Events in an RPC Based Distributed System

Jim Waldo, Ann Wollrath, Geoff Wyant, and Samuel C. Kendall
Sun Microsystems Laboratories

Abstract

We show how to build a distributed system allowing objects to register interest in and receive notifications of events in other objects. The system is built on top of a pair of interfaces that are interesting only in their extreme simplicity. We then present a simple and efficient implementation of these interfaces.

We then show how more complex functionality can be introduced to the system by adding third-party services. These services can be added without changing the simple interfaces, and without changing the objects in the system that do not need the functionality of those services.

Finally, we note a number of open issues that remain, and attempt to draw some conclusions based on the work.

Introduction

Distributed systems are generally built using one of two distinct communication techniques. The first and most common of these is the distributed analogue of the procedure call, generally called remote procedure call (RPC). The other, less common communication base is the notification from one entity to the other of the occurrence of an event.

Each of these communication techniques has its proponents, and each is appropriate for particular kinds of distributed applications. In this paper, we will discuss how one can build a simple event notification system on top of a remote procedure call system. The distinguishing characteristic of the system described is the simplicity of the underlying protocol for simple event notifications. More complex kinds of event notifications are constructed by the introduction of third-party objects that can be interposed between supporters of the basic protocol to provide advanced functionality.

RPC based distributed systems

Distributed systems based on Remote Procedure Calls have been around for a considerable period of time [3]. The basic approach is still being used for the construction of distributed systems [15][8][13]. Extensions of the approach introduce remote method invocation on objects [12][5], support for fine-grained objects [9], and the automatic location and activation of objects [11].

Throughout these extensions, the mechanisms of the approach have remained essentially unchanged. A call to a remote entity is routed through a surrogate in the address space of the caller. This surrogate is responsible for marshalling the parameters of the procedure (method, function) into a form that can be sent across the wire, and transmitting the result of the marshalling across the wire. On the receiving or server end, a skeleton receives the transmission, unmarshals the result into a form that can be understood by the recipient, and then invokes the local function with the appropriate arguments. Any return values are marshalled by the server, sent across the wire, and unmarshalled by the client surrogate.

This sort of structure lends itself to programming aids. Interface definition languages can be used to define the form of calls from client to server, and compilers for such languages can produce much, or all, of the marshalling and unmarshalling code. The reliance on interfaces to define the communication paths lends this technique to description in terms of objects, and many of the languages used to define these interfaces support a notion of inheritance.

The programming model for an RPC based system is familiar to most programmers. A call to a remote object transfers control to that object, and the calling object blocks until receiving the return of the RPC. With the introduction of threads into the client, only a single thread needs to block. Some systems allow genuine asynchronous calls, but in general the programming model is (by design) as much as possible like that of procedural non-distributed systems.

This leads to a model of the client controlling the interaction, and the server providing some service to that client. Servers in this model are essentially passive entities, waiting for some client to request of them that they do their thing.

The sort of application that fits into this model includes such things as distributed compound documents, in which the various parts of the document are separate objects. The user interface to such a document will call each of the objects, asking it to display itself on the appropriate device when the user moves to that part of the document. The objects themselves react to direct calls from the controlling interface object, which is in turn controlled by some user.

Event based models

A less common model for distributed systems is based on the communication model of event notification. Such systems were pioneered by Isis [2], but other exemplars of this approach to distribution include Teknekron [14], Zephyr [6], and InterStage [7].

The model for such systems is that some significant changes in computational entities making up the system are identified as events. An event might be the change in the price of a stock, or the creation of a new file, or editing the value of a cell in a spreadsheet. Classes of such events are identified as kinds of events that may be significant to others. These other entities can register interest in the occurrence of any event of a particular event kind. If an entity has received such an interest registration, any time the event of that kind occurs it is obliged to send a notification to those entities that have registered interest in that kind of event.

On the surface, such systems allow a much lighter-weight communication mechanism for the distributed system. Notifications need not be synchronous, and indeed most such systems are asynchronous in nature. Complications arise when the ordering of the events from different objects must be taken into account; considerable work has been done to deal with these orderings [1].

The event paradigm seems somewhat foreign to the object-oriented approach to software. While the computing entities in such a system could be considered objects, the exportation of notifications when the state of one of those objects changes appears to violate the abstraction boundary of the object. The notion of a kind of event appears to show part of the state of the object, which violates the object-based paradigm.

Where the RPC model of distributed computing appears to promote a style of passive objects waiting for some client to request a service, the event notification based systems promotes a style in which objects react to the occurrence of events in their own way. This allows new objects to be introduced to the system that react to events in new ways without changing existing objects.

The style of programming in an event notification based system is similar to that currently popular among developers of user-interface software. Rather than designing the software in a procedural fashion, objects are designed to react to input events from a user.

The sorts of applications that fit well into this model include such things as distributed workflow systems. In such a system, a change in one object (say, the adding of an order to a database) will trigger a notification to a number of other objects (those having to do with inventory maintenance, manufacturing scheduling, or credit checking). Each of the objects that receives the notification will take an action that is appropriate for the object; what action is taken is not known by the object in which the triggering event occurred.

Events in an RPC based distributed object system

Allowing objects to react to changes in the other parts of the system is a useful programming paradigm. To introduce the functionality of an event notification system into a distributed object system based on RPC requires that we take seriously at least the following goals:

- Events must be introduced in such a way that they do not violate the abstraction boundaries of the objects in the system. In particular, the introduction of events should not change the ability of an object to be totally characterized (from the outside) by the set of interfaces that the object supports;

- If possible, introduction of events should not introduce a universal namespace. Distributed systems, especially those that are large and developed by multiple programmers in multiple organizations, should not require that all developers adhere to the same naming conventions;
- The basic service should be cheap both in terms of implementation effort and runtime efficiency;
- Complex features should be built on a simple base;
- If different levels of service are used to introduce more complex features, those levels should be completely transparent to those not directly using the service;
- Events should not be introduced as an alternative to RPC, but should only be used to allow functionality that does not fit well into the RPC programming paradigm.

To meet these goals, we have developed a system built around the following basic concepts:

- Identification of kinds of events
- Registration of interest in a kind of event happening in some object
- Notification of the occurrence of an event.

We will look at each of these in turn.

We should note that the only object-specific aspect of what follows is the restriction against violating the encapsulation boundaries of the computational entities involved. The same techniques can be used for distributed systems based on non-object programming approaches.

Event Identification

Our system uses fixed-length identifiers for types or classes of events. These identifiers are obtained from individual objects by calling methods defined for that purpose.

Such methods are part of interfaces defined in the usual way[1]. For example, suppose an object supports the exportation of an identifier for an event class that corresponds to changing the name of the object. The method that exports this identifier would most naturally occur in the interface that includes the method called to change the name of the object.

The identifiers used to name an event class are not required to be the same from object to object. All that the system requires is that any particular object issue a single identifier for any particular kind of event and different identifiers for different kinds of events. However, it is possible that the "same" kind of event in different objects will be denoted by different event identifiers, or that identical event identifiers are used by different objects to denote events types that are the "same". We can, however, characterize what it means to say that two event identifiers denote the same kind of event: identifier A and identifier B denote the same kind of event if and only if A and B are returned from calls to the same method (perhaps invoked on different objects).

Making event-kind identifiers relative to the particular object that exports the event class avoids the problem of introducing yet another universal namespace into the infrastructure for distributed systems. Such namespaces can be problematic in a distributed system that has no central authority to ensure that the same name (identifier) is not given to multiple entities. Individual objects can be trusted to make sure that they do not export the same event class identifier for different event classes, and thus act as a local authority on event class identifiers.

Registering interest in an event class

Objects that export interfaces that include methods returning event class identifiers do not, just by that exportation, allow other objects to register interest in those event classes. To do that, the object must export the Event Generator interface.

A short comment on notation. We use the OMG CORBA Interface Definition Language [11] to define our interfaces. This language allows multiple inheritance of interfaces (although we have chosen not to use that feature). We have also adopted the convention of naming the primary interface in any module **T** (following a convention of Modula-3, our implementation language).

The interface that allows registration of interest in the event classes exported by some object is shown in Figure 1.

Some explanation is in order. The interface refers to three other interface definition files. The interface described in the file "VantageObj.idl" is

1. In our case, the usual way is to use the OMG CORBA Interface Definition Language (IDL). We will say more about this later.

one supported by all objects in the overall system we are constructing. This interface contains a single method that returns an identifier that (with a high degree of probability) uniquely identifies the object. The interface defined in the file “EventID.idl” describes event class identifiers of the sort discussed in the previous section. The file “EventCatcher.idl” describes the interface that is used to deliver notifications, and will be discussed more fully in the next section.

The interface **EventGen::T** describes a type of object that is a subtype of the overall **VantageObj::T** type. The two methods that make up the interface allow objects interested in the occurrence of event classes to register interest in some event class and to cancel such a registration of interest.

```
#include "VantageObj.idl"
#include "EventID.idl"
#include "EventCatcher.idl"


module EventGen{

exception UnknownEvent{};
exception NotRegistered{};

interface T : VantageObj::T {

void
register(
    in EventCatcher::T
           toInform,
    in EventID::T
        eventOfInterest,
    in VantageID::T
        whoIsInterested,
    ) raises (UnknownEvent);

void
unregister(
      in VantageID::T
        whoWasInterested,
      in EventID::T
        eventOfInterest
      )raises
    (UnknownEvent,
    NotRegistered);

    };

};
```

Figure 1: IDL interface allowing registration of interest in kinds of events.

The **register** method requires that the caller supply a reference to the object that will receive any notifications of events that are part of this event class, the identifier of the event class of interest, and an identifier for the object that is interested in the event class. On first blush it might appear that identifying either the object that is expressing interest or the object to which the notification is to be sent would suffice. However, the two entities can be distinct, and certainly play logically distinct roles in the protocol. The object expressing interest is the only object that can cancel the registration of interest. However, that object may not be the one that is to receive notifications of occurrences of the event class. Allowing an object to indicate a surrogate for event class delivery allows a third object to enter into the protocol, a feature that we will exploit later to gain more complex functionality.

The **unregister** method cancels a registration. This informs the object that has exported an event class identifier to stop sending notifications of the event class occurrence to the party indicated in the original registration.

The interface also declares two exceptions that can be raised by methods in the interface. The first of these, **UnknownEvent**, will be raised by either method when an attempt is made either to register interest in an event class or to cancel registration of interest in an event class using an identifier that is unknown to the object receiving the call.

The second exception, **NotRegistered**, can be raised by the **unregister** method if some object attempts to cancel the registration of interest in an event class without having first registered interest in that event class.

It would be possible in all the cases in which an exception is raised to try to “do the right thing”, either ignoring the request (in those cases in which **UnknownEvent** is raised) or by simply returning as if everything is all right (in the case in which **NotRegistered** is raised). However, it is more likely that the client will know the right thing to do in such exceptional circumstances than that the server will be able to unilaterally determine the correct action, so we chose to return the exception.

Notifications

To receive a notification of the occurrence of an event, an object not only needs to have registered interest in that event class, but must export the interface that allows receipt of notifications. This interface, contained in the file “EventCatcher.idl”, is shown in Figure 2.

A notification, according to this interface, is simply a message indicating that some event has occurred. Identification of the event class is made by the combination of the object identifier of the object in which the event occurred and the event class identifier of that event as exported by that object.

The interface also defines a single exception, **NoInterest**. This will be raised when an object receives notification of an event class which it does not care about. Raising this exception will allow the object that sent the notification to know that it need not send notifications of this event class to that recipient in the future.

A common feature of other notification systems is the ability to return information other than the occurrence of an event as part of the notification. We have not done this, as part of our goal is using notifications only where RPC will not do. If an object that receives a notification wants to know more about the state of the object in which the event occurred, it can obtain that information by making other method calls to that object.

Having all notifications described with a single signature also means that it is simple to insert third parties into the chain of notifications. If different event classes generated different kinds of information in their notifications, it would not be simple to write an agent that accepted notifications and then handed them off to others.

```
#include "VantageObj.idl"
#include "EventID.idl"

module EventCatcher{
exception NoInterest{};

interface T : VantageObj::T {

void
notify(
    in VantageID::T   from,
    in EventID::T     whatEvent
    ) raises (NoInterest);

    };
};
```

Figure 2: IDL interface for objects that can receive notifications of event occurrences

Implementing the Simple Interfaces

While we use the OMG CORBA IDL to describe the interfaces in our system, we do not use a CORBA-based distributed object management facility (DOMF) in our implementation. Our implementation language is Modula-3 [10], and our distribution substrate is the Network Objects package distributed with the DEC SRC version of that language [4], enhanced to support automatic activation of distributed objects. This gives the CORBA functionality needed for our research in a simple form, integrated into our implementation language.

As with most RPC based systems, much of the code needed to send a message from one object to another is generated by a compiler that takes as input IDL specifications and gives as output source files in the target language. To translate from IDL into Modula-3, our compiler creates objects that derive from the Network Object (**NetObj**) base type. This introduces an additional exception, **NetObj.Error**, to all of the methods declared in an IDL file. This exception is thrown if any of the calls made cannot complete because of problems with the underlying object communication. Thus even though the methods declared for the event generator and event catcher objects return no values and produce no values that must be saved by the client of the calls, the methods are not asynchronous and return of control from such a call without the production of an exception indicates that the call succeeded.

When fed the interface definitions seen above, the IDL compiler will produce a Modula-3 interface for the types **EventGen.T** and **EventCatcher.T.** The task left to the programmer is to implement objects that support these interfaces. Implementations may add other functions that can be called locally. The implementations that follow provide objects that can be utilized by other objects within an address space to keep track of event class registrations. One of these objects will receive notifications, while the other keeps track of which notifications to send when an instance of an exported event class occurs.

The Modula-3 interface for the simple implementation of the event generator object is shown in Figure 3.

This interface extends that produced by the IDL compiler for our **EventGen::T** interface. This interface declares an opaque type **T** (referred to outside of this module as **EventGenImpl.T**) that is a subtype of the type **Public** (referred to outside of this module as **EventGenImpl.Public**). An opaque type is one whose structure is revealed elsewhere; all we know of the type from this declaration is that it has at least all of the methods (and any other structure) defined for the **Public** type.

Type **Public** is declared as a subtype of the **EventGen.T** type, whose definition was produced by the IDL compiler. Objects of type **EventGen.T** must support the **register** and **unregister** methods. In addition to those methods, this interface defines three other methods that all objects that are of type **EventGenImpl.Public** (which includes objects that are of type **EventGenImpl.T**) must support.

The first of these new methods is **addEvent**, which simply tells the object taking care of event class registrations and notifications that the event identified with the indicated event class identifier is one that is exported by this object. The second method, **delEvent**, will end the ability of objects to register interest in a particular event class and will also keep any notifications of that event class's occurrence from being sent. The final method, **trigger**, tells the **EventGenImpl.T** object that an event with the indicated identifier has occurred, thus causing it to send a notification to any object that has registered interest in that event class.

```
INTERFACE EventGenImpl;
IMPORT EventGen;
IMPORT VantageID;
IMPORT VantageObj;
IMPORT EventID;

TYPE T <: Public;

TYPE Public = EventGen.T
OBJECT
METHODS
    addEvent(
        newEvent : EventID.T);
    delEvent(
        delEvent : EventID.T);
    trigger(
        fireEvent : EventID.T)
    RAISES
    {EventGen.UnknownEvent};
END; (* object *)

PROCEDURE New(
    owner : VantageObj.T): T
    RAISES {};
END EventGenImpl.
```

Figure 3: Modula-3 interface for objects that generate event notifications

The interface also includes a **New** procedure, which creates an instance of the **EventGenImpl.T** object. This procedure can take an argument that indicates that the object created should consider itself part of a larger, compound object and should therefore return the object identifier of its owner when asked for its object id.

These additional methods are not available to clients who only know that the object is of type **EventGen.T**. They are part of the particular implementation. There is nothing about the **EventGen.T** interface that requires these functions. Other approaches to implementing the **EventGen.T** interface could use other functions.

It should be noted that the programming model used in this system is one of programming-language level objects being joined as aggregates to form a single distributed object. Thus when we speak of object identifiers, we speak of identifiers that are used to differentiate clusters of programming level objects. This is not a system in which there are "objects all the way down." Instead, objects within the programming language may be hidden as part of the implementation of objects as seen from the point of view of the distributed system.

The interface to the event catcher implementation is similar in that it declares an opaque type as a subtype of a public, abstract type that in turn inherits from the type that is generated from the IDL interface **EventCatcher::T**. The resulting interface is shown in Figure 4.

The **EventCatcherImpl.T** type will implement all of the methods of both the EventCatcher.T type (which includes only the **notify** method) and the **EventCatcherImpl.Public** type. This latter type includes two methods, **register** and **unregister**.

As with the simple implementation of the **EventGen.T** object, this implementation of the **EventCatcher.T** object is meant to provide functionality to a larger cluster of objects that wishes to receive notification of events from other objects. So like the **EventGenImpl.T** interface, there is a **New** function that takes as an argument the containing local object.

The **register** method allows local objects that are using the service provided by the **EventCatcherImpl.T** object to tell the **EventCatcher-**

Impl.T that the object wishes to register interest in some event class occurring in some third object. This method requires that the object of interest be indicated, both by a handle that can be used to call the object and by its object identifier, and that the event class of interest be identified. In addition, this method requires the caller to pass in a closure; this closure includes the procedure to be called on receipt of notification, along with a structure that contains data to be used by that procedure.

On receipt of a **register** call, the **EventCatcherImpl.T** object will register interest in the indicated event class with the indicated object. It will also set up internal structures to insure that a thread will be spawned on receipt of a notification of that event that will run the closure handed in. Exceptions returned to any of these calls are passed along to the caller.

Similarly, the **unregister** method will call the **unregister** method in the **EventGen.T** object that was originally called during registration. Any exceptions raised during the attempt to cancel the registration of interest will be passed along to be handled by the local caller.

```
INTERFACE EventCatcherImpl;
IMPORT EventCatcher;
IMPORT EventGen;
IMPORT EvClosure;
IMPORT NetObj;
IMPORT VantageID;
IMPORT VantageObj;
IMPORT EventID;

TYPE T <: Public;

TYPE Public = EventCatcher.T
              OBJECT

METHODS
register(
    objOfInterest :
EventGen.T;
    objOfInterID :
VantageID.T;
    eventOfInter : EventID.T;
    onNotify : EvClosure.T
) RAISES {
    EventGen.UnknownEvent,
    NetObj.Error};

unregister(
    objOfInter : VantageID.T;
    eventOfInter : EventID.T;
) RAISES {
    EventGen.UnknownEvent,
    EventGen.NotRegistered,
    NetObj.Error};

END; (* object *)

PROCEDURE New(
    owner : VantageObj.T): T
    RAISES {};

END EventCatcherImpl.
```

Figure 4: Modula-3 interface for objects that receive event notifications

Implementation Results

Implementation of the simple **EventGenImpl.T** and **EventCatcherImpl.T** object types took 1,078 lines of Modula-3 code (including comments), of which 604 had to be written by hand and 474 were generated either by the IDL to Modula-3 compiler (450 lines) or by expansions of Modula-3 generic object types (24 lines). Tests were run on a SPARCstation 10 with a single 36 MHz. processor and 96 megabytes of memory running SunOS 4.1.3.

The test program created an **EventGen.T** object that exported 10 event class identifiers and an **EventCatcher.T** object. The test consisted of ten iterations of the **EventCatcher.T** registering interest in each of the event classes, triggering each of the event classes (and thus triggering a notification) in the **EventGen.T,** and then canceling the registration of interest in each event class. Times for the methods were kept separately. The closure registered as the notification handler simply returns without doing anything; however, a separate user-level thread is spawned for each closure.

The test was run in two configurations. In the first, both the **EventGen.T** object and the **EventCatcher.T** object were in the same process. In the second, the objects were in different processes on the same machine. Each test was run multiple times, but variations in the results from different runs of the same test were too small to be significant.

When both objects were in the same address space, 100 event class interest registrations took .045 seconds; 100 triggers, notifications, and notification handlers took .21 seconds; and 100 cancellations of interest took .016 seconds.

When the two objects were in separate address spaces the times, as expected, were considerably slower. In this configuration, 100 registrations took .53 seconds; 100 triggers, notifications, and notification handlers took 1.04 seconds; and 100 cancellations of interest took .48 seconds.

These timings, and the small amount of code needed to implement the objects involved, give us reason to believe that we have met the goals stated earlier of defining a system that is both easy to implement and efficient.

Limitations of the Simple Approach

The simple approach to events and notifications meets a number of the goals we set out earlier for an event and notification system based on RPC. The way in which event classes are identified keeps the system from violating the abstraction boundary of an object, and keeps us from having to introduce a universal namespace for identifying event classes. As the sample implementation shows, the system can be implemented in a way that is cheap in terms of implemenation effort and runtime costs.

The simple approach does have some serious limitations. The CORBA approach to distributed object computing includes a model of objects that allows them to be active (currently loaded into a process and having at least one thread of control) or inactive (not associated with any process or thread, but with any persistent state on some form of stable store). Further, when a call is directed to an object that is inactive, the system will activate the object. We follow this model. However, activation of an object is a heavyweight activity. For notification of some events activation might be justified. However, there are other circumstances in which the delivery of a notification is not time critical and should not cause an activation; the notification should be postponed until the object is activated to service some other call. In our simple approach to notification delivery, there is no way for the system to allow this kind of "lazy" delivery.

Another limitation of the simple approach is the delivery guarantees that can be made for a notification. Currently there are none, and an object that is unable to deliver a notification to another object is left to deal with the failure itself. There is no way for the recipient to indicate that it desires some level of guarantee, nor is there an easy way to provide such delivery guarantees when they are requested.

Rather than change the basic protocol to deal with these problems, we address them by introducing a variety of third-party agents into the system. This is in keeping with our design goal of building complex features on top of a simple base. These agents must also meet the goal that their use should be transparent to those who are not directly involved in using the service. We will now turn to the design of some of those services.

Notification Storage

One way of dealing with the problem of objects being activated to handle notifications that don't warrant that amount of effort is to interpose a notification storage agent between the object generating the notification and the object receiving the notification. This storage object can then implement various policies concerning when to pass the notification on to the object that originally expressed interest in the event class.

Such a notification mailbox would need to support the interface defined in "EventCatcher.idl." Once it did so, however, that object would look like any other event catcher object to an object that sent notifications. This is in keeping with our earlier goal of allowing complex functionality to be introduced in a fashion that is totally transparent to those not directly using the service.

One such notification storage object that we have already defined and implemented is a specialization of the **EventCatcher::T** object. This object is defined by the IDL interface shown in Figure 5.

Since this interface describes a type that is a subtype of the **EventCatcher::T** type, an **EventBox::T** object will look just like any other **EventCatcher::T** to an **EventGen::T** object.

The additional methods defined in the interface allow an object (itself an **EventCatcher::T**) to tell the **EventBox::T** that it wishes to have its notifications stored. The first method, **requestHold**, initiates such a notification storage. The **EventBox::T** object is told the object identifier of the object that is requesting the storage, the event class identifier of the notification to be stored, and the object identifier of the object from which the notification will originate. In addition, the method requires that the requestor indicate to whom the notification should be forwarded, and the maximum number of these notifications to hold.

An object wishing to use this service will need to make two calls to register interest in an event class. The first call will be made to the **EventBox::T** object's **requestHold** method, asking it to store notifications of a particular event class from a particular object. The second call will be to the **register** method in the object that will generate the notification. This call will be made with the

EventBox::T object being supplied as the destination of the notifications. This was the reason that call distinguishes between the object that is interested in the event class's occurrence and the object that is to receive the notification. Since there may be other third-parties in the chain, we also allow this distinction to be made in the call to the **requestHold** method in the **EventBox**.

The calls to the two objects could be made in the opposite order. However, such an ordering makes it possible that a notification could be sent to the **EventBox::T** object before the **requestHold** asking for storage of that notification was complete.

```
#include "EventCatcher.idl"
module EventBox {

exception UnknownClient{};
exception
EventNotRegistered{};
exception
    NotifierNotRegistered{};

interface T :
EventCatcher::T {
void
requestHold(
    in VantageID::T holdFor,
    in EventID::T eventID,
    in VantageID::T holdFrom,
    in EventCatcher::T
                forwardTo,
    in long maxToHold
    );

void
cancelHold(
    in VantageID::T holdFor,
    in EventID::T eventID,
    in VantageID::T holdFrom
    ) raises (UnknownClient,
        EventNotRegistered
        );

void
emptyBox(
    in VantageID::T boxOwner
    ) raises (UnknownClient);

    };
};
```

Figure 5: IDL interface to an event notification mailbox

The **cancelHold** method simply informs an **EventBox::T** object that its services are no longer desired for a combination of client, event class, and event producer. This method will raise an exception if the client is unknown to the **EventBox::T** object, or if the client has not requested that the indicated event class notifications be held.

The final method, **emptyBox**, simply causes all of the stored notifications being held for the object with the indicated object identifier to be delivered. This method will raise an exception if the object identifier is unknown to the **EventBox::T** object.

This storage object implements a fairly simple form of notification storage and delivery. Like a bank of mailboxes, this sort of **EventBox::T** will store notifications. An individual client can ask for its mailbox to be emptied, at which time all of the notifications stored on its behalf will be delivered. The **EventBox::T** will then continue to store new notifications for that client until it asks for its mailbox to be emptied again.

There are other possibilities for delivery schemes. For example, the storage object could have a call that turned on delivery, causing all held notifications to be delivered but also delivering any new notifications immediately until another method was called, telling the storage object to return to a mode where it holds notifications. These different options will only be visible to the object that is asking for its notifications to be stored, and never to the object that is sending the notification.

Implementation of this notification storage type required another 1,131 lines of Modula-3, 688 lines of which had to be written by the programmer and the remainder of which were written by the IDL compiler or by expansion of generic types. As before, this number includes comment and blank lines.

Other Third-Party Services

While the notification storage server is the only third-party service that we have constructed so far, it is easy to think of other services that could be inserted transparently into the stream of event notifications in a similar way.

Perhaps the most obvious of these is a notification store-and-forward service that could relieve the object generating event notifications of the responsibility of dealing with failures in sending those notifications. Such a service would need to support the interface **EventCatcher::T**, enhanced to allow clients of the service to register information concerning the recipients of notifications and to specify reliability guarantees and other policy issues.

An implementation of such a service could receive a notification, attempt to pass it on to the intended recipient, and if that notification failed back off for some period of time and try to send the notification again. The time between notificaiton attempts could increase each time the notification failed, until such a time as the service simply gives up, perhaps informing the object for which it was sending the notification.

Another service that could be provided would be an object that acted as a dispatcher for other objects that generate notifications. Rather than keeping track of all of the objects that have registered interest in getting a notification of some event class, an object could pass responsibility for this task to some third-party object. When some event occurs, only the third-party would need to be informed by the object in which the event occurred. All other objects would be sent a notification from the dispatching object. Indeed, such an agent could also supply store-and-forward functionality as outlined above.

One could easily imagine a distributed system in which each machine on the system had a single, well-known notification dispatcher (perhaps implementing some store-and-forward policy) and a well-known notification storage service. Such a system would have all notification messages go from the object in which the event occurred to the local dispatcher, from that dispatcher to the various notification storage services, and from those services to the (again local) objects that had originally registered interest. Such a design would isolate the network traffic for notifications to be between the specialized services.

Yet another service is one that can track the occurrence of events in objects and generate new events in response. Such an event monitor object could be made arbitrarily complex, generating different event classes for various sequences of events in different objects.

Other sorts of third-party services can be postulated; this is left as an exercise to the reader. What is important is that these services can be introduced so that objects using the basic protocols are unaware of the existence of those objects. This allows new services to be introduced without changing the objects that are not direct clients of (but may, perhaps, be recipients of) the service.

Remaining Issues

While the system described in this paper is powerful and flexible, there are still some open issues concerning the use of events and notifications in an RPC based distributed system.

Perhaps the simplest of these is the question of whether notifications should carry sequence numbers. Currently, an object receiving multiple notifications of an event class from an object has no way of knowing the order of those notifications. However, since notifications carry no information other than that the event occurred, there is no need to order the notifications received, for there is no difference between the notification for the nth occurrence of an event class and the notification of some other occurrence of that event class.

Sequencing might be an issue when notifications are held in a notifications storage server, however. Since a server like the one we have implemented is told a maximum number of notifications to hold, it is possible that an object will miss some notifications because the server has discarded them. Adding a sequence number to the information passed by a notification would allow the client of such a notifications storage server to determine how many notifications were discarded.

We believe that such functionality, if necessary, should be added to the notification storage service rather than to the basic notification interface. However, future experience may change this view.

This does not address the detection of duplicate notifications, which is another reason for introducing sequence numbers. If such detection is needed, we can easily add a way of delivering an event sequence number to the protocol.

A second issue has to do with cleaning up registrations of interest when the object that made the registration disappears (for whatever reason) without cancelling that registration. The current implementation will not find this out. The best that can be done is that the object sending the notification can find out that the object is unreachable. This, however, could be due to repeated bad luck with the network. Just retaining the registration means that over time the amount of “orphaned” registrations could grow without bound.

A final issue is a more theoretical concern. One general principle of object-oriented programming is that interfaces not related by inheritance should be independent at least to the extent that an object may support one interface without supporting the other. Yet our approach to events and notifications appears to violate this principle. An object that supports only the **EventGen::T** interface but does not support any interface that contains a method that returns an event class identifier does not seem very interesting. In the same way, it would seem that any object that supports an interface that contains a method that returns an event class identifier should support the **EventGen::T** interface.

There are a number of unsatisfying approaches to this issue, including requiring any interface that contains a method returning an event class identifier to derive from the **EventGen::T** interface. We are more inclined to the view that this kind of dependency between interfaces shows that there is more to interface relationships than can be captured in an inheritance hierarchy.

Conclusions

We have shown how a system of events and notifications can be built on top of simple interfaces in a distributed system built on the paradigm of RPC for communication. The interfaces used are very simple, allowing cheap implementations that are efficient.

We have also shown how more complex functionality can be introduced into the simple system by introducing third-party servers. These servers can be added to the system in such a way that those who merely receive their services (as opposed to those who make use of those services) need not be aware of them. Further, the addition of such services can be made in a way that does not add any complexity to the basic interfaces.

A number of lessons can be learned from this exercise. The first is that RPC and event notifications can co-exist in a distributed system, especially if one is careful to make sure that each does only the work for which it is best suited. Another is that changes in abstract state can be exported without violating the object metaphor. Finally, we have demonstrated how complex services can be built on top of simple interfaces by introducing third-party objects that support the complexity.

References

[1] Babaoglu, Ozalp and Keith Marzullo. "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms" in Sape Mullender (ed.), *Distributed Systems, Second Edition*, Addison-Wesley (1993).

[2] Birman, K.P. and T.A. Joseph. "Exploiting Virtual Synchrony in Distributed Systems," in *Proceedings of the Eleventh Symposium on Operating Systems Principles*, Austin, Tx. (1987).

[3] Birrell, A. D. and B. J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems* 2 (1978).

[4] Birrell, Andrew, Greg Nelson, Susan Owicki, and Edward Wobber, "Network Objects," Digital Equipment Corporation Systems Research Center Technical Report 115 (1994).

[5] Dasgupta, P., R. J. Leblanc, and E. Spafford. "The Clouds Project: Designing and Implementing a Fault Tolerant Distributed Operating System." *Georgia Institute of Technology Technical Report GIT-ICS-85/29 (*1985).

[6] DellaFera, C. Anthony, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl, and William E. Sommerfeld, "The Zepher Notification Service," *Proceedings of the Winter USENIX Conference* (1988).

[7] Edelson, Daniel. *Enterprise Wide Distributed Programming with InterStage: An Overview*, talk presented at the 1994 USENIX C++ Advanced Topics Workshop, Boston, MA. (1994).

[8] Hutchinson, N. C., L. L. Peterson, M. B. Abott, and S. O'Malley. "RPC in the x-Kernel: Evaluating New Design Techniques." *Proceedings of the Twelfth Symposium on Operating Systems Principles* 23, no. 5 (1989).

[9] Khalidi, Yousef A. and Michael N. Nelson. "An Implementation of UNIX on an Object-Oriented Operating System." *Proceedings of the Winter USENIX Conference* (1993). Also *Sun Microsystems Laboratories, Inc. Technical Report SMLI TR-92-3* (December 1992).

[10] Nelson, Greg (ed.), *Systems Programming with Modula-3*, Prentice Hall (1991).

[11] The Object Management Group. "Common Object Request Broker: Architecture and Specification." *OMG Document Number 91.12.1* (1991).

[12] Parrington, Graham D. “Reliable Distributed Programming in C++: The Arjuna Approach.” *USENIX 1990 C++ Conference Proceedings* (1991).

[13] Shirley, J, *A Guide to Writing DCE Applications*, O’Reilly & Associates (1992).

[14] Skeen, Dale. “An Information Bus Architecture for Large-Scale, Decision-Support Environments”, *Proceedings of the Winter USENIX Conference* (1992).

[15] Zahn, L., T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, and G. Wyant. *Network Computing Architecture*. Prentice Hall (1990).

Jim Waldo (jim.waldo@east.sun.com) is a Senior Staff Engineer at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, he worked in distributed systems and object-oriented software development at Apollo Computer (later Hewlett Packard), where he was one of the original designers of what has become the Common Object Request Broker Architecture (CORBA).

Ann Wollrath (ann.wollrath@east.sun.com) is a Member of Technical Staff at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, she worked in the Parallel Computing Group at the MITRE Corporation investigating optimistic execution schemes for parallelizing sequential object oriented programs.

Geoff Wyant (geoff.wyant@east.sun.com) is a Staff Engineer at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, he worked at CenterLine Software and Apollo Computer (later Hewlett Packard), where he was involved in the original design and implementation of the Network Computing System.

Samuel C. Kendall (sam.kendall@east.sun.com) is a Member of Technical Staff at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Before joining Sun, Sam worked on C and C++ programming environments at CenterLine Software, and on the first compiler for C*, a data-parallel cousin of C++, at Thinking Machines Corporation.