interworking Services, PhD Thesis, Fitzwilliam College, University of Cambridge, 1995.