COBEA: A CORBA-Based Event Architecture

Chaoying Ma and Jean Bacon
University of Cambridge Computer Laboratory
Pembroke Street, Cambridge CB2 3QG, United Kingdom
—
http://www.cl.cam.ac.uk/Research/SRG/opera/
cm@cl.cam.ac.uk, jmb@cl.cam.ac.uk

Abstract

Events are an emerging paradigm for composing applications in an open, heterogeneous distributed world. In Cambridge we have developed scalable event handling based on a *publish-register-notify* model with event object classes and server-side filtering based on parameter templates. After experience in using this approach in a home-built RPC system we have extended CORBA, an open standard for distributed object computing, to handle events in this way.

In this paper, we present the design of COBEA - a COrba-Based Event Architecture. A service that is the source of (parameterised) events *publishes* in a Trader the events it is prepared to notify, along with its normal interface specification. For scalability, a client must register interest (by invoking a *register* method with appropriate parameters or wild cards) at the service, at which point an access control check is carried out. Subsequently, whenever a matching event occurs, the client is *notified*.

We outline the requirements on the COBEA architecture, then describe its components and their interfaces. The design and implementation aim to support easy construction of applications by using COBEA components. The components include event primitives, an event mediator and a composite event service; each features well-defined interfaces and semantics for event registration, notification and filtering. We demonstrate that COBEA is flexible in supporting various application scenarios yet handles efficiently the most common event communications. The performance of server-side filtering for various registration scenarios is presented. Our initial experience with applications is also described.

1 Introduction

Event communications are asynchronous compared with the request/response operations in the standard client/server model for distributed systems. There are many application areas where event-driven operation is the most natural paradigm: interactive multimedia presentation support; telecommunications fault management; credit card fraud; disaster simulation and analysis; mobile programming environments; location-oriented applications, and so on. An event is defined as the occurrence of some interaction point between two computational objects in a system. Such a point may reflect an internal change of state of the system, or an external change captured by the system. An event can be a base event which has a single source of generation, or a composite event which correlates multiple base event occurrences to be signalled as a whole. Events may be pushed by suppliers to consumers (the push model) or pulled by consumers from suppliers (the pull model) through specific or generic interfaces; such communication may be direct, or indirect i.e. through an intermediate object between the consumer and the supplier. Active systems monitor the occurrences of events and push them through to client applications to trigger actions [2, 5]. In contrast, passive systems require client applications to poll to detect event occurrences. An active system is therefore inherently more scalable than a passive system.

For example, in an Interactive Multimedia Presentation support platform [3] a script specifies *event-condition-action* rules to drive the interactive presentation. The events “Roger appears” and “Roger disappears” may be associated with frames 2056 and 3092 of a video presentation. If the user clicks on Roger (an area of the screen marked during preprocessing of the film) after "Roger appears" and before "Roger disappears" then the pause method is invoked on the video, a new window pops up, text on Roger is displayed and the film resumes when the user clicks again. Location devices, such as active badges or electronic tags, are another source of events. We may wish to analyse how users behaved during a fire drill in order to determine bottlenecks in a building [2]. We may arrange for our programming environment to move with us when we are detected moving from one workstation to another [1]. In telecommunication, various events are monitored by the management system and used for network analysis and fault recovery [12].

We have designed an architecture for building active, event-driven systems in a large distributed environment, where there is potential for high volumes of event traffic; for instance in telecommunication applications, a single source can generate tens or hundreds events per second. The design focuses on providing components to support easy construction of applications. The components include event primitives, the mediator and the composite event service; each features well-defined interfaces and semantics for event registration, notification and fine-grain filtering. We demonstrate that COBEA is flexible in supporting various application scenarios yet handles efficiently high event volumes in the prototype implementation. After experience in using this approach in a home-built RPC system we have extended CORBA, an open standard for distributed object computing, to handle events in this way.

1.1 Existing Work

Architectural frameworks for event handling in large distributed systems are discussed in [2, 7, 14, 19, 23, 24, 25]. The CORBA Event Service [14] introduces the concepts of event channel, supplier and consumer. An event channel is an intermediate object which decouples the supplier and the consumer. Event communication may be untyped in which a single parameter of type "any" is used for passing events; applications can cast any type of data into this parameter. For typed event communication, an interface **I** is defined in CORBA IDL. In the typed push model, suppliers invoke operations at the consumers using the mutually agreed interface **I**; in the typed pull model, consumers invoke operations at suppliers, requesting events, using the mutually agreed interface **Pull<I>**. Some application scenarios may be supported by the push and pull models, or a combination of them. But it is not possible for a client to select only those events which are of interest, at fine granularity, by means of a detailed specification of parameters and wild cards. Furthermore, in the push model, the supplier is responsible for acquiring the reference to an appropriate notification interface in order to push events to the consumer. The CORBA event service also lacks the ability to filter events; only filtering by interface type is available. Other major limitations include overly general, thus inefficient for many applications, lack of standard semantics and protocols for event channels and lack of type safety in untyped interfaces. Schmidt and Vinoski have reviewed the CORBA event service [21].

The Cambridge Event Paradigm [2] addresses some of the shortcomings mentioned above as well as some advanced event handling issues. The *publish-register-notify* mode is very well supported: a service that is the source of (parameterised) events publishes in a Trader the events it is prepared to notify, along with its normal interface specification. For scalability, a client must register interest (by invoking a register method with appropriate parameters or wild cards) at the service, at which point an access control check is carried out. Subsequently, whenever a matching event occurs the client is notified. Filtering by parameters including wildcard parameters and by event types at event sources are the key features, which eliminate the need of placing filters between the event server and client. For example, users may specify filtering criteria which describe the **PrintFinished** event on a file named "foo" (by giving the file identifier upon event registration), or on every file (by giving a wildcard file identifier "*"). The Cambridge work focuses on providing event handling primitives which are based on the direct push model. A heartbeat protocol has been incorporated, which can be tuned for the trade-off of computation cost between timely and delayed event evaluation in order to ensure correctness in the light of network failures. Access control on event registration have also been proposed; you may not be allowed to monitor the movement of your boss, for instance. In addition, a Composite Event Language and an evaluation engine have been developed to allow the use of composite events. The language currently has five operators: **Without (-)**; **Sequence (;)**; **And (&)**, **Or (|)**; **Whenever ($)** (see [8] for details). For example, a composite event **$Enter(x, r)** will trigger whenever someone **x** enters a room **r**; and an event **Enter(a, 123) |**

Enter(b, 123) will trigger if either person **a** or **b** enters room **123**.

The design, however, is based on a conventional RPC system rather than an object-oriented paradigm. The implementation uses MS-RPC3, a locally developed RPC system, thus has limited interoperability. Furthermore, it requires extension to the MS-RPC IDL for specification of events, thus has the need to marshal events separately from ordinary RPCs. It does not directly address issues of indirect event communications, although a composite event server may be used as an event mediator.

1.2 Objectives

Motivated by observation of the shortcomings of the existing work and the emerging new requirements, in particular from the application domains such as telecommunications, we have designed COBEA for event handling to extend the existing Cambridge Event Paradigm and the CORBA event service. The main goal is to design an architecture which provides a framework for object-oriented design and development of active application systems, especially in a large distributed environment. COBEA extends the CORBA Event Service, namely by supporting the *publish-register-notify* mode, parameterised filtering, fault-tolerance, access control, and composite events; all these features are missing from the CORBA event service. COBEA is a reincarnation of the Cambridge Event Paradigm with all the features mentioned above as well as support for dynamic addition of new event types and event mediator.

The basic requirement on an architecture for event handling is that events can be identified, classified, detected, specified and asynchronously reported to any interested party through a standard or application-defined interface. A general architecture, based on which large-scale distributed active application systems can easily be constructed, is clearly required. We believe that both direct and indirect event communication should be supported based on a wide range of application requirements, e.g. in the areas of CSCW (Computer Supported Cooperative Work), management in network, telecommunication and distributed systems, multimedia systems and mobile systems [2, 3, 6, 9, 13]. We focus on supporting the push model, i.e. after explicit registration of interest by consumers, events are pushed directly or indirectly by suppliers to consumers (a.k.a. the Notification Model). Event registration is essential for receiving selective event notifications. Such a scheme not only solves the main problems with synchronous communications - the saturation of network resources caused by polling operations, but also solves the problem of end user saturation caused by pushing everything through. The pull model can easily be supported by using existing technologies thus specific support is not necessary. The goals of COBEA are, in outline, as follows:

- Support direct/indirect notification of events
- Support interfaces for event notification
- Support interfaces for event registration
- Support fine-grain event filtering
- Support interfaces for management (e.g. register suppliers with an event mediator)
- Support composite events
- Support dynamic addition of user-defined event types
- Support security on event accesses (e.g. role-based event accesses)
- Support Quality of Service (e.g. reliable or fast, priority-based event delivery)

2 Overview of COBEA

The components of COBEA include event handling primitives with which an event sink and an event source interface are defined, and event services namely a mediator and a composite event server. The architecture is illustrated in Figure 1, where components may be as primitives (grey circles) or stand-alone servers (white circles). These components can be used by applications via standard interfaces; and once included, they handle events for the applications as indicated by the arrows in dotted lines. Figure 2 shows inheritance structure of the component interfaces, in which an application may define its own typed interfaces for handling events with a choice to extend or not to extend the standard/generic interfaces defined in COBEA. We focus on the direct/indirect push model which forms

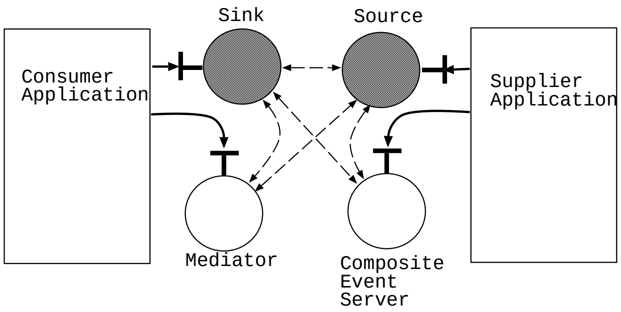

Figure 1: The COBEA Architecture

the core of event communication. The pull model can easily be supported by using the CORBA request/response, or most RPC communication primitives, or the forthcoming CORBA Messaging Service [17], or by incorporating the pull interfaces defined in the CORBA event service.

The event primitives support mainly event registration and notification operations defined by the event sink and event source interfaces. An application object can play the role of an event consumer or supplier by supporting the event sink or source interface respectively, or by supporting both interfaces, while providing other services at the same time (Figure 3 shows the incorporation of the event primitives in an application). Once registered the interest, the consumer will be notified whenever the event occurs. New interfaces for application-specific event handling can be derived from the primitive interfaces.

The event services define a number of objects acting either as event mediators or providing services for handling composite events. The main task of a mediator is to decouple the consumer and supplier by accepting events from the suppliers, and passing events only to the interested consumers. Thus consumers and suppliers do not need to know each other for communicating events. Many applications require notification of events from a number of suppliers in a specified pattern of combined events from these different sources. A composite event service is designed to meet such requirements.

One design principle is that the architecture should be lightweight yet powerful enough in order to support the construction of various distributed active systems. Some of the interfaces in COBEA may be defined by inheriting from the CORBA event service interfaces; for instance, the **snk** interface can extend

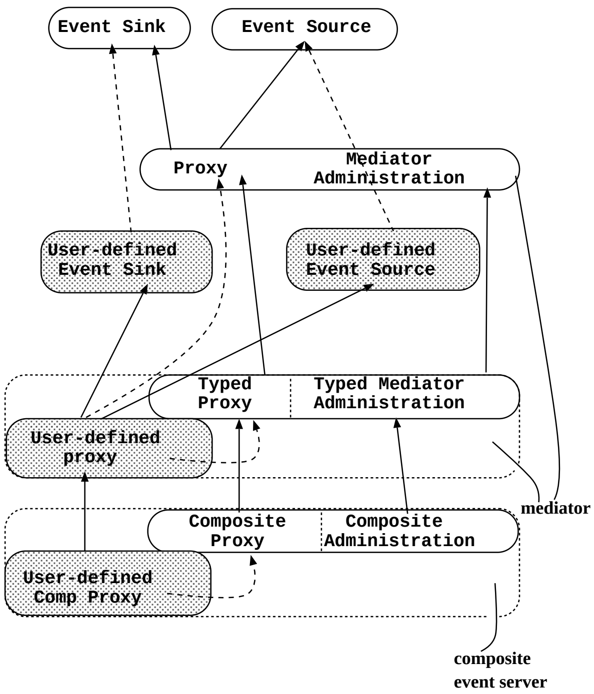

Figure 2: The Inheritance Hierarchy

Legends: round-corner rectangles represent objects; some object such as a mediator supports two interfaces. Shade objects have user-defined interfaces; other objects support standard/generic interfaces. Arrows show the inheritance from the interface that they point to; applications may choose whether to inherit if the arrows are in dotted lines.

the CORBA **PushConsumer** interface. Extending the CORBA event service this way means, however, that all the interfaces defined by the CORBA event service must be supported. We believe it is not necessary because in our Notification Model, most CORBA event service interfaces are undesired to use by applications. Later in Section 5.1, we will discuss how COBEA can be made to work with the CORBA event service.

Another design principle is that a filter should normally be placed on a supplier or a server to reduce the traffic to the consumers; the filtering criteria can be checked either at the supplier or at the server. It is important that filters should be kept as simple as possible. Sophisticated filtering which is less commonly used by most applications can be done at the application level rather than at the event system support level. There is a trade-off between the volume of event traffic generated and the complexity of supplier, mediator or consumer objects. Related work such as the ECA (Event-Condition-Action) rules in active databases [4, 5, 22] uses conditions

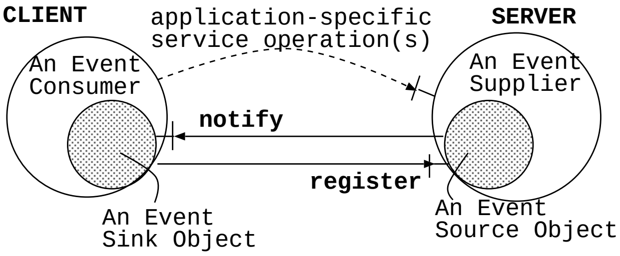

Figure 3: Direct event communication through primitives

which are like filtering criteria but can not be separated from either the evaluation engine or the action to take upon event occurrences. In our event architecture, filters can be placed at an event server, at a supplier, at a consumer, or chained among them, thus allowing less event traffic and greater flexibility.

Three options are available for implementing COBEA on top of the general-purpose communication system (e.g. an RPC system): create a new description language for specification of events, extend an existing IDL, or construct libraries to work with an existing IDL and its RPC system. The first two approaches allow the freedom to experiment with new ideas; the second approach in particular allows a standard extension to RPC systems for event specification. However, experience shows that application programmers are very reluctant to move existing programs, or write new ones, to make use of a non-standard environment. It is also very cumbersome for small research groups to maintain a non-standard RPC system, and keep its capabilities and performance competitive with that of a standard system. Thus, we base the implementation on CORBA - an open standard for distributed object computing [15]. We make the interfaces standard or follow a well-defined design pattern instead of using a non-standard IDL. As interoperability is concerned, CORBA 2.0 is designed to deal with heterogeneity and interoperability while most RPC systems are not.

This CORBA-based approach has the following advantages:

- Uses only standard IDL for events.
- No need to have a separate marshaling package for handling events.
- It is possible to allow the number of parameters in a registration interface to be different from that in the corresponding notification interface for the same type of event, if an application so requires.
- Type safety can be handled properly.

Based on the architectural framework, we are currently implementing a class library for COBEA. We have implemented the primitives, a composite event evaluation engine plus a parser based on the composite event algebra developed at the Cambridge Computer Laboratory. We are implementing two types of event service, namely an event mediator and a composite event service; the latter will incorporate the evaluation engine and the parser mentioned above. Furthermore, applications based on COBEA can easily be made to work with the CORBA Event Service, because all COBEA interfaces are specified using standard CORBA IDL. We have also developed a fault detection system for telecommunication network management based on COBEA.

3 The Design of COBEA

3.1 Primitives and Interfaces

Two objects are identified in the event notification model: a source and a sink of an event. It is essential for an event sink to receive events sent by an event source. The event source should support interfaces for event registration and deregistration; the event sink should support an interface for event notification. Both objects should also support a disconnect operation. The primitives also support the passing of a generic event header with standard attributes (properties), such as event identifier, creation time, type name, event source identifier and priority code. In addition, the interfaces allow a single parameter - event body - for passing application-specific event data dynamically. If more parameters are to be passed in applications, new interfaces should be defined, which can extend the primitive interfaces.

The standard interfaces are specified in the CORBA IDL as follows. The definition of exceptions is omitted.

```
module BaseEvent {
```

```
exception ...
struct EventTime
 {long sec; long usec; string clock_id;};

struct EventHeader {
 long id;  //the event id
 EventTime create_time;
 string event_type;
 string source_id;
 long priority; //severity code of event
 };

struct Duration {
 EventTime begin; EventTime end;
 };

struct ConsumerSpec {
 Object consumer_ref;
          //the consumer object reference
 Duration d; //for time specific filtering
 string QoS; //QoS constraint
 string who; //for access control
 };

interface Snk {
 void notify(in EventHeader e, in any data)
 raises (NotConnected);
 void disconnect_snk();
 };

interface Src {
 void register(
  in EventHeader e,
  in string header_filter,
  in any event_body,
  in string body_filter,
  in ConsumerSpec consumer,
  out long uid, //the consumer id
  out long eid) //the registration id
 raises (RegistrationFailed);
 void deregister(in long uid, in long eid)
 raises (UserNotFound, EventNotFound);
 };
};
```

At registration of interest, a number of filtering parameters are allowed, including a duration for specifying the start and end time of events of interest. A filter each for the event header and the event body can also be specified, which is defined by a string containing operators including "* " for wildcard, "==", ">=", "<=", "<", ">", and "!=" for comparison. Filters can be used in combination with the given value of the parameters in the event header or body. The position of the operators in a filter is important; they correspond to the position of the parameters in the event header or body. Each operator is two characters long with a trailing space in case of ">", "<" or "*". For more complex filtering, see the section on the composite event service. In addition, the relation among the expression of the parameters is conjunction. The parameter **QoS** is for the consumer to specify quality-of-service requirements such as reliable delivery or fast (unreliable) delivery of events. The parameter "who" can be used to pass user identification for access control. Upon registration, a **Template** will be created which describes what event should be sent to which consumer.

For notification, an event matching the template of an event registration should be passed by the source to the registered sinks. Upon notification, actions can be taken at the consumer depending on applications.

An event communication can be broken by invoking a **disconnect_snk** operation at the event sink, or a deregister operation at the event source. A deregister operation will either remove a registered consumer with all related event templates or only a particular event template. If a communication is closed by the supplier, the consumer receives a notification through the **disconnect_snk** operation. The communication can only be resumed upon another **register** operation.

It should be noted that the defined interfaces allow only a standard event header with fixed number of parameters; the interfaces are also standard. For many applications, specific interfaces can be defined by following a well-defined design pattern in IDL files, e.g. **register<T>()**, where **T** may by substituted by a **DrawEvent** or **AccountingEvent** for a drawing or an accounting application respectively. A common set of event manipulation operations, such as comparison of the occurrence of an event against the registered templates, are supported through a common class library.

An example of an application-defined event sink interface may look as follows, where T_i is a type name. k parameters are used in this example. The interface extends to the standard **Snk** interface.

```
interface Snk<T>: BaseEvent::Snk {
 void notify<T>
  (in T1 arg1, in T2 arg2,..., in Tk argk)
 raises (NoSuchType, NotImplemented);
 };
```

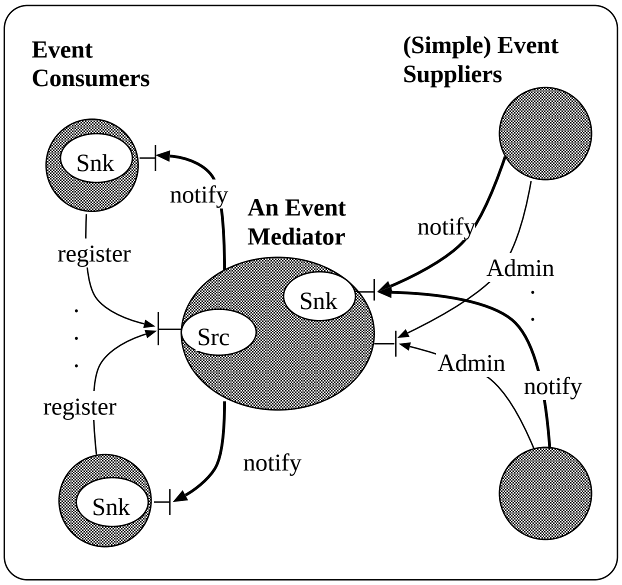

Figure 4: Indirect event communication through a mediator (only simple suppliers are shown)

3.2 Mediator and its Interfaces

For many applications, it is useful to have an event mediator. Figure 4 shows the indirect model of event communication. The advantages of having a mediator are: (1) a consumer or a supplier does not have to keep all the contacts to every event supplier or consumer but only the contact to the mediator; (2) simple event suppliers can be built which do not support a registration method; (3) commonly used filters may be built once for all; e.g. a filter at a mediator may be placed for all **faulty** events from one or more suppliers; (4) it is also easier to adopt group communication protocols such as a reliable multicast protocol at a mediator.

One assumption on the mediator is that a supplier needs a mediator to publish events; an event may be published by its type name and/or attributes (parameter names). A consumer needs to find a mediator to receive events. Finding a mediator is orthogonal to using it. Particular bindings between mediators, suppliers and consumers may also be arranged.

For standard events, the interfaces are as follows. Users may attach an application-specific piece of information when registering a new supplier.

```
module Mediator {
exception ...
interface Admin {
 Object new_supplier(
  in string appl_info,
  in boolean relay,
  out long uid)
 raises(RegistrationFailed);

 void remove_supplier(
  in long uid,
  in string application_info)
 raises (NoSuchSupplier, NoFound);
 };

interface proxy:
 BaseEvent::Snk, BaseEvent::Src {
 boolean lookup(in string type_id)
 raises (NotFound);
 };
};
```

For application-specific events, the interfaces should again be defined in the IDL files. Moreover, a generic interface is required for registration and notification of these events in many applications, e.g. event notification in telecommunication management. The generic interface allows dynamic addition of new event types and does not require all event types to be defined in IDL files. It is possible for a particular implementation to support only the generic interface. An exception **NotImplemented** will be raised if operations defined by **Snk<T>** or **Src<T>** are invoked. The generic interface of a mediator is defined as follows.

```
module TypedMediator {
exception ...
interface TypedAdmin: Mediator::Admin {
 Object new_typed_supplier(
  in string type_id,
  in boolean relay,
  out long uid)
 raises(RegistrationFailed);

 void remove_typed_supplier(
  in string type_id, in string uid)
 raises(NoSuchSupplier, NoSuchType);
 };

interface TypedProxy {
 void typedProxyRegister(
  in string type_id,
  in boolean relay,
  in NVList *arglist,
  in short argno,
```

```
  in string filter,
  in ConsumerSpec consumer,
  out long uid,
  out long eid)
 raises(RegistrationFailed);

 void typedProxyNotify(
  in string type_id,
  in short argno,
  in NVList *arglist)
 raises(NotConnected);

 TypedProxy typedProxyLookup(
  in string type_id)
 raises (NotFound);
 };
};
```

The semantics of a mediator (either generic or application-specific) depends on the types of suppliers, which can be simple or sophisticated. To support a simple supplier, a mediator will not register events at the suppliers. It accepts any event from the registered supplier, matches it against the registered event templates and notifies the consumers. To support a sophisticated supplier, a mediator can relay event registrations to the suppliers as required or process the events as for a simple supplier. To relay, a mediator does nothing but register the consumer's reference and assign a **user_id** to the consumer; the **user_id** is useful for the consumer to deregister its interest, and for the mediator to tell which consumer the received event belongs to. After this, the mediator invokes the **register<T>** method at the supplier with its own reference (instead of the consumer's reference) and the **user_id**, and then waits for notification. Upon notification, the mediator relays the notification to the consumer by invoking the **notify<T>** method at the consumer. Upon registration and notification, the mediator needs to construct a specific interface for registration e.g. **register<T>** at the supplier, and a specific interface for notification e.g. **notify<T>** at the consumer in case a **TypedProxy** is used. In both cases, the supplier should inform the mediator about the event types it notifies before the mediator accepts any registration from a consumer for the events.

3.3 Composite Event Service

There is an increasing demand for using composite events, for example, in telecommunication network management, several alarms raised by some network devices may contribute together towards a particular network problem known to the network manager; this requires that several events (i.e. alarms in this case) be signalled as a whole to the consumer (i.e. the network manager in this case). A composite event server is therefore included in our architecture as one of the main components. Specification of composite events needs to follow a well-defined syntax to allow standard parsing by a composite event server. A composite event algebra has been developed at Cambridge on which a composite event language is based. For instance, a sequence of events **A** and **B** is specified as **A** ; **B**, and event **A** or event **B** happens is specified as **A** | **B**.

One way to register a composite event with a server is using an application-specific interface. Typical parameters in such an interface include event type name, a list of parameters associated with the event in which each parameter is represented by a structure (e.g. NamedValue in CORBA) with attributes such as parameter name, parameter type, parameter value, parameter mode (i.e. in, inout or out), a filtering string, a string expression of the composite event (e.g. **A** | **B**) in the Cambridge Composite Event Language, and other parameters such as the consumer's reference, duration, QoS etc. The interface looks like:

registerComp < **CT** >(**string** $event_1$, **NVList** $parameters\_list_1$, **string** $filter_1$, ..., **string** $event_n$, **NVList** $parameters\_list_n$, **string** $filter_n$, **string** *expression*, **Duration** d, ...);

where **CT** is the type name of a composite event. As before, the position of the characters in the filter corresponds to the position of each of the parameters in the list.

Another possible way to register a composite event is using a standard interface in which expressions of composite event are specified in a well-defined syntax (e.g. the constraint language from the OMG Life Cycle Service [14]) and passed as a string. For example, a composite event may be expressed as:

"**event_type** = "enter"; **room** ="T14"; **person** = "Oliver"; **duration** = "Mon to Fri";" | "**event_type** = "absence"; **room** = "T14"; **person** = "*";"

The interface may look like:

registerComp(**string** *expression*, **Duration** *d*, ...);

We concentrate on the former because it is consistent with our interface design in COBEA. It is also useful to have a generic interface for composite event registration as it is for base events. To be notified of a composite event, a consumer has to submit upon registration the parameters to be passed through the notification. If the base events are not available at the server, the server will look for the suppliers; this is similar to the lookups by a consumer for a supplier. The interface is as follows.

```
module CompositeEventServer {
exception SyntaxError {};
interface CompAdmin:TypedMediator::TypedAdmin {};

typedef struct BaseEvent
 {string type_id; NVList *arglist; string filter};

typedef sequence<BaseEvent> CompEvent;

//generic composite event registration
 void typedRegisterCompEvent(
  in short eventno, //the number of base events
  in CompEvent comp_event, //related base events
  in string expression,//describes the comp. event
  in short out_argno,
  in NVList *out_arglist,
  in ConsumerSpec consumer,
  out long uid,
  out long eid)
 raises (SyntaxError,RegistrationFailed);

  void typedNotifyCompEvent:
   TypedMediator::TypedProxyNotify {};
  };
```

The CompAdmin is for a supplier to register itself with the server by indicating the base events it supports, and to get a reference to a proxy for passing the base events. There is no difference from a supplier's point of view whether a base event is used in a composite event or not.

Upon a registration of a composite event, the server will analyse the parameter **comp_event** to retain the type name, the parameters and the filters for each of the base events. The relation of the base events is obtained from the **expression**, e.g. **A;B;C** where **A, B** and **C** are base event type names. The server also retains from **out_argno** and **out_arglist** the parameters for constructing a notification interface to invoke at the consumer. More complex filtering is possible given the support for composite events. For example, consumers may specify a list of parameter values in events to be received ; a composite event **A**(12, "foo", "<===") | **A**(14, "foo", ">=== ") may be used for an event filter which checks if the first parameter is less than 12 or larger than 14, and the second parameter is "foo". Note that the composite event is expressed here intuitively rather than by using the interfaces defined in this section.

4 Building Applications with COBEA

In this section we list some application scenarios supported by COBEA.

4.1 Application Scenarios Supported

Scenario 1
An application creates a mediator object as a notification service with a generic interface for event registration and notification. OrbixTalk [10] and the TINA notification service can be constructed this way in COBEA. This scenario is not well supported by either the CORBA Event Service or the Cambridge Event Paradigm, in the latter a composite event server would be used for this purpose. Examples of such scenarios can be found in [10, 13, 21].

Scenario 2
An application defines its own typed interfaces for events which can be supported by the class library implemented for the primitives in COBEA. The applications supported by the Cambridge Event Paradigm can all be constructed this way in COBEA. This scenario is not well supported by the CORBA Event Service, and not supported at all by the TINA notification service in which an intermediate server is enforced. Examples of the scenario can be found in [3, 11].

Scenario 3
An application defines its own proxy for typed events without inheriting from the generic **TypedProxy** interface. The CORBA typed

event channel can support this scenario but no provision for registration of events is available. Furthermore, three steps must be carried out for connection to a CORBA event channel: (1) get an object reference for a factory which returns the reference to the proxies; (2) get an object reference for the supplier/consumer proxy; (3) connect to the proxy. It is much simpler to connect to the mediator than to the event channel.

4.2 An Example of Telecommunication Application

Our preliminary experience of using COBEA for building distributed active application systems includes developing an alarm correlation system for network management in telecommunications. This work [13] shares motivation and scenarios with alarm correlation research being done in Nortel Technology [20], but offers a solution to the problem that differs in key design elements, and offers it on the COBEA platform. Alarm correlation including alarm filtering can occur at several levels in the progress of an event from the raising object (usually a network device) through any intermediate critical real-time controlling software to the network manager. These levels (in the order of the lowest to the highest) are: hardware element; real-time control; system management. Our focus is on the system management level.

Alarms can indicate possible problems (i.e. raise hypotheses), can confirm existing hypotheses or can be accepted (without change of state) by existing hypotheses or existing (confirmed) problems. We use the Composite Event Language to express the complex relations between alarms and problems/hypotheses by treating alarms as base events and problems/hypotheses as composite events. We employ the evaluation engine and composite event language parser implemented in order to monitor and trigger these composite events. The system manager supplies the alarms (base events) to the composite event server, which monitors the composite events (problems/hypotheses) and notifies the problem/hypothesis browser in the alarm correlation system. After a problem has been diagnosed, appropriate actions can be taken at the system management level for network restoration. The work has shown that it is feasible to support active alarm correlation using COBEA and the Composite Event Language, and has suggested directions to improve the language [13].

Currently, we are building a trial system that features a graphical user interface for registering, deregistering and browsing composite events, and an event generator which can generate events of any specified type at a given interval. Performance will be measured to see if the system is suitable for online real time alarm correlation. Some real-time issues are discussed in [7]. Our experience shows that an event mediator can be used as a front end active database that incorporates the existing network management information base which supplies the network configuration data, and as a proxy for some of the dumb devices which signal alarms but do not understand CORBA.

5 COBEA Performance and Other Issues

COBEA is a general event architecture for building distributed active systems. Its main goal is to allow scalability by reducing the volume of notifications. The goal is achieved by implementing efficient filtering at event source. Our initial measurement against the prototype implementation has shown that filtering at event source is crucial for the system to scale as the volume of events increases. Some domain specific issues such as real-time issues described by [7] are not particularly addressed by COBEA. COBEA could be tuned for specific domains if required.

These tests were run on two Digital Alpha AXP 3000 workstations connected by a 155Mbit ATM network. One (for the event source and/or the event sink) is AXP 3000/900 with 275MHz CPU; and the other (mainly for the event sink) is AXP 3000/300 with 150MHz CPU; both have dual-issue processors running the OSF V3.2D-1 operating system. The event system and applications were built with g++ 2.7.2 with -O2 optimisation. The load was light on both machines most of the time during the testing, although the Alpha AXP 3000/900 normally had about 50 users and around 400 processes. We run many sinks on the Alpha AXP 3000/300 to avoid causing significant delay on the shared Lab machine (Alpha AXP 3000/900).

Firstly, we conducted the latency tests to determine

# events registered	filter & copy (μs) / # consumers		
	1	10	50
10	4.9	28	140
50	5.9	32	N/A
100	6.7	33	N/A

Table 1: Event Filtering Latency for One or More Consumers

the latency of filtering at event source. Secondly we conducted volume tests to determine the impact of event volume increase upon the event server.

Table 1 shows the result of the latency tests. When events occur, the server tries to match them against the registered event templates. The latency of event template matching increased logarithmically as shown in columns 1 and 2. For instance, column 1 shows that as the number of registered events increased from 10 to 100, the latency was 4.9μs, 5.9μs, and 6.7μs for 10, 50, 100 registered events respectively, for an average of 1000 runs. The latency for one server and multiple consumers increased linearly as the number of consumers increased (as shown in row 1); this is due to preparing events to send to each consumer after template matching. However, as shown in Figure 5, the overall cost of filtering is small; only the matched events are copied. Event dispatching delay in the case of one source and one consumer was 39μs on average when 100 events were registered. The overall delay for event creation, template matching and event dispatching (i.e. moving the events to the "sendqueue") were 271μs. The total latency between the occurrence of an event and delivery to the consumer is estimated to be less than 1 ms.

The advantage of filtering at event source is clearly shown in Table 2. We tested the effect of increasing event volume in two modes: *raw*, when no attempt to recover missing events was taken; and *normal*, when event sequence numbers were checked and attempts to recover missing events were made. When 10 events were registered by the consumer at the event source, the consumer detected no missing event at an event volume less than or equal to 2000Hz. In contrast, when 100 events were registered, events started to be missed when the volume reached 20Hz. If events were not recovered after loss, all registered events were correctly delivered to the consumer at the event rate as high as 8000Hz or beyond. Our test events were randomly generated integers between 1 to 1000. Parameterised filtering means that the consumer can register, say, number

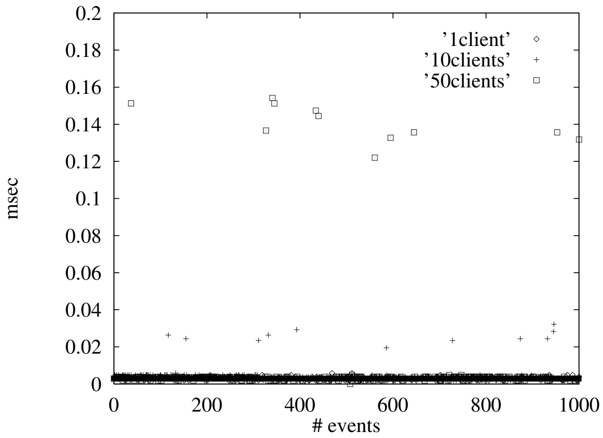

Figure 5: The Overall Cost of Filtering and Preparing to Send Events at Source

# events registered	100% delivery	
	raw (Hz)	normal (Hz)
10	> 8000	2000
100	20	10

Table 2: Event Volume from a Single Source to One Consumer

1 to 10 or as required. This is a real advantage over type-based filtering, in which case the consumer has no choice but to be overwhelmed by events.

It is interesting to notice that the cost of the integrated fault tolerance (i.e. sequence number checking and event recovering) was not as costly as we first thought, especially when the event volume and the number of registered events were low: i.e. at a volume less than 2000Hz when 10 events were registered. However, as the number of registered events increased, the system became much more sensitive to event volumes. Our experiments show that as 100 events were registered, about 98% of events were received by the sink in *raw* mode, while only 55% were received in *normal* mode. As the event volume goes extremely high, such cost will become expensive as it contributes to the load on both the source and the sink. In the case that many sinks try to recover missing events from a single event source, the source may eventually not be able to cope. COBEA provides solution to this by allowing replication of event sources and partition of the sinks, so each source is responsible to only a small number of sinks.

The current implementation may be improved for better scalability by allowing multicast of events, thus removing the need to copy events. One potential obstacle to scalability is the integration of fault tolerance irrespective of the underlying communication platform; the cost can be eliminated if the communication support is reliable.

5.1 Other Issues

- **Dynamic Addition of New Event Types or Data**
  Dynamic addition of new event types is supported by using the generic typed interfaces defined in COBEA. Dynamic attachment of application-specific data to a generic event is supported by using the parameter **event_body**. Any type of data may be cast into this CORBA **any** parameter, which has two fields: **_type** and **_value**; **_type** can be checked in order to get the **_value** correctly.

- **Working with the CORBA Event Service**
  COBEA may be implemented alongside, or as an extension, to the CORBA Event Service. We have chosen the former for our current approach. A better alternative might have been to extend the CORBA Event Service interfaces to incorporate the new features supported in COBEA for standardization purposes. For example, the COBEA **snk** interface may be defined as follows:

```
#include CosEventComm.idl
module BaseEvent{
      ...
interface Snk:CosEventComm::PushConsumer{
 void notify(
  in EventHeader e,
  in any data)
 raises (NotConnected);
 void disconnec_snk();
 };

interface Src { ... };
};
```

  Also as mentioned earlier on in this paper, the pull model can easily be supported in COBEA by simply incorporating the pull interfaces of the CORBA Event Service. For example:

```
#include CosEventComm.idl
module BaseEvent{
      ...
interface COBEAPullSupplier:
 CosEventComm::PullSupplier{};
      ...
 };
```

- **Event Naming and Locating**
  In COBEA, event type names are the same as interface type names, therefore a trader can be used to handle event naming and location; an event with parameters is like a service with attributes. If consumers are concerned with the content of an event (i.e. particular values of parameters), e.g. IBM stocks and Microsoft stocks of the **StockQuote** event, filters must be used to receive the quote of a particular stock only; a trader is not enough here.

- **Event Buffering and Logging**
  It is possible for events to happen before interest has been registered. Sometimes, a consumer can not digest all the events being supplied. It is therefore useful to buffer events at suppliers or mediators. A Time-To-Live (TTL) parameter can be associated with an event instance to make sure an event will not be discarded too quickly by the supplier or the mediator. It is useful to allow the consumers to specify a TTL. For some applications, events should be logged or made persistent if they may be subject to frequent query later. For instance, a security audit server may want to monitor logins by users, and log those events as evidence in case somebody attempts to use unauthorised resources.

- **Service Configuration**
  We propose a hierarchical structure (i.e. a rooted acyclic graph) for organising the servers which provide an event service cooperatively. Events generation may be partitioned among the servers, thus if a server does not know about a certain event itself, it can get help from the server which knows. In Figure 6, five servers are responsible for supplying events partitioned in the event groups Gr, G1, G2, G3 and G4 respectively.

- **QoS**
  In COBEA, a priority parameter in the event header can be used to specify the priority of an event. In addition, event queues are maintained according to priority for each event type, and events are sent FIFO within a priority. The **QoS** parameter in the **register()** operation can be used to specify speed or mode of event delivery, i.e. fast or reliable mode; the former has no guarantee of delivery of the

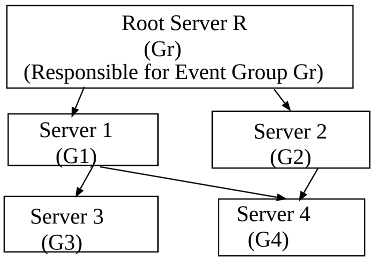

Figure 6: Composite event servers involved in handling a composite event

event, while the latter is implemented on top of a reliable transport protocol, thus can guarantee reliable delivery (e.g. events are delivered at least once and in order). Events are notified in the order of occurrence within a priority. The parameter may be used in many applications, for example, in telecommunication where fault events (or alarms) have to be delivered quickly so that the fault can be identified and rectified, however, performance events can tolerate some delivery delay as they are generally analysed off line at some later time.

- **Security**
In many applications, events are required to be delivered or viewed only by authorised consumers. An object system where access control is based on object or method invocation only has not sufficient support for such requirements. Event-based access control requires that access control can be carried out against each event occurrence. On the one hand, suppliers or mediators (i.e. servers) are responsible for checking if an event can be delivered to the interested consumers. On the other hand, the servers must have the right to invoke the **notify()** operation at the consumers. In COBEA, a consumer must supply its user/role name using the **who** parameter when registering events. The parameter will be validated by the server to make sure that the consumer has the right to access the particular event. In order for the supplier to invoke the notification operation at the consumer, an ORB supporting secure method invocation, as specified by the recently adopted OMG CORBA Security Standard [16], can be used. Although many events have been excluded at registration time, more access control is still required in an event-based system. To avoid computation explosion in the order of $\mathbf{O}$(no. of events $\times$ no. of consumers), coarse-grained access control based on object domain, event type, or method invocation only can be used as an alternative to the fine-grained control based on event occurrence. Failing this, optimised validation is still possible for access control depending on client credentials or event data only but not both.

6 Related Work

COBEA shares the major goals with the existing architectural frameworks for event handling in large distributed systems [2, 14, 18, 23, 24, 25]. The CORBA Event Service and the Cambridge Event Paradigm were reviewed above. Despite its weaknesses, the CORBA Event Service is nonetheless influential. Some recent work and products [7, 18, 10] have extended it; Expersoft, Iona, Sun Systems and Visigenic Software have developed commercial CORBA-compliant event services. The OMG TELECOM SIG has issued a Request For Proposal (RFP) on a notification service which has received several responses [18]. The proposed Notification Service must address issues such as filtering, assured notification delivery, security, QoS, and notification server federation. The Notification Service, however, is based on the indirect event communication model, and does not address implementation issues.

Work in real-time event notification [7, 19] has produced useful designs and implementations which use real-time threads for event publication in order to prevent priority inversion. Performance has been a major emphasis. [7] in particular, also address issues such as event filtering and correlation. However, its filtering and correlation mechanisms are not as powerful as the Cambridge Event Paradigm [2]. Moreover, filters can only be placed at the event channel.

Work in active databases is of direct relevance to the research on active systems [4, 5, 22]. In an active database, events include time events, start- and end-of-transaction events, operation invocation events, abstract events (events signalled from outside the database system). These are monitored and conditions are checked before actions are triggered upon event occurrences. The concept of composite events is introduced in active database for events correlation. Most techniques used for implementation are designed for a centralised database, thus do not address directly issues required for a distributed implementation.

7 Concluding Remarks

We focused on the design of an architectural framework for event handling and showed how COBEA can be used to build large-scale distributed active systems under various application scenarios. We also reported our preliminary experience of using COBEA for building real applications.

COBEA supports the fundamental building blocks for developing active event-driven systems, namely the primitives, the mediator and the composite event service. The primitives form the foundation of the COBEA event architecture, in which the publish-register-notify mode is well supported for efficient asynchronous event communication. The mediator decouples the supplier from the consumer by accepting events from the suppliers, and passing events only to the interested consumers. Thus the supplier and the consumer do not have to know each other in order to communicate events. The composite event service, in particular, provides a powerful means of composing events via a number of operators and a convenient interface for the user to specify composite events. The distributed implementation of the service includes an evaluation engine for composite events, events timestamped at source, event streams and fault tolerance in the form of a heartbeat protocol.

The work focuses on the Notification Model and features well-defined interfaces for event registration, notification and filtering. Future work for COBEA includes incorporating security measures and implementing support for reliable event delivery.

Acknowledgements

This work is funded by EPSRC and Nortel Technology. We gratefully acknowledge the support from Nortel for developing the application scenarios in telecommunications, especially the help of Niall Ross. We would like to thank Richard Hayton, Christof Karl for implementing the components of COBEA.

References

[1] J. Bacon, J. Bates, and D. Halls. Location-oriented multimedia. *IEEE Personal Communications*, 4(5):48–57, October 1997.

[2] J. M. Bacon, J. Bates, R. J. Hayton, and K. Moody. Using events to build distributed applications. In *Proc. of the Second International Workshop on Services in Distributed Networked Environments*, pages 148–155. IEEE Computer Society Press, June 1995.

[3] John Bates and Jean Bacon. Supporting interactive presentation for distributed multimedia applications. *Multimedia Tools and Applications*, 1(1):47–78, March 1995.

[4] A. P. Buchmann et al. Building an integrated active OODBMS: requirements, architecture, and design decisions. In *Proc. of the 11th Intl. Conf. on Data Engineering*, Taipei, March 1995.

[5] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. of the First Intl. Workshop on Rules in Database Systems (RIDS)*, Edinburgh, August/September 1993.

[6] W. Gaver et al. Europarc's RAVE System. In *Proc. of the ACM CHI'92 Conference on Human Factors in Computing Systems*, Monterey, Calif., 1992.

[7] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA event service. In *Proc. of the OOPSLA'97 conference*, pages 184–200, October 1997.

[8] Richard Hayton. *OASIS - An Open Architecture for Secure Interworking Services*. PhD thesis, Computer Laboratory, Cambridge University, June 1996.

[9] A. Hopper, A. Harter, and T. Blackie. The Active Badge System. In *Proc. of ACM INTECHI'93*, 1993.

[10] IONA. OrbixTalk - The White Paper. Technical report, IONA Technology, April 1996.

[11] JavaSoft. JavaBeans^{TM}. Version 1.00-A, December 1996.

[12] G. B. Kendon and N. F. Ross. Alarm correlator prototype: Demonstration script. Nortel Technology Internal Report, February 1996.

[13] Chaoying Ma. An Alarm Correlator Based on the Cambridge Event Technology. Active Systems Project (Cambridge University and Nortel Technology) Internal Report, April 1997.

[14] OMG. Common Object Services Specification, Volume I. OMG Document No. 94-1-1, March 1994.

[15] OMG. The Common Object Request Broker: Architecture and Specification. Revision 2.0, July 1995.

[16] OMG. CORBA Security. OMG Document No. 96-08-03 through 96-08-06, July 1996.

[17] OMG. Messaging Service: RFP. OMG Document No. ORBOS/96-03-16, March 1996.

[18] OMG. Notification Service: RFP. OMG Document No. Telecom/96-11-03, Nov. 1996.

[19] Ragunathan Rajkumar, Mike Gagliardi, and Lui Sha. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *Proc. of the First IEEE Real-Time Technology and Applications Symposium*, May 1995.

[20] N. Ross and K. Burn-Thornton. Design for Saffron Alarm Correlator. Nortel Technology Internal Report, February 1996.

[21] Douglas C. Schmidt and Steve Vinoski. The OMG Event Service. *C++ Report*, 9, Feb 1997.

[22] Scarlet Schwiderski. *Monitoring the behaviour of distributed systems*. PhD thesis, Cambridge University Computer Laboratory, April 1996.

[23] TIBCO. Tib/Rendezvous: White Paper. Technical report, TIBCO Software Inc., 1994.

[24] TINA-C. Engineering modelling concepts (DPE architecture). Technical Report TB_NS.005_2.0_94, TINA-C, December 1994.

[25] J. Warne. Event management for large-scale distributed systems. Technical Report APM.1633.01, APM Ltd., November 1995.