Event Storage and Federation Using ODMG

Jean Bacon, Alexis Hombrecher, Chaoying Ma, Ken Moody, and Walt Yao

Cambridge University Computer Laboratory
New Museum Site, Pembroke Street
Cambridge CB2 3QG, UK
Tel: +44 1223 334600
{Firstname.Lastname}@cl.cam.ac.uk

Abstract. The Cambridge Event Architecture has added events to an object-oriented, distributed programming environment by using a language independent interface definition language to specify and publish event classes. Here we present an extension to CEA using the ODMG standard, which unifies the transmission and storage of events. We extend the existing model with an ODL parser, an event stub generator, a metadata repository and an event library supporting both C++ and Java. The ODMG metadata interface allows clients to interrogate the system at run time to determine the interface specifications for subsequent event registration. This allows new objects to be added to a running system and independently developed components to interwork with minimum prior agreement. Traditional name services and interface traders can be defined more generally using object database schemas. Type hierarchies may be used in schemas. Matching at a higher level in the type hierarchy for different domains is possible even though different specialisations are used in individual domains. Using metadata to describe events provides the basis for establishing contracts between domains. These are used to construct the event translation layer between heterogeneous domains.

1 Introduction

Since the early 1990s we have worked on event-driven distributed applications. The Cambridge Event Architecture (CEA) is compatible with both message passing and object-based middleware. The architecture we describe in this paper is an extension of our previous work. The aim of the extension is to provide an architecture that handles, as well as event transmission, storage and federation of event services. Our previous approaches have tried to tackle event transmission and storage separately [1] [3] [11] [17]. The notion of a federation stems mainly from the database world [16] [15] and has had very little mention in event-based systems. It is our goal to unify these ideas in a single architecture.
We have taken this approach because we have realised that many different application areas exist where events must be stored and/or existing event systems must be linked. For example, a system wishing to make certain guarantees in the face of system failure must store details about event transmission. Events

© Springer-Verlag Berlin Heidelberg 2001

from banking transactions may need to be stored so that they can be analysed at a later time for fraud detection.

In the past, systems that produce a high frequency of events have stored event occurrences in the form of logs. But retrieval of specific records from a log for analysis is highly inefficient. Also, in an environment where high level queries must be processed to discover useful information, events must be typed. Storing them in a database rather than in a log allows a database query language to be used.

Emerging applications often require intercommunication, while still being autonomous. A loosely coupled federation allows for this. In a federation event occurrences need to be communicated between domains. Since our architecture uses typed events, we can reconcile event occurrences from different, but related domains. In a federated event architecture metadata describing event types is used for event translation.

To support these requirements, we have extended the existing Cambridge Event Architecture by defining events using the Object Data Management Group's (ODMG) Object Definition Language (ODL). For transmission of events we have previously used OMG's IDL as a means of describing events. Using ODL allows us to keep metadata about events, and also store events in an ODMG compliant Object-Oriented Database (OODB) with powerful query support from OQL.

The implications for distributed system design of this generality are:

– Application data can be defined in a uniform way for transmission and storage.
– Objects can use the ODMG metadata interface to allow clients to interrogate them at run time to determine their interface specifications.
– Traditional name services and interface traders can be defined more generally using a metadata repository. The ability to use type hierarchies within schemas allows, for example, agreement across domains at a high level with different specialisations within domains.
– Standard database techniques can be used for retrieving event patterns of interest from event stores.

In the rest of this paper we will describe our new architecture. We will focus specifically on event federation as we have described the details of the transmission and storage of events elsewhere. Section 2 explains the motivation and application of our architecture. We will then give some background information to familiarise the reader with the basic terminology and existing research in this area. Following that, we describe our architecture in more detail. In section 5 we describe an instance of a multi-domain event architecture.

2 Motivation and Applications

Our architecture has been motivated by different factors. We feel the world will not restrict itself to using a single programming language or middleware platform. It is essential to support inter-working between heterogeneous systems

for wide and long-term impact. Furthermore, persistent data management is a fundamental aspect of system design but a universal persistent programming language is not the answer. It is essential to link components that have been developed in different languages. Our view is that an object model is a realistic basis for software system design including persistent data management. The concepts of types, interfaces and interface references are well established with trader technology for name to location mapping. More novel is that encapsulation of object attributes and methods may be extended to include object management aspects such as concurrency control and other transaction support. A standardisation of data types, such as the Object Data Management Group's ODL [4], forms a minimal basis for interoperability between programming languages and database query languages.
We will begin by discussing applications where integration of distributed entities is done via events. We will then look at applications requiring event storage. Finally, we will discuss environments in which existing notification-based event systems may need to be integrated into a loosely-coupled federation.

2.1 Event-Based Integration

Event-based integration is possibly the most common loose integration approach. Often it is described as a *publish-subscribe* model. The components generating events are *sources*, while the consumers of the events are *sinks*. Sources publish the events they generate, while sinks register interest with particular sources, for certain event types. Upon the occurrence of an event, interested sinks are notified.
Four broad categories of application, which have the requirement for event-driven notification, are described in [18]. This list is not exhaustive.

- composing and building large scalable distributed systems; enterprise application integration; concurrent/parallel programming languages; modelling of information flow; distributed debugging and fault-tolerant distributed systems
- modelling and supporting work practices relating to communication between individuals
- building computer supported cooperative applications
- applications based on monitoring and measurement-taking; real-time applications; active databases; windowing systems and graphical user interfaces

2.2 Event Storage

In the past, events have been mainly transient entities, acting as a glue between distributed components. Increasingly, applications require that events can be made persistent. [18] defines generic application scenarios where event storage is required. These include:

- *Auditing* of event generation, transformation and consumption
- *Identification* of repeating events
- *Analysis* of event registration, notification, and consumption
- *Tracking* of repeating patterns of activity covering one or more event types
- *Checkpointing* of execution for the aid of constructing fault-tolerant distributed systems

Checkpointing is probably the most obvious use for event storage. In an environment where the event system makes an *exactly once* or *at least once* delivery guarantee, events must be stored in case of network or system failure.

Auditing and analysis of event data is useful in applications such as fraud detection, simulation environments and workflow. A bank may want to store all events generated from bank transactions at automated teller machines. This data can then be analysed to discover inconsistencies such as money having been taken out of the same account at different locations at *nearly* the same time. In a workflow environment stored event data can be analysed to determine how work between individuals is being coordinated. This may lead to possible changes, making processes more efficient.

2.3 Event Federation

The term federation denotes a coupling of existing systems. Different degrees of coupling exist, ranging from loosely to tightly coupled federations. The degree of coupling is directly related to the degree of autonomy each member of the federation retains. In a loosely coupled federation, each member retains a greater amount of autonomy, and vice versa.

An example where event federation may be needed is a tracking system, monitoring the whereabouts of objects within a domain. Many different types of tracking system exist. Current systems include electronic badge-based tracking systems [20] [8], login-based systems, or Global Positioning Systems (GPS). Unfortunately, they are all closed domain systems, meaning they can only track individuals within their domain. The electronic badge-based system, for example, can only monitor the location of people within its domain, where hardware for badge location is available. Once an individual leaves the domain, the tracking system can no longer locate the electronic badge. For certain application domains, this type of tracking is sufficient, but we envision a world where different application domains can communicate with each other.

In a global event architecture, a client can register with the tracking service in any given domain. The client is notified according to the local event definition, when events occur at the source, either in the local domain or in a foreign domain. A user can then be tracked world wide (everywhere tracking system domains are installed and can communicate with each other), with registration for tracking events having occurred only in the local domain. Global registration is handled automatically by the system, even though event types and technology may differ.

3 Event Middleware

Distributed middleware can be grouped into different categories. These are *client / server*, *distributed object framework* and *event-driven systems including message oriented middleware.* In many instances there is an overlap between the characteristics of these types of systems.
It is outside the scope of this paper to analyse all existing distributed middleware. Although our architecture is inherently object-oriented through our use of ODMG, we feel that the event paradigm is dominant and will therefore focus on event-driven systems and message-oriented middleware (MOM) in our discussion.

3.1 Event-Driven Systems

Event-driven systems have originated from internal system needs, such as database triggers and graphical windowing. The emphasis is on event sources and clients. Clients register interest with sources, which then notify clients upon the occurrence of the specific event instance. This is somewhat similar to the *publish/subscribe* paradigm used in MOM systems, but event architectures tend to go further by providing a rich type system, template registration, event brokering, and possibly composite event specification.

The Cambridge Event Architecture. CEA uses a *publish-register-notify* paradigm (see Figure 1). Services publish their interfaces, including the typed events they are willing to notify. Clients register interest in events, specifying parameter values and wildcards. Services then notify any events that match their stored registration templates, thus employing source-side filtering. The architecture includes event mediators and composite event services. A standard, language-independent interface definition language (IDL) is used to specify and publish event classes. This allows automatic stub generation to be used for event notification as well as for traditional object invocation. This facility has been implemented in CORBA environments and in a purely message-passing system. For a detailed description of the generic distributed event architecture see [2].

CORBA Event and Notification Service. The CORBA Event Service [12] specification is based on an indirect channel-based event transport within distributed object frameworks. An *EventChannel* interface is defined that decouples the *suppliers* and *consumers* of events. Suppliers generate the events, while consumers obtain events from the channel. The channels are typed, meaning that only a specific type of event can be placed onto it. Events are not objects or entities in themselves. The specification provides for different types of interaction between the *suppliers* and *consumers*, mainly *push* and *pull* interaction. The *push interaction* is *supplier* driven, while the *pull interaction* is *consumer* driven. The CORBA Event Service does not support typed or structured events.
The CORBA Notification Service [14] tries to address the shortcomings of the CORBA Event Service. It supports typed events, content-based filtering and a

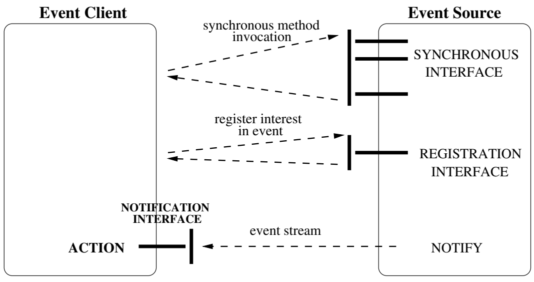

Fig. 1. The Cambridge Event Paradigm

quality-of-service (QoS) interface. Here, an event is similar to an object, having a type defined as a *domain/type/name* triple. The user specified body of an event has two sections. The first allows for the specification of filterable fields, while the second contains additional parameters as defined by the user. Channels support filtering, usually before events come to the consumer. Hence, only events matching the parameters specified in the filter are received by the consumer. Finally, the CORBA Notification Service includes an *Event Type Repository*, which allows for dynamic evolution of events and dynamic adaptation of consumers.

CORBA Persistent State Service. The CORBA Persistent State Service (PSS) introduces PSDL (Persistent State Definition Language), a superset of OMG IDL, that address the problems of persistence [13]. PSDL extends IDL with new constructs to define storage objects and their homes. Like Java and ODMG ODL, it makes distinctions between abstract type specification and concrete type implementation. PSS provides an abstraction layer between a CORBA server and its background storage. This allows a variety of storage architectures to be plugged in and accessed through a set of CORBA-friendly interfaces.

Despite these recent extensions, CORBA does not provide as rich a set of constructors and facilities for data management as the ODMG does. For example, PSS only supports a **find_by_pid** method for access to objects, while ODMG supports OQL. Furthermore, PSS has yet to specify recovery which is well supported by the ODMG standard.

3.2 MOM Systems

MOM systems are the mainstream commercial equivalents of event-based systems. They originated from the need to connect application programs reliably to central database servers. They employ an asynchronous peer-to-peer invocation model, whereby a message is sent to the local middleware before it is sent to its destination. Generally, MOM systems do not have a type system for structuring message data.

IBM MQSeries. MQSeries [6] is not only one of the earliest middleware products commercially available, it is also the most widely used. Its feature set is largely representative of the available solutions.
MQSeries applications place messages on message queues. The receiving queue notifies the application when it is ready to accept a message from the queue. The *Message Queue Interface* is used by applications to manipulate the queues. MQSeries is responsible for moving messages around the network and for handling the relationship between applications and queues. Furthermore, it provides transaction guarantees, either through its built in transaction processing monitor or by an external X/Open compliant TP monitor.
Extensions are available that sit on top of MQSeries, such as IBM Message Broker and MQSeries Integrator. The broker provides *publish/subscribe* facilities, while the latter provides for a limited monitoring of composite events.
MQSeries provides persistence only in so far as it stores events until they have been successfully received by the correct application. Proper provision for storage outside this transactional support is not provided.

Oracle8i AQ. Oracle8i Advanced Queuing [7] is interesting because this architecture is focused around storing messages. Advanced Queues are represented as relational tables and a message is simply a row in the table. The Advanced Queues are an integral part of the database management system. Multiple applications access the database to transfer messages between them, or database systems communicate to move messages between their queues.
Since queues are represented as tables, they can be queried using SQL. This is a very different approach from the usual logging functionality provided by MOM systems.

3.3 Analysis of Existing Systems

None of the systems reviewed provide the functionality required for the scenarios outlined in section 2. Although some systems have support for message storage, these tend to be highly specialised solutions, lacking general applicability. Furthermore, most systems lack the required type system, necessary for event federation. These shortcomings are addressed in our architecture.

4 The Extended Cambridge Event Architecture

We make minimal assumptions on how our world is constructed. In our view, it consists of a set of cooperative domains, possibly federations. A *domain* is a logical scope of a collection of event sources where a single administrative power is exercised. The event sources in a domain are sometimes inter-related, but their relationship is not a requirement for membership of a domain. A domain represents a unit of autonomy. The domain administrator has full control over the event sources and types defined by each source. Domains dramatically reduce the degree of complexity of a distributed system by partitioning it into smaller, more manageable units and allow independent evolution.

Components in our architecture may be *producers* or *consumers* of events, or both. Producers publish their event types, while consumers register interest with producers and receive event occurrences of interest to them. This is no different from other *publish-subscribe* or *publish-register-notify* architectures. Novel is that we view components that store or translate events as consumers. This greatly simplifies the complexity of the system.
We have based our architecture on the Object Data Management Group's standard for object-oriented databases. ODMG has released three standards since the early 1990s, the most recent being ODMG 3.0 [4]. The core of the standard is the ODMG Object Model, which defines a rich type system compatible with major object-oriented programming languages, including the arbitrary nesting of collection types. It defines the Object Query Language (OQL), a powerful superset of SQL. Bindings have been defined for C++, Java and Smalltalk. Types can be specified through the Object Definition Language (ODL) which serves as the data definition language for database schemas. ODMG includes a standard for schema representation by meta-objects, which themselves conform to the object model, thus allowing standard database tools to interrogate metadata as well as data.

4.1 Events

In the extended CEA, events are defined using ODL. Every event is derived from a non-instantiable base type called *BaseEvent*. The *BaseEvent* type is the header of each event occurrence. Amongst other things, it contains a unique ID (consisting of a unique object ID and a source definition for the object), a priority tag and a timestamp.

```
class BaseEvent {
  attribute ID id;
  attribute short priority;
  attribute timestamp signal_time;
};
```

One can construct complex event hierarchies through the use of inheritance (see Figure 2). Grouping related events within a hierarchy can be useful when reconciling different event taxonomies (see Section 4.5).
The event definitions are compiled by an ODL parser and stub generator which translates ODL files into stubs generated as wrappers around event types. The stubs present the application programmers with an intuitive programming interface. *EventSource* and *EventSink* classes have been implemented to allow for a simple connection to the library and to create the service objects.
The example below shows the program code written using the generated stub. In the example we present a simple event, `GPSEvent`, which is derived from a base event of type `LocationEvent`. We show both the client-side and server-side code. The client registers with the source for events that signal GPS sightings of any observable subject at 100 degrees longitude.
The ODL definition for the event of type `GPSEvent` is defined as follows.

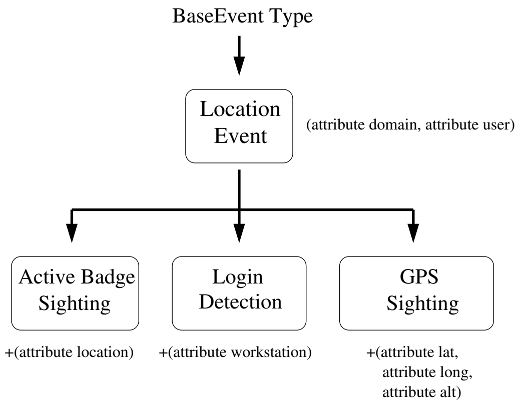

Fig. 2. Event Inheritance

```
class LocationEvent extends BaseEvent {
  attribute string user_name;
};

class GPSEvent extends LocationEvent {
  attribute float longitude;
  attribute float latitude;
  attribute float altitude;
};
```

When registering interest with an event source the client must first initialise an event sink to establish a channel with the source. In the example below, the client constructs a registration template for `GPSEvent`. Wildcard filtering is the default which, in this example, is deactivated on the parameter “longitude”. This allows the client to receive all the event occurrences matching longitude 100. The client can then register with the event source by invoking the method `register_event_interest` with the prepared template.

```
EventSink sink(argc, argv, "sink", "c1@cl",
               "source", "c2@cl", disconnect);

//construct an event template
templ.longitude = 100;

//construct an event template
GPSEvent_Sink templ;

//set value for parametrised filtering
templ.longitude = 100;

//activate parametrised filtering
```

```
temp.set_longitude_wildcard(false);
...
templ.register_event_interest(sink, gps_callback);
...
```

Classes for event templates are generated by invoking the stub generator. The class name for an event type is the name of the event type as defined in ODL, suffixed by `_Sink`. In this example, the client constructs an event template for `GPSEvent`. Parametrised filtering in a template is deactivated by default. This means that the client will be notified upon the occurrence of all events of the type handled by the template. The client can set parameters of interest by explicitly setting the attributes and then activating the filter. Once a template is created, it can register with an event source by invoking the API `register_event_interest` and supplying the sink object created earlier.
On the server-side, the programming is similar, except for an additional step of activating the source. The source application creates a source by initialising a source object. This binds the source application under the specified name in the domain-wide name service. In order to be able to produce events of a certain type, the type must be made known to the source object. The API method `register_new_event` is provided for this purpose. Once the event type is known to the source, it can signal events at any time.

4.2 Event Schemas

An event schema is maintained for each event type in a domain. Per domain, many event schemas may exist. As the ODL files are compiled, the metadata is stored in an event repository. We have opted to treat this event repository as a generic event store. Ideally, the repository is ODMG compliant, but this is not a requirement. Mediators must be used to translate from the ODL representation to the internal representation of the store if it is not ODMG compliant. If it is, the ODL compiler of the store can be used and the event metadata is stored as such in the store. This is very useful when the same store is to be used to store event occurrences.

4.3 Naming and Event Brokering

There are a number of options for interface and event publication: an interface trader can be used for the entire service signature including events; event signatures may be published in a separate name service; each service may provide an introspection, or metadata, interface to allow clients to interrogate it at run time to determine its interface specification. The latter approach is the easiest way to allow new services to be added to a running system.
Each domain has a name service, which stores the run-time binding for an event type and an event engine. This information is required by clients so that event sources which are associated with event engines can be located dynamically at runtime. For example, a client wishing to register interest in a particular kind of event must discover whether that event type exists in the domain. To do this,

it uses the metadata interface to query the ODMG store. After discovering the appropriate event type, the naming service must be queried to locate the event server. Besides providing a name-to-location mapping, the naming service also provides the client with other information required for registration.

4.4 Event Storage

Event storage can occur at two levels. Firstly, it can be embedded within the event source or client. Secondly, dedicated storage nodes can be used, providing an explicit service. The first is useful for services like buffering at the source or client, while the second is useful for analysis of event histories.
In the latter case, we view event stores as consumers of events. A storage component registers interest with a source when it plans to store event occurrences. Where the object model of the event system is different from the object model of the event store, mediator applications are used to translate between the two. This means that event stores do not directly register interest with the event sources, but rather the mediator handles this task on behalf of the storage system. In instances where events are stored in an ODMG compliant store, this transformation is not necessary, since the object model used at the event source is based on the ODMG standard.
Ideally, the event repository at dedicated nodes of our architecture is an ODMG compliant store. We are currently investigating the use of a commercial product for this. This would provide us with the functionality of a high level query language for event analysis. We are also testing the limitations of such a system in terms of event stream throughput. In systems where thousands of event occurrences need to be stored, special log-like mediators [18] may need to be used before events are actually transferred into the store.
For embedded storage facilities, we have built a lightweight ODMG compliant store. This implementation is used for embedded repositories in either the event consumer or producer. It consists of four major components, a *storage manager*, a *transaction manager*, an ODMG *object layer* and a *parser generator*. These provide the functionality required for storing and retrieving event metadata and event instances.
When constructing event producers or consumers, the application programmer can specify whether event occurrences are to be stored. We have implemented our storage layer as a library that can be embedded within the applications.

4.5 Event Federation

Federating event systems involves schema federation [16] and object translation. Specifically, it involves reconciling semantic differences in schemas for the purpose of constructing a federation. Since the event architecture is based on the ODMG standard, we are limiting the problem to having to reconcile only semantic heterogeneity [9], or more precisely, structural semantic heterogeneity [5].
Structural semantic heterogeneity occurs when the same information occurs in two different domains in structurally different but formally equivalent ways. Different examples of this are:

- the same entity with different attribute names
- the same instance identified differently
- the same conceptual model represented structurally different
- the same universe of discourse represented by different conceptual models
- the same instances aggregated incommensurably in two systems

Structural semantic heterogeneity can be resolved by a series of view definitions which in principle respect the autonomy of the component systems. There are cases in which semantic heterogeneity cannot be resolved by view definitions, even in principle. This more general semantic heterogeneity is referred to as fundamental semantic heterogeneity. Where it occurs between information systems, tight coupling cannot be achieved without changing at least one of the systems, thereby compromising design autonomy.
Fundamental heterogeneity occurs when objects in the two domains under consideration share insufficient attributes to be reliably identified. The metadata in the two domains does not contain sufficient information to determine whether object instances relate to the same entity. This phenomenon has been studied by a number of authors, and has been given several different names. [19] have termed it the instance identification problem, [10] the entity identification problem.
In our architecture we have focused on resolving structural semantic heterogeneity.

Gateways and Contracts. In order to federate event services from different domains, components are needed, which handle the event translation between domains. These components act as a gateway service and must be available within each domain wishing to participate in the federation.
In our architecture, gateway communication is based on contracts. The notion of a contract is derived from real-life examples of how entities interact with each other. Conceptually, a contract is a binding agreement between two or more parties. We employ this idea and apply it to our model. We use the term *contract* to mean an agreement between two or more cooperating domains. From a database federation point of view, a contract is similar to the external schema proposed in the five-level schema architecture put forward by Seth and Larson [16]. A more formal definition of a contract is given below:

> Let T_A be an event type defined by a source S_A of domain A, and T_B be an event type defined by a source S_B of domain B. We can define sets E_A and E_B as the sets of all instances of T_A and T_B respectively. A contract between A and B with respect to T_A and T_B is defined as an event type $\mathcal{T}$, whose set of all instances is denoted as $\mathcal{E}$, and which satisfies the condition that there exists a function $f : E_A \rightarrow \mathcal{E}$ for A and $g : E_B \rightarrow \mathcal{E}$ for B.

The function that maps a local event to an event of the type defined by a contract is called a *schema translation function.* A schema translation function that translates an event from type T to type S is a translation function from T to S. Schema translation takes place at the intersection of two domains, the

gateways. An event which is going to be sent across domains will be translated within its domain of origin. This means the foreign domain always gets events conforming to a contract it has agreed to. One advantage of the elegance and simplicity of this model is that applications are easier to build. Applications only need to handle the types that they are familiar with, so that dynamic handling of foreign types is unnecessary. Another advantage is that this model effectively prevents the details of an event local to a domain being exposed to the foreign domain. This is desirable if an event carries confidential information, such as a credit card number in an `orderMade` event.
Another advantage is that translation only arises at the gateway for events to be exported, with respect to a specific contract. This is entirely demand driven.

5 A Federated E-commerce Example

In this paper we have opted to use an example which focuses on the ability of our architecture to link semantically heterogeneous systems. We leave out event storage here, but the reader should easily be able to see how event stores can be added to the system.
With the recent excitement over dotcoms, we will illustrate the use of our architecture by looking at online travel agents specialising in last minute flights and holidays. Specifically, we will look at two online services, *LastHour.com* and *FirstMinute.com*. Each dotcom represents a domain and allows clients to register interest with event sources available within the domain. For example, customers may want to be notified of last minute travel offers to specific destinations at a specific price. We will illustrate how multiple dotcoms can be joined in a federation to provide a global, highly competitive, holiday shopping environment.
We will first explain event registration, followed a by a more detailed look at event federation.

5.1 Event Registration

As mentioned before, the ODL for the event type definitions will be compiled into header and body files, which implement the event class and interface with the existing event source and sink object implementations. A sample client of *LastHour.com* wishing to register interest in flights with *United Airlines* at that particular price would look as follows (in pseudo C++):

```
Flights fl;
fl.price = 1500;
fl.airline = "United Airlines";

Callback* clb = new Callback(orb, ckfn,
                             sink_host, sink_server,
                             source_host, source_server);

Sink_i *myobj = new Sink_i(clb);
myobj->_obj_is_ready(boa);
```

```
string sink_name = si_server + "@" si_host;
const char* sn = sink_name.c_str();

BaseEventServer::Snk_var myobjRef = myobj->_this();

if(!bindObjectToName(orb, myobjRef, sn))
   ...
boa->impl_is_ready(0,1);
position.register(clb);
```

In the example, the client stub for the *Flights* class has implemented a method *Flight::register(Callback* clb)*. This method constructs the event body expected by the *Src::reg_event* operation, and then calls the *Callback::register_interesT* method. This in turn calls the *Src::reg_event* operation. At the time the callback object is created, the client binds to a *Src* object and stores the object reference in the callback object. Thus an invocation to the event registration method *Src::reg_event* can be made at a later time via the method *fl.register(clb)*.

5.2 Event Federation

Each domain has event types for which clients can register interest. These event types are part of an event hierarchy. Figure 3(a) and 3(b) show the event hierarchies for two domains in our example.

(a) LastHour.com (b) FirstMinute.com (c) A Contract

Fig. 3. Event Hierarchies

A contract between *LastHour.com* and *FirstMinute.com* must exist for specific event types, if the two domains wish to form a federation. The contract specifies that each domain must be able to translate to and from the type defined in the contract. Thus, translation functions must be available at the gateway level that honour the contract.

For the purpose of this example, we will consider a client that wants to register interest in an event locally and globally. The event of interest is the *Flights* event from the *FirstHour.com* domain. The client may be interested in a last minute

flight at a price less than $1000 to Sydney with no more than 1 stop-over. The client does not care about the airline.
The client can register interest in the above event by sending the appropriate registration to the event engine for local registration. Alternatively, the client may wish to register interest not only with *LastHour.com*, but also with *First-Minute.com*, or all domains having a contract with LastHour.com for that event type. The registration will be different only in that it must indicate the other domain(s) of interest.
Cross-domain registration has to translate the local registration into a format understood in other domains, in the same way that event types have to be translated between domains. This translation is performed according to the contracts between domains. The translation takes place when the registration leaves the local domain (*LastHour.com*) and when the converted registration enters the foreign domain (*FirstMinute.com*). Upon leaving the local domain, the registration is converted to the contract type and upon entering the foreign domain it is converted into a format understood by that domain. Once translated, the registration is sent to the correct event engine in the foreign domain for the actual event registration.
Event notification in the local domain, *LastHour.com* is trivial. If a flight to Sydney at a price of less than $1000 with no more than 1 stop-over is available, the client is contacted asynchronously by the event engine. On the other hand, when an event of interest occurs in *FirstMinute.com*, a flight to Sydney at a price of less than $1000 (remember the attribute stop-over does not exist in this domain), the appropriate notification is sent from the event engine to the gateway. Here the notification is converted to the contract type. It can then be sent to the gateway of *LastHour.com.* At the gateway of *LastHour.com* the notification gets translated into an event of the local domain. In this instance, the attribute for stop-overs must be given a default value, or left blank (the ODMG *Object Model* supports this through *null-extended* domains). The notification is passed to the event engine, which then notifies all registered clients.

6 Conclusion

We have outlined the functionality and shortcomings of current middleware products with respect to their support for event-driven applications. We argued that such applications require storage and query facilities for events as well as efficient, asynchronous transmission. Also, such applications need to interoperate in an open distributed environment. We have extended our event architecture, CEA, to meet these requirements.
CEA events are specified in ODL, compatible with IDL, and benefit from standard middleware technology such as automatic stub generation. Clients may register interest in events at their source, with specific parameters as well as wildcards, allowing efficient filtering at source before immediate asynchronous notification of occurrences.
Storage of events may be incorporated into the architecture either by embedding persistent storage locally at sources, sinks and mediators, or as separate event

stores. The latter may be integrated simply as event clients using the publish, register, notify paradigm.
Using ODMG's ODL and OQL has given us the potential for great flexibility in designing the components of our architecture. All objects have a metadata interface which may be used to interrogate them to determine their schemas; this provides a good way to add new objects to a running system. ODMG data stores may be queried, using OQL, for event patterns of interest in fault or fraud detection.
In this paper we have focussed on the open interoperability of event systems. The use of a type hierarchy for event definition, together with the *publish-register-notify* paradigm, allows for registration at a high level to be specialised in different ways in different domains. We have described our use of gateway servers and contracts; registration is always in the local domain and a local gateway server negotiates contracts with event engines in relevant non-local domains on demand. Parameter values that are not required do not leave the domain in which they occur; this may be important in restricting information flow.
Further work is required in several areas. We must explore the ability of ODMG stores to log large volumes of events for subsequent querying using OQL. Reconciliation of type hierarchies for contract negotiation is a difficult problem in general. We are exploring the role of XML as an alternative interchange format and have built an ODL parser which generates XML.
Although this work is at an early stage we have built demonstrations as proof of concept. We believe that the approach we have taken is more generic than that of existing systems and provides an excellent basis for federation of existing systems as well as incremental evolution.

Acknowledgements. We acknowledge the support of UK EPSRC through the grant GR/M22413 (Global Computation using Events). We would also like to thank past and present members of the OPERA group for their insights and helpful discussions.

References

1. J. Bacon, J. Bates, R. Hayton, and K. Moody. Using Events to Build Distributed Applications. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.
2. J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic Support for Asynchronous, Secure Distributed Applications. *IEEE Computer*, pages 68–76, March 2000.
3. J. Bates, D. Halls, and J. Bacon. A Framework to Support Mobile Users of Multimedia Applications. In *ACM Mobile Networks and Nomadic Applications (NOMAD)*, pages 409–419, 1996.
4. R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D.Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann Publishers, San Diego (CA), USA, 1999.
5. R. M. Colomb. Impact of Semantic Heterogeneity on Federating Databases. *The Computer Journal*, 40(5), 1997.

6. IBM Corporation. MQSeries. http://www.ibm.com/software/mqseries/, 1999.
7. Oracle Corporation. Oracle8i Advanced Queuing. http://www.oracle.com/database/features, 1999.
8. A. Harter and A. Hopper. A Distributed Location System for the Active Office. *IEEE Network*, 8(1), January/February 1994.
9. R. Hull. Managing semantic heterogeneity in databases: a theoretical prospective. In ACM, editor, *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12–14, 1997, Tucson, Arizona*, pages 51–61, New York, NY 10036, USA, 1997. ACM Press.
10. E. P. Lim, J. Srivastava, S. Prabhakar, and J. Richardson. Entity Identification in Database Integration. In *International Conference on Data Engineering*, pages 294–301, Los Alamitos, CA, USA, April 1993. IEEE Computer Society Press.
11. C. Ma and J. Bacon. COBEA: A CORBA-based Event Architecture. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS-98)*, pages 117–132, Berkeley, April 1998. USENIX Association.
12. Object Management Group - OMG. Event Service Specification. ftp://www.om.org/pub/docs, 1997.
13. Object Management Group - OMG. CORBA Persistent State Service 2.0. 99-07-07, August 1999.
14. Object Management Group - OMG. *Notification Service Specification*, June 2000.
15. E. Radeke. Extending ODMG for federated database systems. In Roland R. Wagner and Helmut Thoma, editors, *Seventh International Workshop on Database and Expert Systems Applications, DEXA '96, Proceedings*, pages 304–312, Zurich, Switzerland, September 1996. IEEE Computer Society Press, Los Alamitos, California.
16. A. P. Sheth. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. In *International Conference On Very Large Data Bases (VLDB '91)*, pages 489–490, Hove, East Sussex, UK, September 1991. Morgan Kaufmann Publishers, Inc.
17. M. Spiteri and J. Bates. An Architecture to support Storage and Retrieval of Events. In *Proceedings of MIDDLEWARE 1998, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 443–459, Lake District, UK, September 1998.
18. M. D. Spiteri. *An Architecture for the Notification, Storage and Retrieval of Events (TR494)*. PhD Thesis, University of Cambridge Computer Laboratory, Computer Laboratory, New Museum Site, Pembroke Street, Cambridge CB2 3QG, England, July 2000.
19. Y. R. Wang and S. E. Madnick. The Inter-Database Instance Identification Problem in Integrating Autonomous Systems. In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 46, Los Angeles, CA, February 1989.
20. R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.