

Using EJBs with the OMG Notification Service

Challenges and Benefits

A Borland® White Paper

*By Brenton Camac
Borland Software Corporation*

September 2002

Borland®

Contents

Introduction.....	1
What is the OMG Notification Service?	1
The four kinds of Notification Channel	3
Using the Typed-Event Notification Channel with EJBs	5
An Illustration	6
The application without VisiNotify™	6
Adding VisiNotify to the application	7
Subscribing EJBs to VisiNotify.....	10
Benefits of using Typed-Event Notification Channel	11
Considerations using Notification Channel.....	14
Conclusion	17
Using the Structured-Event Notification Channel with EJBs ...	17
An Illustration	18
Making an EJB an event consumer	18
Subscribing EJBs to VisiNotify.....	19
Benefits of using the Structured-Event Channel	20
Considerations using Notification Channel.....	21
Conclusion	22
Summary of EJB and Notification Service interworking	23
About the author	24

Introduction

Applications based on Enterprise JavaBeans™ (EJBs™) can benefit in many ways from using the Object Management Group™ (OMG™) Notification Service. However, there are challenges to operating the Notification Service in an EJB environment. To highlight some of the more significant challenges and benefits, two cases are presented here. The first involves a distributed EJB-based application that uses the Typed-Event Notification Channel to improve its throughput, reliability, and scalability. Here, the challenge faced is propagating ValueTypes through the Notification Channel. The second case involves subscribing EJB to the Structured-Event Notification Channel as one approach to integrating the Java™ 2 Platform, Enterprise Edition (J2EE™) and the CORBA® platform for event-based applications. In this case, the challenge faced is the requirement to “RMI-ize” structured events. Complete source code for both cases is available on the Borland Web site [3].

What is the OMG Notification Service?

The Notification Service is a standardized service of the CORBA platform defined by the OMG[1]. It provides applications with the ability to use decoupled communication between their elements instead of traditional direct/synchronous communication. The Notification Service supercedes a similar, but more basic service—the Event Service[2]—by offering more features and maintaining backwards compatibility.

A key element of the Notification Service is the Notification Channel (referred to here as the Channel) whose role is to propagate events from suppliers to consumers. Once an event has been delivered to the Channel the Channel takes responsibility for delivering it to each subscribed consumer. This arrangement is shown in Figure 1.

The default behavior of the Notification Channel is to deliver every event it receives to every subscribed consumer. This is also the behavior of the Event Channel. However, the Notification Channel has the facility to filter events and thereby provide selective delivery. To use this facility, consumers specify which events they are interested in receiving by

registering a filter expression with the Notification Channel. The Channel then applies the filter expression to each event to determine whether it should be delivered to that consumer or not. This and other features, such as Quality of Service parameters, can be used to tailor the behavior of the Channel. However, these facilities are only mentioned here to provide an overview of the capabilities of the Channel. The examples discussed in this paper do not require such advanced facilities, even though their use in these cases is conceivable.

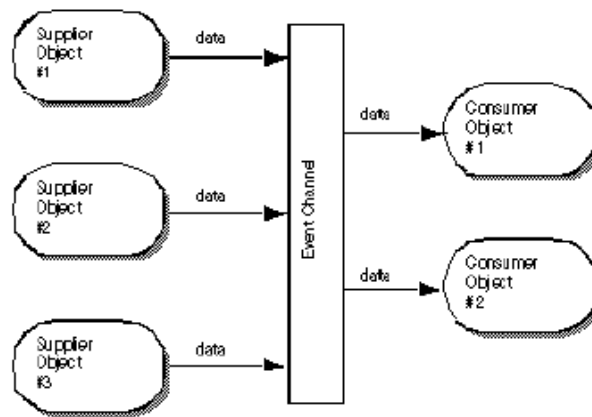


Figure 1. Notification/Event Channel

A consequence of application elements using the Notification Channel is that they no longer communicate directly with each other but indirectly via the Channel. There are many benefits arising from this decoupling, including the following:

- Supplier elements can deliver events at different rates to which consumer elements process them. And therefore, they can produce events at a different rate as well. In this respect, the Channel acts as a buffer, accommodating and leveling-out peaks in an application's processing activity.
- The absence or unavailability of consumer elements does not prevent supplier elements from delivering events. In this respect, the channel allows an application to continue functioning even when parts of that application are unavailable.
- A supplier can send an event to every consumer by creating a single event and delivering it to the Channel. In this capacity, the Channel acts as a broadcast medium for the

application. If filtering is used in the Notification Channel, then the Channel acts as a multicast medium.

- The identity of consumers is not needed by suppliers in order to reach them; only the identity of the Channel is needed by consumers and suppliers. Because of this, suppliers and/or consumers can be introduced to a system without requiring reconfiguration of existing suppliers or consumers in order to accommodate them. This has enormous benefit for large distributed applications.

Distributed application architectures can use these characteristics of decoupled communication to improve their performance, reliability, scalability, and adaptability. Before investigating such uses of the Notification Channel with EJB-based applications, a brief description of the four different kinds of Notification Channel is necessary since two of them—the Typed Channel and the Structured Channel—will be used in subsequent examples.

The four kinds of Notification Channel

The Notification Service specifies four different kinds of Notification Channel: the Untyped-Event Channel, the Structured-Event Channel, the Sequenced-Event Channel, and the Typed-Event Channel. All propagate events from suppliers to consumers, but each represents the event using a different form. This choice of representation allows applications to choose a channel most suited to their needs.

The Untyped-Event Notification Channel uses the `CORBA : Any` datatype to represent events. To deliver such datatypes, it also defines a set of predefined IDL interfaces with standard operations for communicating events between the channel and its clients. Typically, clients of the Untyped-Event Notification Channel will define their own data structure for storing event information and then package that into the `Any` datatype. This allows applications to continue using their own event datatype with the Notification Channel and is one of the primary advantages of the Untyped-Event Channel.

The Structured-Event and Sequenced-Event Notification Channels use predefined structured datatypes to represent events. For the Structured Channel, the datatype comprises a fixed header, plus a variable header, and a body. The headers can include standard fields, as well as

name/value pairs. A complete definition of the structured event datatype is found in the Notification Service specification (Section 2.2, [1]). The Sequenced-Event Channel's event datatype is a sequence of the Structured-Event Channel's event datatype. In this way, the Sequenced Channel can transport batches of events via a single invocation. A significant advantage of both Structured-and Sequenced-Event Channels is that their event filtering activities are much more efficient than the other kinds of channels because the Event Structure is known to the Channel *a priori* meaning that event demarshalling and inspection algorithms that must be implemented in the Channel can be significantly optimized at design-time.

The fourth kind of Channel is the Typed-Event Notification Channel. This differs from the other kinds of channels in that it does not use an explicitly defined data-structure to represent events. Nor, therefore, does it provide predefined IDL operations such as `push()` or `push_structured_event()` to transfer events between client and channel. Instead, clients communicate with Typed Channels through application-defined interfaces. The Notification specification refers to these interfaces as <I> interfaces. Operations of an <I> interface which are to be used to transfer events must specify a void return type and not use any `out` or `inout` parameters. One or more event operations may be defined in an <I> interface. Calling an event operation of an <I> interface obtained from a Typed-Event Notification Channel causes the invocation together with its argument values to be sent to the Channel. The Channel then treats this invocation as a Typed-Event. Similarly, to receive Typed Events, consumer applications implement the <I> interface and subscribe that CORBA object to the Typed-Event Channel. The Channel then delivers the events by invoking the specified operation with the accompanying parameter values as received from the supplier.

One significant advantage of using the Typed-Event Notification Channel is that an application's elements communicate via strongly typed application-level interfaces and therefore do not need to encode or decode to and from an event datatype when publishing or consuming events. By contrast, when using a Structured, Sequenced or Untyped Channel, an application must use a generic interface which requires events to be formatted into an explicit event structure, delivered using an infrastructure-level operation, and extracted from the event

datatype. Non-Typed channels therefore force clients to use a contract which is semantically weaker, less object-oriented, and less type-safe than that offered by the Typed-Event Channel.

Although beyond the scope of this paper, it should be noted that the Notification Specification also defines rules for how events are to be mapped between their different forms—Untyped, Typed, Structured, and Sequenced. Hence, it is possible for suppliers and consumers to be bound to different kinds of Notification Channel yet interoperate with each other.

In summary, there are four different kinds of Notification Channel specified by the OMG. Their distinguishing characteristic is the form used to represent events externally to clients. All four kinds of Notification Channel are implemented by Borland® VisiNotify,™ an implementation of the CosNotification Service and add-on service available with Borland® Enterprise Server, VisiBroker® Edition. The Structured-Event and Typed-Event Channels are used in the examples that follow.

Using the Typed-Event Notification Channel with EJBs

This section will examine how the Notification Service can be used as an asynchronous communications mechanism by EJB-based applications. Benefits to the application of this include greater throughput and the potential to improve reliability and scalability, all while requiring a minimal amount of change to an existing application. However, operating the Typed-Event Notification Channel in such an environment presents certain challenges. Those challenges are also examined, along with how VisiNotify resolves them and thus qualifies for use in such a role.

An Illustration

To examine how the Typed-Event Notification Channel can be used in this way, the following illustration is used. The illustration initially comprises a simple stateless-session EJB that is invoked by a remote client. The source code for this entire illustration is available separately[3]. The Typed-Event Channel is then added as an intermediary between the client and EJB, and as a result, the client and EJB no longer interact directly with each other but indirectly via the channel. Hence, the application's communication mode is changed from being a traditional synchronous mode to an asynchronous one.

The application without VisiNotify™

The stateless session EJB provides a simple application-level interface. Its remote EJB interface is shown in Figure 2. This interface exposes the business methods of EJB.

```
// OrderProcessor's EJB Remote interface

public interface OrderProcessorRemote extends javax.ejb.EJBObject

{
    public void createOrder( NewOrderSpec theOrder )
        throws RemoteException;

    public void updateOrder( ModifyOrderSpec theOrder )
        throws RemoteException;

    public void deleteOrder( DeleteOrderSpec theOrder )
        throws RemoteException;
}
```

Figure 2 The remote interface of the EJB

Before VisiNotify is introduced to the design, the client interacts directly with the EJB, using the RMI-IIOP protocol. (The J2EE 1.3 specification [4, section 7.2.2] requires that compliant EJB containers support the RMI-IIOP protocol.) When using the RMI-IIOP protocol to interact directly with the EJB, the client is operating in a synchronous mode. That is, once the client dispatches the request to the EJB, it must wait (or block) until a response is received. For this period of time, the client's processing becomes synchronized with the EJB processing. Synchronous invocations are the primary mode of interaction found in the majority of EJB-based application designs of today.

Notice that none of the operations in Figure 2 are expected to return information or raise application-defined exceptions. This condition is necessary if migration from using synchronous mode to asynchronous mode is to be transparent to the application. If this condition is not met, then design patterns can be applied to accommodate the return flow of information [7,8] as well as other facilities[5]. However this does not imply that all processing activities of a distributed application should be migrated to use this form of communication or that all applications activities will benefit from asynchronous communication. Typically, there will exist within an application certain activities that are more suited to the synchronous communication mode. Hence, this technique should not be applied indiscriminately to distributed application architectures, but selectively and after thoughtful consideration.

Adding VisiNotify to the application

To this configuration, VisiNotify is inserted between the client and remote EJB as a mediator. Since the Typed-Event Channel is being used in this case, the client continues to use the same EJB (remote) interface which the Notification Service would regard as the <I> interface. Hence, no change is required to those parts of the client that invoke the remote operations. But, the client's service discovery algorithm will need to be changed from its present approach of obtaining the remote interface (object reference) of the EJB to obtaining a TypedPushSupplier proxy (object reference) from the Notification Channel. Sample code to perform this activity is shown in Figure 3.

```
public class ServiceLocator
{
    // some sensible defaults
    boolean syncMode = true;
    String channelURL = "corbaloc:127.0.0.1:14100";
    String ejbHomeName = "OrderProcessor";

    /**
     * Set mode to determine where the remote object reference is
     * obtained from.
     * @param syncMode    when true, connect to EJB directly,
     * otherwise use N.Channel.
     */
    public void setMode (boolean syncMode)
    { this.syncMode = syncMode; }

    /**
     * Sets the JNDI name to use to locate the EJB's Home i/face.
     * @param ejbHomeName JNDI Name of Home interface
     */
    public void setEJBHomeName (String ejbHomeName)
    { this.ejbHomeName = ejbHomeName; }

    /**
     * Sets the URL to use to locate the Notification Channel
     * @param URL URL of Notification Channel
     */
    public void setNotificationChannelURL (String URL)
    { this.channelURL = URL; }
```

```
/**
 * Obtains the remote object reference of an object
 * supporting the OrderProcessor interface.
 * @return A bound OrderProcessor stub
 */
public OrderProcessorRemote getOrderProcessor()
{
    if (syncMode == true) {
        // lookup via JNDI
        java.lang.Object ref =initialContext.lookup(ejbHomeName);
        // narrow ref (using RMI) to a usable type
        OrderProcessorHome home = (OrderProcessorHome)
            javax.rmi.PortableRemoteObject.narrow(
                jRef, OrderProcessorHome.class);

        // obtain reference to Remote i/face of SLSB
        return home.create();
    }
    else { // syncMode == false
        // obtain proxy from Typed Notification Channel
        org.omg.CORBA.Object ref =
            TypedPushSupplier.get_typed_consumer(channelURL);
        // narrow ref (using CORBA) to usable type
        return OrderProcessorRemoteHelper.narrow(ref);
    }
}
}
```

Figure 3 Service discovery for client

(Note. Many of the methods used by the Service Locator class in Figure 3 can throw exceptions, none of which are caught by the code. This, and the many performance optimizations of the code which could be included, have been omitted for the sake of brevity.)

To use the Service Locator, the client would first configure it to use either synchronous or asynchronous mode via `setMode()`; then specify the appropriate target using either `setEJBHomeName()` or `setNotificationChannelURL()` as appropriate; then it can invoke the `getOrderProcessor()` method to obtain a reference to an `OrderProcessor` object. Notice that this last method does not require the client to know which communication mode is being used.

Good design practice calls for applications to encapsulate such service location activities and factor them into one place within the application[6]. When this practice is followed, changes such as that outlined previously can be made easily and with little impact on the remainder of an application.

Subscribing EJBs to VisiNotify

With the above changes in effect, the EJB will no longer be receiving invocations directly from the client as it did before. Therefore, the EJB will need to be subscribed (i.e. connected) to the Notification Channel in order to receive invocations from this client. Although the subscription activity can be done programmatically, it is more easily achieved with the subtool utility of VisiNotify.

The subtool utility performs the following actions:

1. Using a supplied JNDI, it obtains the EJB home interface object reference from JNDI.
2. Invokes the parameterless `create` method on the home interface to obtain an object reference to the EJB remote interface.
3. Registers the EJB remote interface with the Notification Service identified by the supplied URL.

For this sample application, the `subtool` utility would be invoked with the following parameters:

```
prompt> subtool \  
-type typed  
-home OrderProcessor  
-key OrderProcessingChannel  
-ORBDefaultInitRef corbaloc::127.0.0.1:14100
```

Figure 4 Using the subtool utility to subscribe an EJB to a Typed Channel

The *type* argument indicates the kind of Notification Channel being subscribed to. The *home* argument is the JNDI name of the EJB home interface. The *key* argument has the effect of creating a subchannel within the overall Typed-Event Channel. See the VisiNotify product documentation for more information about this feature. The *ORBDefaultInitRef* argument provides the URL to the Notification Service. In this instance, VisiNotify is running on the same machines as where `subtool` is being executed, and by default, VisiNotify operates on port 14100.

Since *subtool* uses only standard JNDI and EJB operations, it is able to subscribe any Session or Entity EJB hosted in any J2EE-compliant application server—including Borland Enterprise Server. And for the same reason, it also works with any standards-compliant Notification Channel provided that implementation supports the Typed-Event Channel facility.

Benefits of using Typed-Event Notification Channel

Inserting the Typed-Event Notification Channel between the client and EJB changed the communication mode from synchronous to asynchronous. To examine the effect this change has on the application's operation, the following analysis was undertaken. The length of time a client spent making an invocation when running against the VisiNotify channel (asynchronous mode) was measured, as was the length of time taken when contacting the EJB directly (synchronous mode). These tests were performed in batches of one, five, and 10

invocations. For the synchronous calls, every batch incurred the one-time cost of binding to the remote EJB (i.e. establishing a network connection from the client to the EJB server).

The summary of this information is recorded in Figure 5.

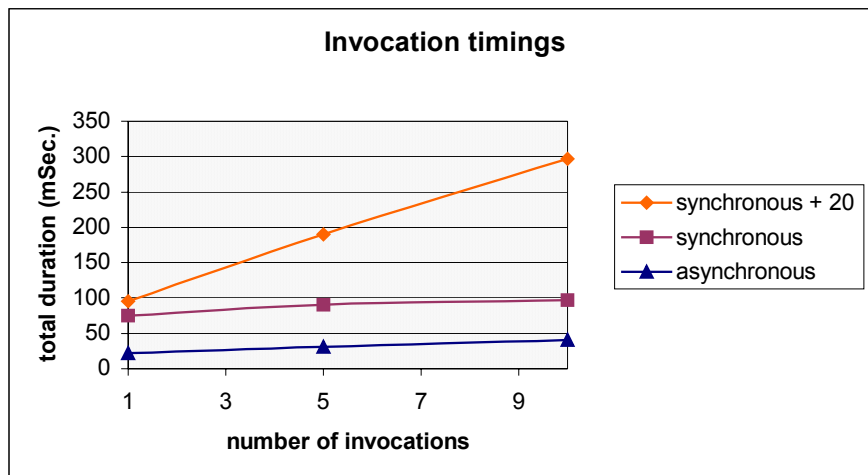


Figure 5. Timing synchronous and asynchronous invocations

[Figure 5 shows measurements of the application's invocation timings using TypedChannel (asynchronous) and direct EJB invocations (synchronous) executing on a dual-800MHz processor/512Mb RAM/Windows® 2000 server. The EJB server was Borland® Enterprise Server, AppServer™ Edition. Sample size of each data point = 5, standard deviations of each sample ranged between 6.8 and 10.6]

Two important details can be observed from these measurements. First, asynchronous invocations are completed sooner than synchronous ones. A single asynchronous invocation, for instance, took an average 22 milliseconds ($\sigma=8.4$) to complete compared with 75 milliseconds ($\sigma=7.4$) for the ideal synchronous equivalent. This is quicker by a factor of 100:29. And even at 500 invocations per test, it is still quicker by a factor of 20:10. Second, this performance gap between synchronous and asynchronous modes widens substantially when the processing time of the EJB is included. This is shown by the plot 'synchronous+20' which records an EJB spending 20 milliseconds per invocation on user-level processing. This

gap becomes more pronounced as the number of invocations increase. To account for these observations, the following explanations are offered.

When the client uses the asynchronous model to communicate with the EJB it is interacting with a VisiNotify proxy co-located with it in the same process. That proxy blocks the client only to receive and buffer the invocation; it does not block the client while the invocation is being delivered remotely to the Notification Service. Consequently, the client is not blocked on any remote activity. Whereas when the client makes the same invocation directly on the EJB, it is blocked while the request is remotely delivered. Thus, an asynchronous invocation takes less time to execute than the equivalent synchronous one.

Furthermore, in reality, the EJB will also need to perform some quantifiable amount of work to process the request, whether it is to store the received data for later processing or process the request immediately. The amount of work will vary across applications and operations, but will take some amount of time. Such additional activity is represented in Figure 5 as the ‘synchronous+20’ plot, depicting the EJB spending 20 milliseconds per invocation to process the request. For clients that interact with EJBs synchronously, this remote processing time will become part of their processing time since they are blocked until the remote task completes. This accounts for the gap and its widening as the number of invocations per batch increases. Consequently, with synchronous communication, when performance degrades in the remote task, the client’s performance similarly suffers.

In addition to performance improvements, there are other benefits from using the asynchronous mode including improved reliability and scalability. By using asynchronous communication, it is not necessary for the EJB to be reachable or even running in order for the client to dispatch an invocation to it. This feature allows the application to be more resilient when communication between the EJB and client is interrupted, or when the EJB container or platform itself becomes unavailable. Especially for large distributed applications, being able to tolerate partial outages improves overall reliability because such problems remain localized rather than propagating to other tiers and affecting other parts of the application. Furthermore, asynchronous communication can increase an application’s scalability as it allows replica EJBs to be subscribed to the channel easily, i.e. without

requiring changes to the application. This feature can be used to provide fault tolerance and load balancing at the infrastructure level of the architecture, which is generally preferred over implementing such at the application level.

There is also another benefit to using the Typed-Event Channel for this purpose instead of the other kinds of Notification Channel. By using the Typed-Event Channel, the EJB interface did not need to be changed and thus could continue to be reached via synchronous facilities (i.e. RMI-IIOP) as well as through a new asynchronous route (i.e. VisiNotify). That is, the EJB is not concerned with the mode of communication used to reach it. Furthermore, this additional capability can be introduced to the application without requiring any changes to an EJB interface and without requiring the development of additional components such as Message Driven Beans (MDB) to receive and interpret the message content. Only on the client side are changes required, where the service location activity would need to be augmented to retrieve a TypedPushSupplier reference from VisiNotify.

The above benefits are available by using the Notification Channel with EJBs. However, there is also a challenge faced by the Notification Channel when used in this way: the propagation of ValueTypes.

Considerations using Notification Channel

To use the Typed-Event Notification Channel with EJB in this way requires the channel to propagate ValueTypes. ValueTypes are CORBA IDL constructs for representing sophisticated data structures and are used extensively by the RMI protocol to transfer serializable and other Java-specific data structures via the IIOP transport protocol. Since the J2EE 1.3 specification requires RMI-IIOP be supported by compliant EJB containers [4, section 7.2.2], the Notification Channel can expect ValueTypes to appear within Typed-Events when used in such an environment. Depending on how the Notification Channel is implemented, this can present a problem.

For Notification Services implemented as user-level applications, propagating ValueTypes is, for all practical purposes, impossible. A user-level implementation of the Notification

Channel is one where the channel is implemented as an application that uses application-level facilities of an ORB. This is the approach taken by many commercial implementations today. In this design, when an event is to be propagated, the ORB will first demarshall it, then submit it to the application (channel), which then submits it to the ORB again for remarshaling and delivery to the desired consumer.

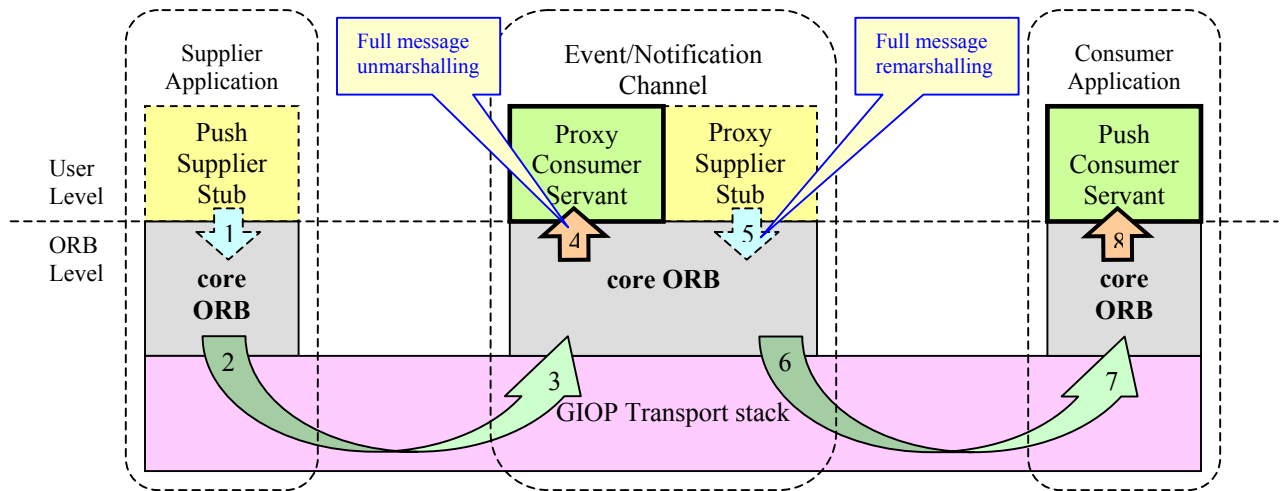


Figure 6. User-level Event Notification Channel

When the event contains a ValueType, the ORB will need to use a ValueType factory to demarshall and remarshal the event. ValueType factories are implemented by application developers or, in the case of valueboxes, by IDL pre-compilers. Since most event channels will not have application specific value-factories built-in, they will not be able to demarshall and marshal such arguments. Consequently, user-level channels can not, in general, propagate events containing ValueTypes and therefore are not usable in an EJB environment.

User-level implementations are the only approach possible when access to the underlying ORB internals is not available. However, when access to the internal APIs of an ORB are available, an alternative approach can be taken—an infrastructure-level implementation. See Figure 7.

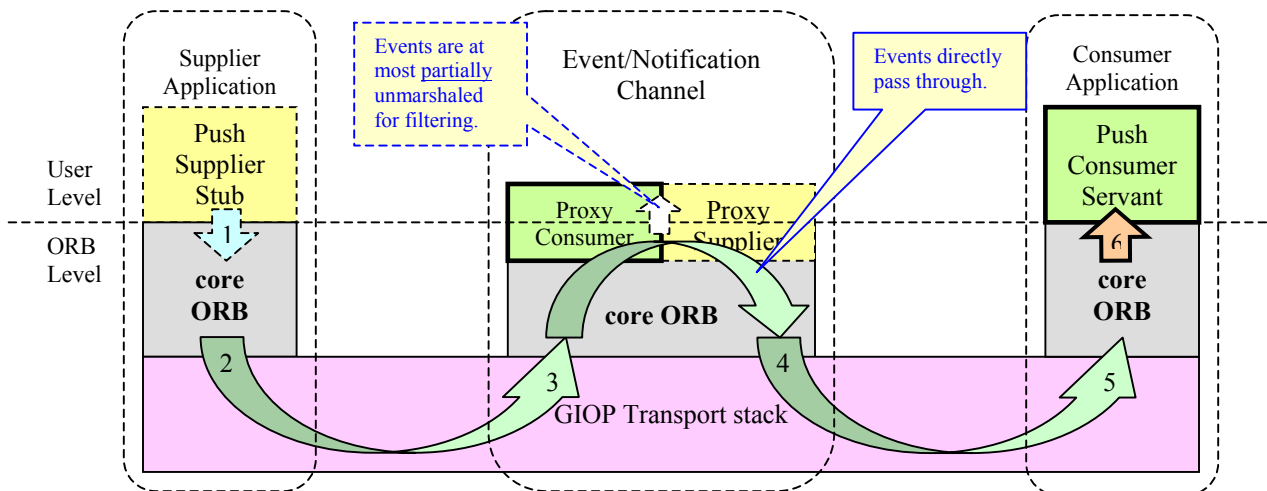


Figure 7. ORB-level Event Notification Channel

The Borland VisiNotify Service uses such an approach in which GIOP requests from the client are not demarshalled but modified for dispatch *in situ*, thus avoiding the ValueType factory problems mentioned previously. With this approach, VisiNotify can accommodate arbitrary ValueTypes without having to load application-defined value factories. Furthermore, not having to demarsh and remarshal events nor use DII/DSI for the Typed-Event facility contributes to the superior performance of the VisiNotify Service compared to other implementations.

The prerequisite to this approach—access to internal APIs of a standards-compliant high-performance ORB—is a significant barrier of entry for many Notification Service vendors; one that has precluded most from developing a general purpose Notification Channel capable of propagating ValueTypes and therefore from developing a Notification Service usable in an EJB environment.

Conclusion

Using the Notification Channel in distributed applications to achieve asynchronous communications can result in improved response time, increased reliability, and the potential to achieve scalability at the infrastructure level. In addition, the Typed-Event Notification Channel is an ideal candidate for this because existing clients and servers are not required to map their application-level, type-safe, object-oriented interfaces into explicit Structured-Event representations. One restriction though is that the approach does require application-defined operations to meet certain conditions. With the use of appropriate design-patterns or services[5], such constraints can be overcome, resulting in application designs that are more appropriate for distributed systems of enterprise scale.

To operate successfully in this environment, the Notification Channel must support the Typed-Event facility and be capable of propagating arbitrary ValueTypes as well. The Borland implementation of the Notification Channel—VisiNotify—satisfies both criteria. Its ORB-level implementation not only provides a high-performance solution but also supports arbitrary ValueType propagation which qualifies it to be used in an RMI-IIOP environment such as the J2EE platform.

Using the Structured-Event Notification Channel with EJBs

EJB-based applications can also use the Notification Service in the more traditional role of an Event Service. In this scenario, EJBs subscribe to the Notification Channel to receive structured events that were created by CORBA-based suppliers through Notification Service defined operations such as `push()` or `push_structured_event()`. The benefits and challenges of this approach compared with other approaches of interworking EJBs with the CORBA suppliers will now be examined.

An Illustration

To examine how the Notification Channel can be used in this capacity, the following illustration is used: an EJB (a stateless session bean) is subscribed to a Structured-Event Channel. The source code for this example is available separately [3].

Integrating an EJB with the Structured-Event Channel is done in three stages: at development time, the EJB must implement the appropriate interface; at deployment time, the EJB must be connected (subscribed) to the Notification Service; and at runtime, the Structured Event must be translated to RMI form. Each of these steps is discussed in further detail below.

Making an EJB an event consumer

For an EJB to be a Structured-Event consumer, it must implement the `push_structured_event()` operation. This operation is called by the Notification Channel to propagate structured events to consumers. The specification of the `push_structured_event()` operation and its only argument—the `StructuredEvent`—is prescribed by the OMG in the Notification Service IDL, see Figure 8.

```
// File: CosNotifyComm.idl from the Notification Service spec.
module CosNotifyComm
{
  interface StructuredPushConsumer : NotifyPublish
  {
    void push_structured_event (
      in CosNotification::StructuredEvent notification
    ) raises ( CosEventComm::Disconnected );

    void disconnect_structured_push_consumer();
  };
  . . .
}
```

Figure 8. *StructuredPushConsumer* IDL definition

Although this interface can be mapped from IDL to Java, it is not possible for the EJB remote interface to simply extend this interface. This is because its methods will not declare an `java.rmi.RemoteException` exception in their throws clause, which is mandatory for methods of RMI remote interfaces. Hence, the EJB must explicitly declare this operation itself using the predefined datatype definition for the event. This is shown in Figure 9.

```
// EJB StructConsumer Remote interface
public interface StructConsumer extends javax.ejb.EJBObject
{
    public void push_structured_event (
        org.omg.CosNotification.StructuredEvent event
    ) throws RemoteException;
    . . .
}
```

Figure 9. EJB structured consumer

This new operation is defined on the remote interface since `VisiNotify` executes outside the EJB server and therefore must reach the EJB remotely. Because of this, the candidate EJB types are restricted to Stateless, Stateful, and Entity, i.e. MDB is precluded because it does not offer remote interfaces. Of these choices, the stateless EJB would be most practical, although using the other two types is conceivable.

Subscribing EJBs to VisiNotify

Before the Notification Channel can relay invocations to an EJB, that EJB must be subscribed to the Channel. Although this activity can be done programmatically, it is more easily achieved with the `VisiNotify` subtool utility.

The subtool utility performs the following actions:

1. Using a supplied JNDI URL, the subtool utility obtains the EJB home interface object reference, then
2. Invokes the parameterless create method on the home interface to obtain an object reference to the EJB remote interface, then
3. Registers EJB with the Notification Service identified by the supplied URL.

For this sample application, the subtool utility would be invoked with the following parameters:

```
prompt> subtool \  
-type struct  
-home StructConsumer  
-key TMNEvents  
-ORBDefaultInitRef corbaloc::127.0.0.1:14100
```

Figure 10. Using the subtool to subscribe an EJB to a Structured Channel

Since the *subtool* uses only standard JNDI and EJB operations, it is able to subscribe any session or entity EJB hosted in any J2EE-compliant application server—including Borland Enterprise Server. And for the same reason, it also works with any standards-compliant Notification Channel which supports the Typed-Event Channel facility.

Benefits of using the Structured-Event Channel

Subscribing EJBs to a Structured-Event Channel can be used to integrate newer EJB-based applications with existing CORBA-based event-oriented applications. It can also be used in hybrid J2EE and CORBA applications as a common messaging service for both platforms.

The following benefits result from using this technique to integrate EJBs with legacy CORBA applications. First, this approach does not rely on non-standard capabilities from the EJB container, the CORBA supplier, or the existing CORBA Event Channel. Thus, the approach is generally applicable and portable. Second, it can be applied to the existing CORBA-based

applications without requiring changes to them. This is an important consideration when dealing with legacy systems because modifications are typically expensive and place existing operations at risk, and therefore minimizing the need to change them is significant.

Connecting VisiNotify to an existing Event Channel can be achieved by subscribing it as a consumer. This is possible because the standardized interfaces of the Notification Channel support chaining of channels and both channel instances will be using a common transport layer protocol (IIOP). Third, the EJB can reuse event processing designs, algorithms, and filter expressions from existing non-EJB consumers, since both consumers use the same event data-structure.

There are other advantages to using a single event-service in a hybrid J2EE and CORBA environment as opposed to integrating the native event services of each platform—JMS and CosNotification respectively. If JMS is used to deliver events published in the Notification Service, then those events must be translated into JMS event structures. Such translation is non-trivial, see [9,10]. Then, once the event is translated, it must then be delivered to the EJB which would require the development of an extra EJB (an MDB) to receive the event from JMS and invoke the target EJB. Because the event has been translated from its original form, the potential of reusing designs, algorithms, and filter expressions from non-EJB consumers for this activity is significantly limited, as these would need to be translated into the JMS domain.

For these reasons there is significant benefit to be gained from using the Notification Channel in this way with EJBs. However, associated with this approach is a challenge which must be resolved by the Channel: It must RMI-ize the Structured Events before delivering them to the EJB.

Considerations using Notification Channel

For the Structured-Event Channel to operate with EJBs in this way, it must be able to RMI-ize the structured events it passes to the EJB. This is an important challenge that the Channel must resolve in order for it to be used in this capacity. An explanation of this challenge is as follows.

Ordinarily, the consumers of the Structured-Event Channel are CORBA objects which implement the `CosNotifyComm::StructuredPushConsumer` interface defined by the OMG in the service's IDL (see Figure 8). However, in this case, some consumers will also be EJBs. Although the EJBs will define the same required operation as their CORBA counterparts (the `push_structured_event()` operation, see Figure 9), in this case, it is defined within an RMI interface and therefore inherits RMI semantics. Consequently, if the channel attempts to push the event to an EJB using this operation, it will fail because the event to be delivered would have been marshaled as a CORBA `struct` datatype (without RMI semantics) by the supplier as per the Notification Service IDL.

To overcome this language impedance, VisiNotify translates Structured Events from their standard IDL-defined `struct` datatype into one acceptable for the EJB. (Informally, such translation is referred to as RMI-zing the datatype). VisiNotify only applies this translation to events destined for consumers it has determined (at runtime) to be RMI objects. This facility allows VisiNotify to deliver structured events to both EJB-based consumers as well as to regular CORBA-based consumers.

Conclusion

There are many advantages and benefits to using the Notification Channel as the event-service in a hybrid J2EE/CORBA environment. These include, but aren't limited to: the ability for EJBs to be subscribed directly to the Notification Channel and not require development of an MDB; the ability for an event to be produced in the CORBA domain and consumed in the J2EE domain without translating it from one form to another and therefore without demarshalling and re-marshalling the information; and the ability for the Notification Channel to seamlessly integrate with other implementations of the Notification Channel which may be present in legacy systems without the need to change them.

To operate the Notification Channel in this environment requires it RMI-ize events before delivering them to EJBs. Since the Borland implementation of the Notification Channel—VisiNotify—performs such a function, it can be used for such purposes.

Summary of EJB and Notification Service interworking

Two separate cases were presented to highlight the challenges and benefits of using the Notification Service with EJBs.

The first case examined the use of the VisiNotify Typed-Event Notification Channel as a means of performing asynchronous communications with EJB-based applications. The benefits of this style of communication to the application included increased throughput, which was seen in the empirical measurements, as well as the potential for greater reliability and scalability. Additionally, the Typed-Event Notification Channel support for strongly typed EJB interfaces made it an ideal candidate for this role. Because the EJBs use the RMI-IIOP protocol, the Notification Channel in this scenario had to support ValueType propagation. This presented a challenge for user-level implementations of the Notification Service because their de-marshalling and re-marshalling activities would need access to ValueType factories. However, infrastructure-level implementations do not face this problem, and VisiNotify, being an infrastructure-level implementation, therefore can be used in such a capacity.

The second case examined the use of the VisiNotify Structured-Event Notification Channel as a common event-service for simultaneous use by both CORBA-based and EJB-based applications. The advantages of this approach over other event-service interworking approaches were discussed. The benefits to the EJB applications include being able to receive events originally produced by CORBA producers, being able to receive them through remote interfaces, and being able to receive them without requiring event translation. Because the EJB remote interfaces are defined with RMI semantics, this means events have to be RMI-ized before being delivered. This presents a challenge for the typical Notification Service implementation because it does not RMI-ize events. However, the ability of VisiNotify to RMI-ize Structured Events for EJB consumers means it can simultaneously service CORBA-based and EJB-based consumers, and thus provide an effective bridge between CORBA and EJB-based environments.

Collectively, these two cases illustrate many of the benefits available from using the Notification Service with EJBs as well as the challenges faced. The cases also illustrate how VisiNotify resolves these particular challenges and thus qualifies itself for use in such capacities.

For more information, contact an enterprise sales representative in your region. In the U.S., call 1-800-632-2864. International customers can find the nearest regional office by referencing http://www.borland.com/company/borland_worldwide.html.

About the author

Brenton Camac (bcamac@borland.com) is a Principal Consultant with Borland Software Corporation where he works with clients on architecting and developing enterprise-scale J2EE applications and distributed CORBA-based systems. Brenton has worked in the software development industry since 1990, has an M.S. and B.S. in computing science, and is a Sun® Certified Enterprise Architect for the Java™ Platform.

References

1. **Notification Service Specification.** Version 1.0, June 2000. (document reference—formal/2000-06-20). The Object Management Group.
2. **Event Service Specification.** Version 1.1, March 2001. (document reference—formal/2001-03-01). The Object Management Group.
3. **Using Typed- and Structured-Event Channels with EJBs.** Java source code by Brenton Camac. <http://codecentral.borland.com/codecentral/ccWeb.exe/listing?id=19238>
4. **Java 2 Platform, Enterprise Edition Specification, v1.3.** July 2001. OMG Protocols. Sun® Microsystems.
5. **Native Messaging** Borland® Enterprise Server documentation
6. **The Service Locator Pattern** by D.Alur, J.Crupi, D.Malks, "Core J2EE Patterns: Best Practices and Design Strategies", p 367-387.
7. **The Distributed Callback Pattern** by T.J.Mowbray & R.C.Malveau, "CORBA Design Patterns", p 83-89. Wiley & Sons, 1997.
8. **Asynchronous Completion Token—An Object Behavioral Pattern for Efficient Asynchronous Event Handling** by D.Schmidt, T.Harrison, and I.Pyarali. October 1996. <http://www.cs.wustl.edu/~schmidt/PDF/ACT.pdf>
9. **JMS and CORBA Notification Interworking** by S.Trythall. December 2001. O'Reilly—OnJava.com. http://www.onjava.com/pub/a/onjava/2001/12/12/jms_not.html.
10. **Notification/Java Message Service Interworking RFP.** July 2001. (document reference—eab/01-07-01). The Object Management Group.
11. **Pattern-Oriented Software Architecture—A System of Patterns** by F.Buschmann, R.Meunier, H.Rohnert, P.Sommerlad, M.Stal. Wiley & Sons, 1996.
12. **Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects** by D.C.Schmidt, M.Stal, H.Rohnert, and F.Buschmann. Wiley & Sons, 2000.

Made in Borland®. Copyright © 2002 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the U.S. and other countries. CORBA is a trademark or registered trademark of Object Management Group, Inc. in the U.S. and other countries. All other marks are the property of their respective owners. Corporate Headquarters: 100 Enterprise Way, Scotts Valley, CA 95066-3249 • 831-431-1000 • www.borland.com • Offices in: Australia, Brazil, Canada, China, Czech Republic, France, Germany, Hong Kong, Hungary, India, Ireland, Italy, Japan, Korea, the Netherlands, New Zealand, Russia, Singapore, Spain, Sweden, Taiwan, the United Kingdom, and the United States. • 13467