

# Strategies for integrating OSS/J systems with CORBA<sup>®</sup>

---

A Borland White Paper

*By Brenton Camac, Ke Jin, and Dave Stringer*

March 2004

---

**Borland<sup>®</sup>**

## Contents

Executive summary .....	1
Introduction.....	1
Invoking CORBA objects from J2EE.....	2
Selecting a Naming Service .....	2
Using the CosNaming service .....	3
Using the JNDI service.....	4
Invoking a CORBA object's operation .....	5
Sample implementation.....	5
Summary .....	6
Invoking EJB operations from CORBA .....	7
Synchronous communications .....	7
Case 1: A modifiable CORBA-based system .....	8
<i>RMI clients</i> .....	8
<i>Non-RMI clients</i> .....	8
Summary .....	10
Case 2: A sealed CORBA-based system .....	10
Summary .....	12
Asynchronous communications .....	12
Approach 1: Interworking via a JMS Bridge.....	13
<i>Translating events into messages</i> .....	13
<i>Nonstructured event fields</i> .....	15
<i>Structured Event fixed header mapping</i> .....	15
<i>Structured Event variable header mapping</i> .....	15
<i>Structured Event optional header fields and filterable body mapping</i> .....	16
<i>Structured Event remainder of body mapping</i> .....	17

*Considerations when using this approach* ..... 17

Approach 2: Interworking directly with CosNotification service ..... 18  
    *Structured Event Channel interworking* ..... 19

*Interface requirements* ..... 19

Summary ..... 21  
    *Typed Event Channel interworking* ..... 21

Summary ..... 23  
Comparison of approaches ..... 23

Appendix 1: A simplified IDL mapping ..... 25

Appendix 2: The OMG Notification Service ..... 27

    The four kinds of Notification Channel ..... 28

References ..... 31

## Executive summary

OSS/J systems are Operational Support System (OSS) components and subsystems designed for the telecommunications domain and developed for the J2EE™ (Java™) platform. By definition, an OSS must interface with other systems of an operational telecommunications environment in order to obtain information and to effect changes. For many of these existing environments, CORBA® technology is used as the means to achieve interoperability between subsystems. Consequently, for such environments, the OSS/J system requires interoperability between its J2EE host platform and the CORBA platform of the operational telecommunications environment.

This document discusses how interoperability between CORBA and J2EE platforms can be achieved for the purposes of OSS/J systems. The possible integration points between the two platforms are several, resulting from the many ways in which systems from both platforms can potentially interact with each another. This document enumerates the primary integration points and presents approaches for achieving interworking between the platforms –utilizing standardized capabilities of the platforms where available.

One of the more complicated integration points is between the CORBA Notification Service and J2EE Enterprise JavaBeans™ (EJBs™). One approach at interoperability is to use a CosNotification-Java™ Message Service (JMS) bridge, which is technically complicated. A more recent proposal is to directly subscribe EJBs to the Notification Channel. The details of each are presented and compared.

Finally, to illustrate the programming concepts introduced, functional examples are included. The examples are developed with Borland® Enterprise Server J2EE and CORBA technology but are vendor-neutral and should work with any other J2EE-compliant application server and CORBA environment.

## Introduction

The OSS through Java Initiative is a working group of industry leaders who have joined resources to speed the development of innovative OSS/BSS solutions and to facilitate faster deployment and integration of OSS components. Building on the success of J2EE technology in enterprise applications, the Initiative defines and implements an open, standard set of Java technology-based APIs that help jumpstart the implementation of end-to-end services on next-generation wireless networks and leverage the convergence of telecommunications and Internet-based solutions.

To date, the OSS/J initiative has defined many API specifications for OSS components and systems. These OSS/J-based systems typically are deployed into preexisting OSS environments, where they converse with

existing operational OSS systems in order to obtain information and to effect changes. Because many existing OSS environments use CORBA as the basis for such interworking, it is important to consider how J2EE-based OSS/J application components may interwork with CORBA-based systems.

This document identifies the primary points of interoperability between the J2EE and CORBA platforms for OSS/J systems. For each point, one or more approaches for interworking are presented along with their advantages, disadvantages, and limitations. Because this information is intended to provide the OSS/J developer with practical information on the subject, several functional code examples are also included.

The goal of this information is to provide the OSS/J developer with specific practical information on J2EE and CORBA interworking in the context of OSS/J systems. The use of these approaches is not required by the OSS/J Initiative. Indeed, the Initiative does not mandate how OSS/J APIs are to be implemented. This document is provided as a resource for OSS/J developers, to make them aware of the issues involved in order to help them implement OSS/J systems that must communicate with preexisting CORBA-based systems.

This document has two parts. The first part concerns how J2EE components, primarily EJBs, can invoke CORBA systems. The second part concerns the inverse: how CORBA-based systems can propagate information to J2EE components, specifically EJBs.

## Invoking CORBA objects from J2EE

This section examines how a J2EE application component may invoke a remote CORBA object's operations. Such a scenario might arise, for instance, when an OSS/J system must interact with an existing CORBA-based OSS system through predefined CORBA IDL interfaces.

### Selecting a Naming Service

Before invoking the operations of a CORBA object, a caller must first obtain a reference to the object. The reference is then used to locate and contact the object. In CORBA, objects are located by their Interoperable Object References (IORs). CORBA systems generally bind IORs of their objects to designated names in a Naming Service. The standardized naming service of the CORBA platform is the CosNaming service. Therefore, a standardized and hence portable way to locate a CORBA object is to obtain its object reference from the CosNaming service.

A J2EE component could use the CosNaming service to obtain a reference to a remote CORBA object. Because the CosNaming service is required to be supported by J2EE-compliant application servers,<sup>1</sup> this would be a portable approach. And even though the naming service instance supplied by the J2EE server may not be the same instance used by the CORBA system to bind its object reference, the J2EE component can use the same CosNaming interfaces to access that other instance. Locating object references using this approach is examined in more detail in “Using the CosNaming service,” below.

J2EE components can use JNDI (the native naming service of the J2EE platform) instead of CosNaming. The advantage is that they will be using the same type of naming service and therefore the same procedure to retrieve object references, regardless of whether those objects are hosted on the J2EE platform or CORBA platform. Interacting with the CosNaming service via JNDI is possible because the JNDI specification defines Service Provider Interfaces (SPIs) to allow JNDI to interwork with other popular naming and directory services (including CosNaming). The J2EE specification requires that access to the CosNaming service be available via JNDI (SPIs).<sup>2</sup> Locating object references using this approach is examined in more detail in “Using the JNDI service,” below.

Regardless of how J2EE server vendors choose to resolve these requirements in their product offering, the J2EE component can expect two naming service interfaces: JNDI and CosNaming. And because the CORBA object likely is registered in CosNaming (because it is a CORBA-based system), then the EJB has the choice of using either naming service interface. Thus, both approaches are discussed here and shown in the accompanying source code example.

### Using the CosNaming service

Traversing the CosNaming service’s namespace to locate IORs begins with obtaining a reference to the service’s root name-context. The portable way to achieve this is using the two steps as follows:

1. Obtain a reference to an ORB in the J2EE environment. The portable way to achieve this is through the component’s standard environment.

```
// File: ServiceLocator from accompanying code / example-1
javax.naming.Context jndiRootCtx = new InitialContext();
org.omg.CORBA.ORB orb = (ORB) jndiRootCtx.lookup("java:comp/ORB");
```

---

<sup>1</sup> “All J2EE products must provide a COSNaming name service to meet the EJB interoperability requirements.” (J2EE 1.3 Specification, Section 6.2.4.7).

<sup>2</sup> “... a COSNaming JNDI service provider must be available through the web, EJB, and application client containers. A COSNaming JNDI service provider is a part of the J2SE 1.3 SDK and JRE from Sun, but is not a required component of the J2SE specification.” (J2EE 1.3 Specification, Section 6.2.4.7).

2. Query the ORB for its registered Naming Service. CORBA defines a portable way to retrieve references to its various object services (e.g., Naming Service, Event Service, Transaction Service, etc.) with the ORB's `resolve_initial_references()` method. How an ORB is configured with such initial references is vendor-specific and a server administration task.

```
// File: ServiceLocator from accompanying code / example-1
org.omg.CORBA.Object ns = orb.resolve_initial_references("NameService");
org.omg.CosNaming.NamingContext cosNamingRootCtx =
    NamingContextHelper.narrow(ns);
```

Having obtained the root naming context of the desired name service, the component can now navigate the name graph to locate the desired object reference.

In the CosNaming service, the name of the binding is represented as a structured data type – the `NameComponent`. Converting between this structured form and a string form is facilitated with convenience methods. This example uses a convenience method `to_name()` to achieve this. The componentized name is then supplied to the resolve operation of a Naming Context object – typically the root context.

```
// File: ServiceLocator from accompanying code / example-1
    org.omg.CosNaming.NameComponent[] compoundName =
com.inprise.vbroker.naming.NamingUtil.to_name(nameString);
    java.lang.Object objRef = cosNamingRootCtx.resolve(compoundName);
```

This returns the object reference bound to that name in the naming service.

### Using the JNDI service

If the component is to use the JNDI naming service to lookup the CORBA object reference, then the root naming context of that service must first be obtained. The portable way to achieve this in the J2EE environment is as follows:

```
// File: ServiceLocator from accompanying code / example-1
    javax.naming.Context jndiRootCtx = new InitialContext();
```

Having obtained the root naming context to the JNDI naming service, the component can now navigate the name graph to locate the desired object reference. With the JNDI naming service, the name is resolved against the naming context using the `lookup` method with a `String` argument.

```
// File: ServiceLocator from accompanying code / example-1
    java.lang.Object objRef = jndiRootCtx.lookup(nameString);
```

This returns the object reference bound to that name in the naming service.

## Invoking a CORBA object's operation

Having obtained a reference to the remote CORBA object (represented by the variable `objRef` in the code), it must then be narrowed to the appropriate type. Narrowing the data type does not depend on which method was used to obtain the object reference. Narrowing the object reference is performed by the `narrow()` function of the Helper class generated by the IDL-to-Java compiler.

```
// File: EJB1Bean from accompanying code / example-1

    // narrow object for use
    ServiceAInterface1 corbaObjRef = ServiceAInterface1Helper.narrow(
        (org.omg.CORBA.Object) objRef);

    // invoke operation of CORBA object
    corbaObjRef.echo("test data");
```

Once narrowed, the object reference can be used as any other local object, and its methods invoked. This is shown in the code fragment above, which is based on the `testJNDI()` and `testCosNaming()` methods of `EJB1` from example 1. The `echo()` method is an operation implemented by the remote CORBA object.

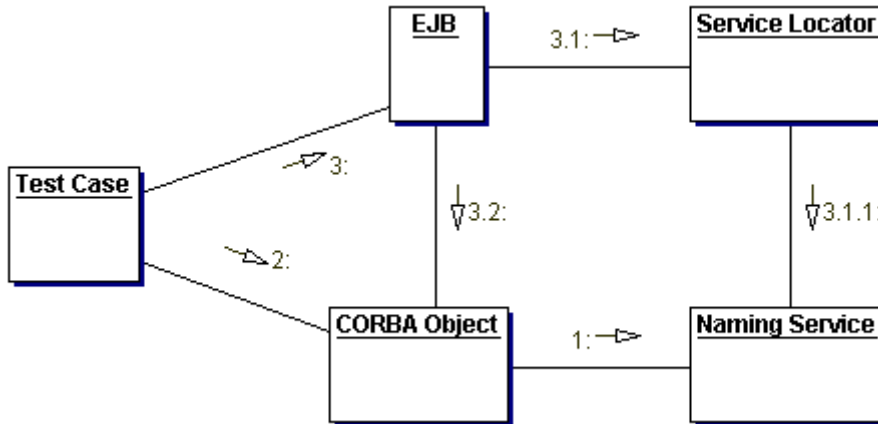
## Sample implementation

The preceding concepts are combined and illustrated in the Example 1 project [1]. This example demonstrates that a CORBA object can be registered in the CORBA CosNaming service and subsequently located by an EJB either through the JNDI or CosNaming services. Once located, the EJB is able to invoke operations of the CORBA object. The supplied test case exercises the EJB to use both JNDI and CosNaming services to retrieve the CORBA object reference.

The example consists of a CORBA server (`ServiceAInterface1`), which offers an operation “echo” that returns the string data it is supplied. When running the supplied test case, the following steps are performed.

Step 1. When the CORBA service is started, it first locates the CosNaming service. There are several mechanisms for discovering a CosNaming service, both standard and proprietary. Typical of the standard mechanisms is to supply the CORBA server with a URL indicating the CosNaming service location on the

network: e.g., corbaloc::localhost:683/NamingService. Once the service is located, the CORBA server registers a CORBA object reference with it.



**Figure 1:** Steps taken when exercising Example 1's test

Step 2. testCORBASvcExists is run by the JUnit conformant test case (TestCase1). This is a precondition test for the two that follow (see Step 3) and ensures that the CORBA object is available for them.

Step 3. testJNDI and testCORBA follow the same execution path: the EJB testJNDI() or testCORBA() method is invoked, depending upon the test case.

Step 3.1. The EJB contacts a utility class (ServiceLocator) to look up the object reference either through JNDI or CosNaming as directed.

Step 3.1.1. The Service Locator contacts the naming service (either through JNDI or CosNaming protocols) to retrieve the requested object binding.

Step 3.2. The EJB narrows the returned object reference and then invokes the echo operation of the CORBA object. If the data returned from the echo operation is the same as what was supplied, then the EJB method returns true, otherwise false.

## Summary

This section examines how a J2EE component can locate and invoke the operations of a CORBA object. Because the J2EE component can use either naming service of the J2EE platform (JNDI or CosNaming), both approaches are examined. Although each approach uses different techniques, both achieve the same result – obtaining an object reference to a remote CORBA object. To illustrate these concepts, fragments of source code taken from complete examples (see [1]) are included.

The choice of which approach to use depends on various factors. In a pure EJB environment, the JNDI approach might be preferred because it uses techniques already commonly used in that programming environment to locate other objects (e.g., other EJBs, etc). However, if existing client-side CORBA code is available and is to be reused in the EJB, then the CosNaming approach may be favored because it would require fewer changes to that code. Another consideration that might influence the choice of approach is which Naming Service instance hosts the sought-after object reference. If an existing CORBA application is using its own Naming Service to store object references, and if this naming service is not available to EJBs through the JNDI service of the EJB container (i.e., the container is not configurable to work with foreign naming services), then it might be necessary for the EJB to use the CosNaming approach and explicitly connect to the foreign Naming Service.

The ability for J2EE components to use standard means to interoperate with CORBA objects and services this way is possible because the J2EE platform explicitly requires support for the CORBA CosNaming service and RMI-IIOP™ protocol in J2EE-compliant application servers.

## Invoking EJB operations from CORBA

This section examines how an EJB's operations might be invoked from a CORBA-based system. Two modes of interaction are considered. The first mode is direct (synchronous) communications, in which a CORBA-based system directly invokes an operation of an EJB's remote interface. The second mode is indirect (or, more precisely, asynchronous) communications, in which a CORBA-based system propagates events to the EJB container through a messaging system. For both modes, more than one approach can be taken; these are presented and compared.

### Synchronous communications

This section examines various approaches whereby a CORBA-based system can directly invoke the methods of an EJB. In the context of OSS/J systems, this arrangement may arise, for instance, under the following situations:

1. An existing CORBA-based system is to be extended to interact with an OSS/J system's EJBs. In this case, the interface being used is defined by the EJB and therefore involves RMI semantics. Because many existing CORBA systems are implemented in C++ as well as Java, both Java and non-Java scenarios are examined.
2. An OSS/J system must register a callback interface with an existing CORBA-based system. In this case, the callback interface is defined by the CORBA-based system in IDL.

**Case 1: A modifiable CORBA-based system**

This case examines how a CORBA-based system can be extended to interact with an EJB. Such might occur, for instance, when an existing CORBA-based system is modified to invoke the operations of an OSS/J system. Because the interaction between the two systems is according to the EJB's interface, that interface must be mapped to the local implementation environment of the CORBA-based system.

**RMI clients**

If the CORBA system is implemented using Java, then it is sufficient to create RMI-IIOP stubs from the EJB interface. This can be done using tools such as the Java2 SDK "rmic" (with `-iiop` option) or the Borland® VisiBroker® "java2iiop" utility. An illustration of this is shown as example-2 in the accompanying source code, which demonstrates a CORBA client invoking the methods of an EJB's Remote interface. The CORBA system can then use the generated stubs to invoke operations of the EJB.

**Non-RMI clients**

If the CORBA system uses C++, then the interaction becomes more complex because the EJB's (Java) RMI semantics and its native language objects (Collections, etc.) must be expressed in C++. One approach to achieve this is to map the EJB interface first to language-neutral IDL and then from IDL to C++. Both mappings are defined by the OMG, and tools exist to perform this automatically (VisiBroker has `java2idl` and `idl2cpp` compilers).

An illustration of this is in Example 2. The remote interface of the EJB (EJB2) is shown below.

```
// File: EJB2 from accompanying code / example-2
package ossj.example2.myEJBapp;

public interface EJB2 extends javax.ejb.EJBObject {

    public String echoString(String s) throws RemoteException;

    public String[] echoStrings(String[] ss) throws RemoteException;

    public Date echoDate(Date d) throws RemoteException;
}
```

Using the JavaIDL mapping ([2]) to translate this Java interface to IDL produces the following IDL definition. (Some elements have been removed for brevity).

```

...
module org {
  module omg {
    module boxedRMI {
      module java {
        module lang {
          valuetype seq1_String sequence< ::CORBA::WStringValue >;
          ...
        };
      };
    };
  };
};

module java {
  module lang {
    custom valuetype _Exception : ::java::lang::Throwable {
    };
    ...
  };
  ...
  module util {
    custom valuetype Date : ::java::lang::Comparable
    supports ::java::lang::Cloneable {
    };
    ...
  };
};

module ossj {
  module example2 {
    module myEJBapp {
      interface EJB2 : ::javax::ejb::EJBObject {
        ::CORBA::WStringValue echoString (in ::CORBA::WStringValue arg0);
        ::org::omg::boxedRMI::java::lang::seq1_String echoStrings (in
          ::org::omg::boxedRMI::java::lang::seq1_String arg0);
        ::java::util::Date echoDate (in ::java::util::Date arg0);
      };
    };
  };
};

```

The non-Java client of this interface would need to provide implementations for the custom valuetypes Date and \_Exception in this instance. Other valuetypes used in the interface, such as seq1\_String, are defined in terms of existing IDL constructs (e.g., sequence< ::CORBA::WStringValue >).

Consequently, this approach of mapping an EJB interface to IDL and then to an implementation language typically requires that the client provide implementations for some of the ValueTypes. In addition, it also results in complex interfaces, even for EJBs with relatively simple interfaces. The complexity arises because the OMG-defined mappings attempt to accommodate as many features of the EJB interface as possible.

For a discussion of a nonstandardized approach to the problem of mapping EJB interfaces to IDL, see Appendix 1: “A simplified IDL mapping”.

## Summary

For CORBA systems implemented in Java, integration with EJBs is achieved using commonly available RMI-IIOP compilers such as `rmic` and `java2iiop`. For CORBA systems implemented in C++, mapping the EJB interface to the target language requires more effort. A “brute-force” approach of transforming the Remote interface to IDL and then to C++ is possible but results in complex C++ constructs that must be implemented by the client and manipulated by the client at runtime. Typically, such mapped interfaces are quite complex because they attempt to accommodate all of the semantics of the original EJB (RMI) interface. If the EJB’s interface is relatively simple, then such a comprehensive mapping may not be required. In such cases, a “simplified” Java-to-IDL mapping might be considered. Such a scheme is nonstandard and is offered by at least one vendor (see Appendix 1: “A simplified IDL mapping”).

## Case 2: A sealed CORBA-based system

This section examines the case in which an existing interface (defined in IDL) is to be implemented by an EJB. Such might occur, for instance, if an OSS/J system is to receive callbacks from existing CORBA-based system. In this case, the callback interface is defined by the CORBA-based system (in IDL) and the EJB must implement it in order to be contacted by the CORBA-based system.

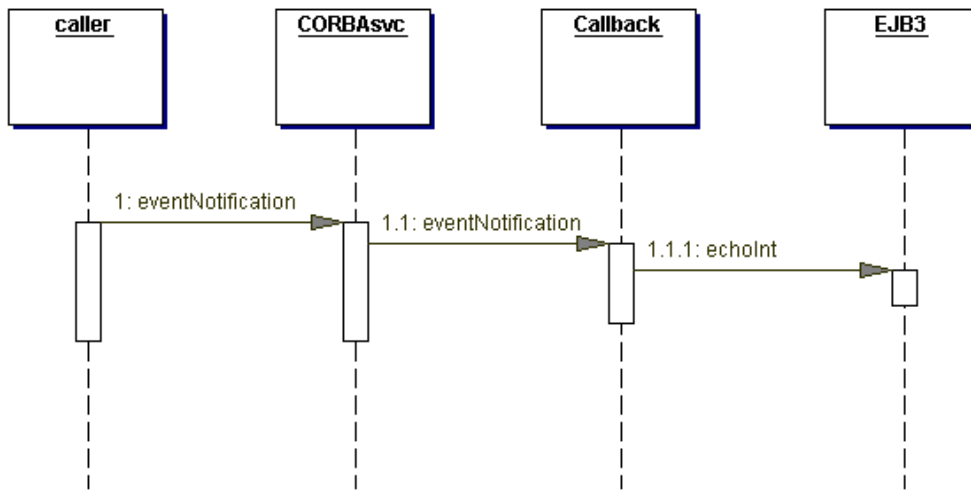
The task then is for the EJB to provide an implementation of the IDL interface. The IDL can be mapped to Java using standard language mapping [4]. However, this cannot be implemented directly because an EJB Remote interface is given RMI semantics [5]. Because the caller using the original IDL would not be using RMI semantics, the invocations would fail.

This issue can be resolved by using a proxy object to add necessary RMI semantics. The proxy’s implementation is based on the IDL-to-Java language mapping [4]. Invocations it receives through that interface are propagated to the target EJB by stubs compiled from the EJB’s interface.

An illustration of this is shown in Example 3. There, an IDL interface (with name `Callback`) is used by a CORBA-based service to notify an interested party of events. The CORBA system has another interface (`CORBASvc`), which exposes two operations: `registerCallback()`, which allows an object implementing the `Callback` interface to be registered in order to receive future callbacks; and `eventNotification()`, which contacts the registered callback object via its `eventNotification()` method for testing purposes:

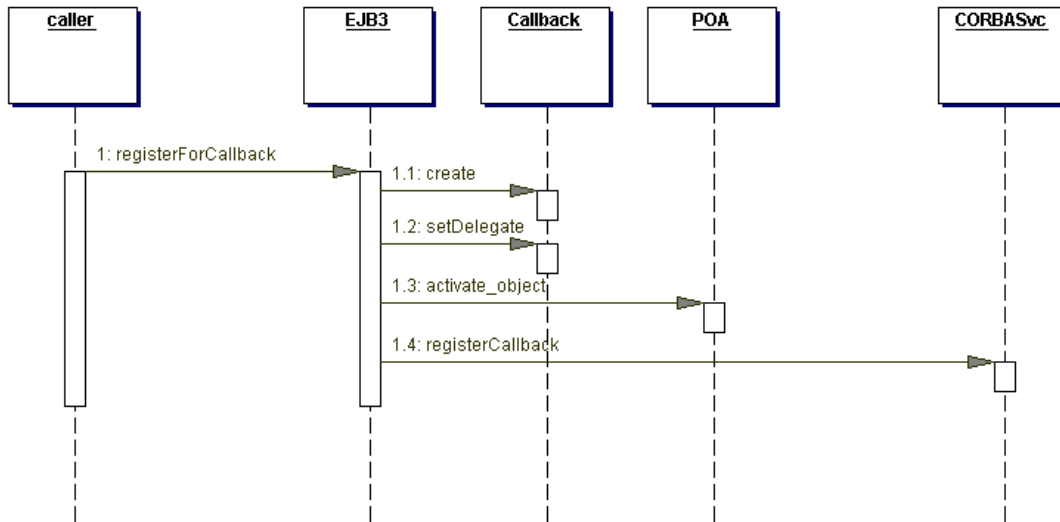
```
module MyCORBASystem3 {  
    interface Callback {  
        long eventNotification(in long param1);  
    };  
  
    interface CORBASvc {  
        void registerCallback(in Callback c);  
        long eventNotification(in long param1);  
    };  
};
```

Once the configuration has been initialized, invocations on the CORBAsvc's eventNotification() method will propagate to the registered Callback object which in turn propagates to the EJB. The EJB3's implementation of echoInt() is to return the value supplied it. In this way, the test driver (caller) can confirm that the call to the CORBAsvc has successfully completed its round trip. This is depicted below.



**Figure 2:** Testing the configuration of Example 3

To produce this configuration, the following steps are taken in the example. (For convenience, the proxy object is collocated with the EJB in the EJB container). When the EJB (EJB3) is initialized, it creates a POA to host the callback object. Then when EJB3's registerForCallback() method is invoked, it creates a local instance of the Callback object, configures it to delegate its calls to this EJB3 instance, registers the object with the POA so it can be contacted by remote CORBA clients, and then contacts the CORBAsvc to register it as a Callback object. The methods used are shown in the sequence diagram below. This sequence of activities is initiated by the test client TestCase3 during its setup phase.



**Figure 3:** *Configuring EJB3 to receive callbacks from CORBASvc*

## Summary

EJBs can be integrated with existing CORBA services through preexisting IDL interfaces. EJBs require callers to use RMI semantics that typically will not be satisfied by existing pure CORBA systems. One approach to resolve this is to use a proxy that implements an IDL interface and delegates its invocations to a configured EJB using the required RMI semantics.

## Asynchronous communications

Many OSS/J services require interaction with existing underlying operational systems. For some OSS/J systems, the interaction will be the receiving of events produced within those existing systems. For instance, the OSS/J Quality of Service (QoS) system is interested in alarm events and threshold triggers occurring in underlying OSS systems in order to fulfill its service to its clients.

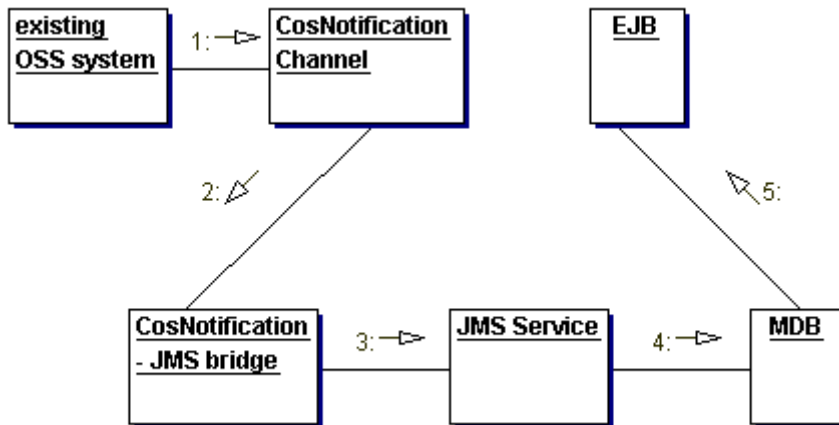
The majority of these existing operational systems use the CORBA CosNotification service<sup>3</sup> to propagate events. (The CosNotification service, explained in more detail in Appendix 1, is the CORBA service that provides asynchronous event propagation. CosNotification is used widely in this domain with many telecom standards (e.g., recommendations ITU-T X.780 and TMF 814A) defining how telecom-domain specific events are to be expressed as CosNotification events. Because there are no equivalent standards,

recommendations or tools devised for alternative messaging technologies (e.g., JMS) it is reasonable to assume that OSS will continue to use the CosNotification service and that OSS/J systems will therefore need to integrate with CosNotification in order to interact with underlying OSS systems.

The problem confronting OSS/J systems then is how their EJB components may receive events from a CosNotification Channel. Because the J2EE platform possesses its own asynchronous messaging service – JMS – one approach is to map CosNotification events into JMS messages and then deliver these to an EJB via a Message Driven Bean (MDB). A proposal for this approach is discussed below. Another approach is for an EJB to be directly subscribed to the notification channel as an event consumer. This also is discussed below, as well as a comparison of the relative merits and limitations of each approach.

### Approach 1: Interworking via a JMS Bridge

This approach considers a bridge that links the CosNotification service with JMS and translates events from one domain to the other. The bridge thus acts as an event consumer to the Notification service and a message supplier to JMS. Such an interworking service is currently being discussed within the OMG as part of a general purpose JMS/CosNotification Interworking Service [6].



**Figure 4:** *Interworking via the CosNotification-JMS Bridge*

#### Translating events into messages

According to the proposal [6], information in a CosNotification StructuredEvent would be mapped into a JMS message using the following general formula.

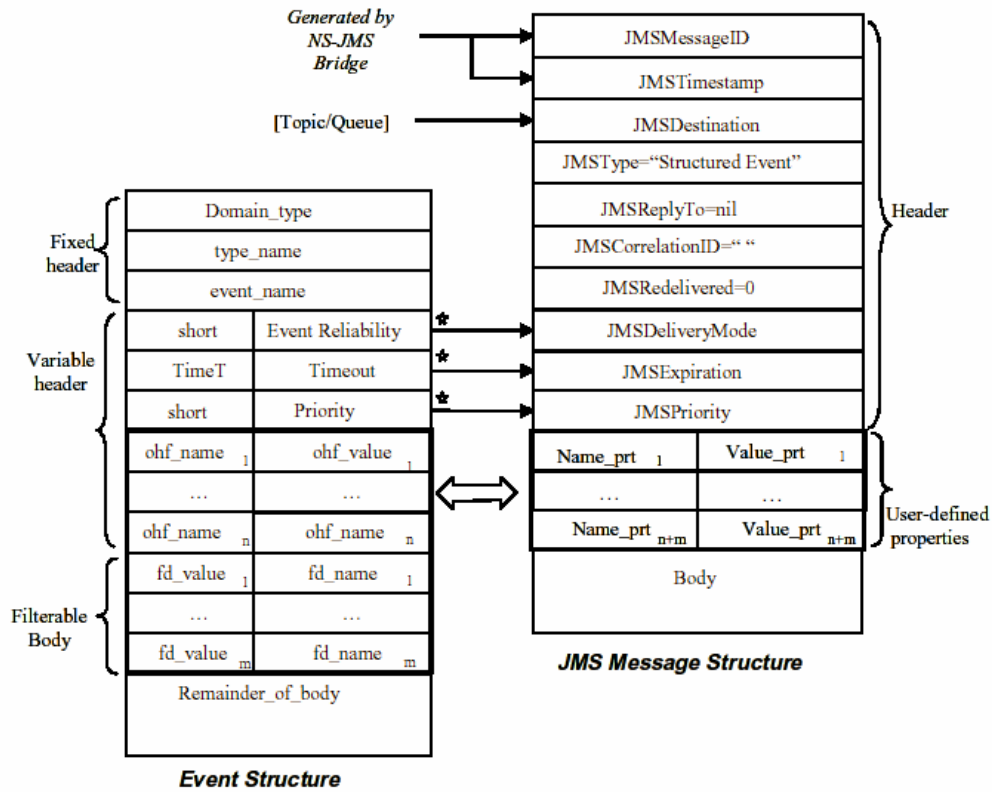
<sup>3</sup> The use of the CosNotification service in the telecom domain should not come as a surprise; it was the telecom membership of the OMG who were the main force behind the proposal and definition of an event service for CORBA.

If the event's variable\_header standard optional part is present (i.e., Event\_Reliability, Timeout, and Priority fields), then these are mapped to corresponding fields of the JMS message header. If the event supplier does not define these fields, then the JMS header fields are populated using default values as specified in the JMS specification.

The event's fixed\_header, the rest of the optional header fields, and its filterable body are placed in JMS properties fields. These will therefore be seen by the JMS receiver application as user-defined fields.

The translator populates fields in the JMS message that have no counterpart in the event domain, such as JMSDestination and JMSMessageID.

The event's remainder\_of\_body section is mapped to a JMS message body type depending on the type and complexity of the data. Conceptually, this translation is depicted as follows.



**Figure 5:** Structured Event-to-JMS Message mapping

This translation formula is described in more detail below.

### Nonstructured event fields

Values for the JMS message fields `JMSMessageID`, `JMSTimestamp`, `JMSDestination`, and `JMSType` are created by the JMS-NS Bridge. Specifically:

- `JMSMessageID` is a `String` value that should be a unique key, prefixed by “ID.” The exact scope of uniqueness is not defined.
- `JMSTimestamp` field contains the time the message was submitted to JMS for sending. It is in the format of a normal Java millisecond time value.
- `JMSDestination` contains the Topic or Queue name to which the message is being sent.
- `JMSType` is a `String` value that should be set to “Structured Event”.

### Structured Event fixed header mapping

The Structured Event’s fixed header fields are mapped to user-defined property fields of the JMS message. (The name for these properties must obey the rules for a message selector identifier<sup>4</sup> specified in Section 3.8.1.1 of the JMS Specification [7]). Specifically:

- The `domain_type` event header field is mapped to a new JMS user-defined property whose name is “`$domain_type`” and whose value is the event’s `domain_type` field converted to `java String`.
- The `type_name` event header field is mapped to a new JMS user-defined property whose name is “`$type_name`” and whose value is the event’s `type_name` field converted to `java String`.
- The `event_name` event header field is mapped to a new JMS user-defined property whose name is “`$event_name`” and whose value is the event’s `event_name` field converted to `java String`.

### Structured Event variable header mapping

If the Structured Event’s variable header contains the fields `EventReliability`, `Priority`, or `Timeout`, then these are mapped to `JMSDelivery`, `JMSPriority`, and `JMSTimetolive`, respectively. If these

---

<sup>4</sup> An identifier is an unlimited-length character sequence that must begin with a Java identifier start character; all following characters must be Java identifier part characters. An identifier start character is any character for which the method `Character.isJavaIdentifierStart` returns true. This includes “\_” and “\$.”

---

fields are not defined in the event, then the following values are used: `JMSDelivery` is set to `PERSISTENT`; `JMSPriority` is set to 4; and `JMSTimeToLive` is set to `Unlimited`.

#### **Structured Event optional header fields and filterable body mapping**

Optional header fields (`ohf_*`) and filterable data fields (`fd_*`) of the Event are mapped to user-defined JMS properties with names `ohf_*` and `fd_*`, respectively. These JMS property names must obey the rules for a message selector identifier specified in Section 3.8.1.1 of the JMS specification. The content of the `ohf_*` or `fd_*` field is converted to a Java primitive data type.

If the optional header field or filterable data field is a complex IDL type, then multiple JMS properties are used, one for each primitive element of the complex type. The complex structure is decomposed into the individual JMS properties as follows:

- The new `ohf_*` or `fd_*` field name is the concatenation of the event field name, the structure name(s), and the member of the structure name(s). The structure member operator “.” is used to delimit each name component in the concatenation. When a sequence structure is encountered, then the ordinal value of the particular element is used as its name (e.g., 1, 2, 3, etc.). This process is repeated if the structure contains nested data structures as elements along with primitive elements.
- The content of the decomposed field is converted to a corresponding Java primitive data type.

For example, a structured event containing a field with name `<Fd1>` and value of type `<CORBA::Any>` which encapsulates an IDL struct named `Alarm { string AlarmName; int Severity }`, is transformed into two JMS user-defined properties: `<${Fd1}.AlarmName, String>` and `<${Fd1}.Severity, Integer>`.

Another example is a structured event containing an optional header field with name `<Ohf1>` and value of type `<CosNaming::Name>` (defined a sequence of `CosNaming::NameComponent`) whose content is `NC1/NC2/NC3`. This is transformed into the JMS user-defined properties: `<${Ohf1}.Name.1.id, String>`, `<${Ohf1}.Name.1.kind, String>`, `<${Ohf1}.Name.2.id, String>`, `<${Ohf1}.Name.2.kind, String>`, `<${Ohf1}.Name.3.id, String>`, `<${Ohf1}.Name.3.kind, String>`.

If the optional header or filterable data field has other complex data types, it is mapped to a byte stream. The JMS client will then need to use the appropriate CORBA Helper classes to unmarshal this user data.

### Structured Event remainder of body mapping

The Structured Event's remainder\_of\_body is of type CORBA::Any. Its contents are mapped to one of the five JMS message body types (Text, Stream, Map, Bytes, or Object) depending on the complexity of the field's encapsulated data type. When the Event's remainder\_of\_body Any type involves:

- IDL basic type elements – the elements are mapped to a Java primitive type using standard IDL-to-Java mapping. The resulting set is packaged in a JMS StreamMessage body.
- String type element – only the element is mapped to a java String type and packaged in a JMS TextMessage body.
- Sequence of Properties (PropertySeq) – the element is mapped to a set of name-value pairs where names are Strings concatenated with the ordinal value of the element in the sequence, and values are Java primitives. This is packaged in a JMS MapMessage body.
- Octet Sequence element or other type (e.g., user constructed type) – the element is mapped to a JMS BytesMessage body. In order to reconstruct the IDL type, the JMS client then would receive a BytesMessage and would use the appropriate CORBA Helper class.

### Considerations when using this approach

An important characteristic of this approach is the decomposition or “flattening” of the structured event's nonprimitive data type fields into a series of JMS properties in the counterpart JMS message. The need for decomposition arises because the Java Messaging Service allows only primitive data types to appear in user-defined properties. However, the use of nonprimitive data types in CORBA and in the telecom domain is common. Examples include the IDL structure TimeBase::UtcT (see ITU-T X.780) and the IDL sequence CosNaming::Name (see TMF 814A). Therefore, applying this approach to the telecom domain results in message consumers frequently needing to deal with “flattened” structured information.

Flattening this information into JMS message properties allows the JMS message to be filtered by the JMS service in a way similar to the original structured event being filterable in the CosNotification domain. However, there are many disadvantages to flattening the structured event information, including the following:

- a) Substantial programming effort is required to reassemble the event information in the consumer. Because this information is no longer represented explicitly by a structured data type in the JMS message but implicitly in the names of JMS properties, JMS message consumers must formulate property names (in dotted notation) in order to navigate and retrieve event information. Such an approach is less type-safe than using first-class data structures of a language (e.g., those produced by

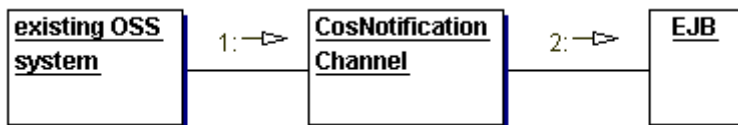
an IDL compiler). With no automated, compile-time checking to ensure that property names are formed correctly, there is potential for programming errors that will not become evident until runtime.

- b) Algorithms and designs from existing CORBA-based event consumers cannot be reused in the JMS consumer unless the event is first reformatted into its original structure. Using existing algorithms and designs to interpret events can significantly decrease development costs and improve code quality because such artifacts have already been developed and used and therefore more likely will have defects identified and corrected.
- c) JMS message consumers likely will need to access CORBA Helper classes as well, even though the event's filterable data has been decomposed by the translation process. This is because user-defined structures in the event's `remainder_of_body` field are not decomposed during the translation process as filterable data is. Hence, the JMS message consumer will typically incorporate both CORBA Helper classes to access nonfilterable event data as well as custom code to access filterable event data; i.e., it will contain code to work with two technology domains – JMS and CosNotification. Such requirements make the resulting consumers complex and cause much of its implementation to be duplicated.

A final consideration with this approach is that it relates only to Structured Events, not Typed Events (events from the Typed Event Notification Channel). As discussed in the section “Typed Event Channel Interworking,” Typed Channels are much more efficient and effective at event propagation, which is why some telecom domain standards (e.g., ITU-T/T1M1) recommend their use. To employ this approach with Typed Events would require that they first be converted into Structured Events before being translated to JMS. Such an arrangement adds yet another level of translation, effort, and subsequent administration.

### Approach 2: Interworking directly with CosNotification service

Another approach whereby EJBs can receive events from a CosNotification channel is to register EJBs with the channel as event consumers. In this approach, CosNotification events are delivered directly to EJBs.



**Figure 6:** *Interworking with the CosNotification Channel*

The following sections discuss how this is achieved for the Structured Event and Typed Event Channels.

---

### Structured Event Channel interworking

For EJBs to receive events from a Structured Event Channel,<sup>5</sup> they implement the `CosNotification` interface for a structured event consumer and subscribe to the Notification Channel as an event consumer. In this way the Notification channel treats the EJB as an event consumer and delivers events to it directly.

### Interface requirements

Two kinds of Structured Event consumers are defined by the Notification service: a `PullConsumer` and a `PushConsumer`. A `PullConsumer` is one that polls the Notification Channel for events; a `PushConsumer` is one that is sent events from the Notification Channel. The EJB as an event consumer should implement the `PushConsumer` interface<sup>6</sup> in keeping with the intention of the EJB specification, which desires that EJBs be passive targets of operations rather than self-initiated actors of activities. Being a Push Consumer requires that the EJB declare and implement the `push_structured_event()` operation through which events will be received from the channel. The IDL for this interface is shown below.

```
// 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();
    };
    ...
};
```

This IDL interface and associated data type (`StructuredEvent`) can be mapped to Java via the IDL-to-Java language mapping [4]. Note, however, that the `StructuredPushConsumerOperations` interface produced by the mapping is not an RMI interface (its declared methods do not declare the `java.rmi.RemoteException` exception in their throws clause), and therefore is not suitable for being extended by the EJB's Remote interface (although it could be extended by its bean interface). Hence, the EJB must explicitly declare the `push_structured_event()` operation itself in its Remote interface. This is shown below.

---

<sup>5</sup> The Structured Event Channel is explained in more detail in Appendix 2.

<sup>6</sup> Although an EJB could choose the `PullConsumer` interface instead, this would require that it be proactive in periodically polling the channel for events. The J2EE specification discourages such autonomous behavior.

```
// EJB StructConsumer Remote Interface
public interface StructConsumer extends javax.ejb.EJBObject {

    public void push_structured_event(
        org.omg.CosNotification.StructuredEvent event )
        throws java.rmi.RemoteException, org.omg.CosEventComm.Disconnected;

    public void disconnect_structured_push_consumer()
        throws java.rmi.RemoteException
}

```

Note also that the `push_structured_event()` operation declares an argument of type `org.omg.CosNotification.StructuredEvent`. The definition of this data type is produced by the IDL-to-Java mapping. Because it is used within a method of an RMI interface, it will inherit RMI semantics, one of which is to allow the data type to be null. This concept is represented in IDL by “boxing” the data structure in a valuetype, and is expressed in IIOP by passing the data structure using the Object-By-Value mechanism.

```
module org {
  module omg {
    module boxedIDL {
      module org {
        module omg {
          module CosNotification {
            valuetype StructuredEvent ::CosNotification::StructuredEvent;
          };
        };
      };
    };
  };
};

```

This fact is important because it means that the EJB container is expecting callers of this method to supply their IIOP argument as type

`::org::omg::boxedIDL::org::omg::CosNotification::StructuredEvent`. If the caller derives its stubs from the EJB’s Remote interface, then its invocations will conform to this requirement because they will be using RMI-IIOP. However, if the caller is using the interface found in `CosNotification.idl` for event suppliers, as would be typically be the case for preexisting CORBA-based systems, then such requirements will not be met, because that interface does not declare that the `StructuredEvent` is to be boxed in a `ValueType`. This is a special case of the same general problem encountered earlier in the paper, when synchronous calls from CORBA-based systems to EJBs were discussed (see Case 1). A solution was to introduce a proxy to provide the required RMI semantics. Fortunately, in this instance, the operation and data type to be translated are known *a priori* to the Notification Channel. This means that the Notification Channel itself could box the argument if the consumer is determined to be using RMI-IIOP, and deliver the same event unmodified (unboxed) to consumers who have directly implemented the `CosNotification` IDL. This is the approach taken by the

Borland® VisiNotify™ implementation of the CORBA Notification Channel. The alternative of providing an IIOP/ RMI-IIOP proxy between the EJB and Notification Channel, based on what was discussed in earlier sections, is also an option.

An illustration of the approach whereby the Notification Channel acts as the proxy to supply RMI semantics is shown in Example 4 of the accompanying code. It consists of :

- `EJB4` – an EJB that can be subscribed to the Notification Channel as an event consumer
- `StructPushConsumer` – a CORBA-based event consumer, which can also be subscribed to the channel
- `StructPushSupplier` – an event supplier that produces events based on simplified definitions of selected ITU-T TMN notifications (from the ITU-T X.700 recommendation series)

This example shows that ITU-T TMN notifications can be propagated by a `CosNotification` and directly received by both a CORBA-based consumer as well as an EJB-based consumer without the need for custom proxies.

## Summary

Subscribing EJBs directly to the Structured Event Channel as event consumers is possible if the EJB implements the necessary operations. The RMI semantics of the EJB's Remote interface require that the event supplier push events that are boxed. Because the event consumer's interface is known *a priori* to the Notification channel, the channel can perform this value-boxing of those events that are to be delivered to RMI-based event consumers. This capability is demonstrated through Example 4 using the Borland implementation of the `CosNotification` service – VisiNotify.

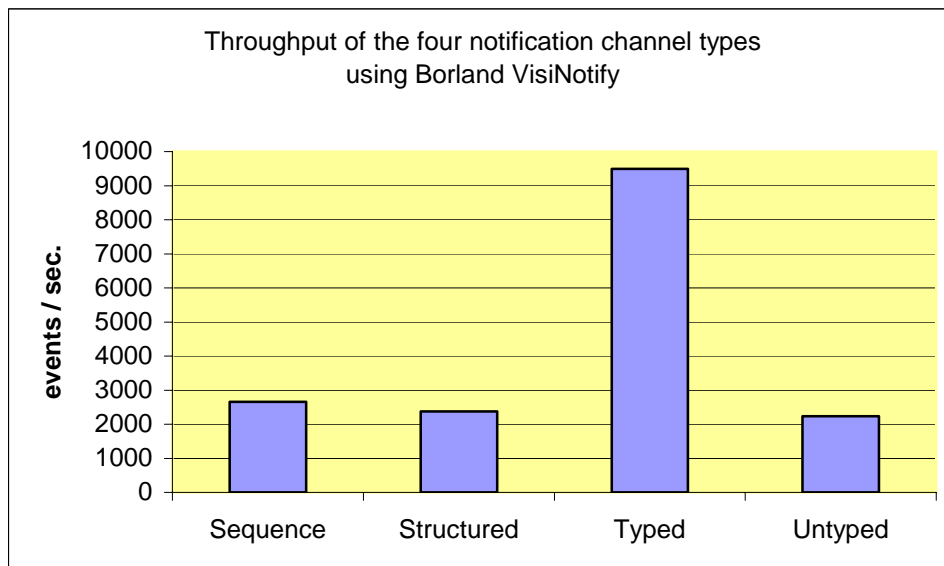
## Typed Event Channel interworking

The `CosNotification` service defines four kinds of channel: the Untyped Event Channel, the Structured Event Channel, the Sequenced Event Channel, and the Typed Event Channel. The Typed Event Channel differs from the other kinds of channels in that it does not use a data structure defined by the `CosNotification` service to represent events, nor does it use predefined IDL operations such as `push()` or `push_structured_event()` to transfer events between client and channel. Instead, suppliers propagate events to consumers via typed channels using application-defined operations and data structures. The Notification specification refers to these interfaces used by typed suppliers and consumers as <I> interfaces.

Using Typed Event channels has many advantages over using the other forms of event (notification) channels. Arguably the most significant advantage is that applications communicate via strongly typed application-level interfaces. In fact, the events being exchanged can be declared directly in IDL as data structures along with the operations used to convey them. In contrast, a Structured, Sequenced, or Untyped channel uses a generic event type with predefined operations. This flexibility allows compile-time checking of typed suppliers and consumers to ensure that the events they will be exchanging will be compatible, which is not possible with the other types of channel other than asserting that clients correctly exchange a Structured Event data type for instance. Support for compile-time checking results in detecting errors before operational use and does not rely on testing to expose such errors.

Another benefit of using Typed Event channels, related to the previous benefit, is that clients require less code to parse event structures. Parsing data types that have statically defined structures is simpler than parsing structures that use meta-level descriptions such as property lists that contain name/value pairs. Consequently, parsing in Typed Event channel clients requires less code, and therefore fewer code defects are likely.

Because clients of typed event channels spend less time parsing event structures at runtime, and typed events comprise smaller messages, there is the potential for greater throughput, scalability, and overall efficiency. Examples of this compared to the other forms of event channels are shown below. These tests used the Borland implementation of the CosNotification service – VisiNotify version 5.2 executing on a 700 MHz Intel® Pentium® III running Windows® 2000, with one local supplier and four local consumers.



**Figure 7:** Comparing throughput of different notification channel types

For these reasons, ITU-T/T1M1 recommends using typed events.

One restriction with the Typed Event channel is that the <I> interface used by both suppliers and consumers must match. Therefore, if EJBs are to be subscribed as typed consumers to the channel, then the suppliers will need to implement the RMI Remote interface of the EJB. This is the same issue faced by CORBA clients contacting EJBs discussed previously in Case 1: “A modifiable CORBA-based system”. The solutions discussed there can also be applied to this environment. EJBs that need to receive events from non-RMI suppliers could either use a proxy as described in Case 1, or a simplified mapping of the EJB interface could be exposed to the supplier and subscribed to the channel.

Example 5 illustrates an EJB being subscribed to a Typed Event channel.

### **Summary**

The Typed Notification channel offers a more strictly typed environment for event suppliers and consumers to use. The Typed channel typically also provides the best performance of the four kinds of Notification channel. For these reasons the ITU-T/T1M1 recommends their use. Interfacing CORBA systems with EJBs via the Typed channel faces the same challenges as when these two domains are interfaced for synchronous communication. Solutions to those problems (as discussed in Case 1) are therefore applicable here, too.

### **Comparison of approaches**

Two approaches were examined by which EJBs can receive events from the CosNotification service. The first approach bridges the CosNotification and JMS services, translating events from one domain to the other, thus delivering the events as JMS messages to an MDB and then to an EJB. The most significant challenge faced when using this approach is the translation employed for structured data, particularly structured data types defined by various telecommunications recommendations. Under this approach, parts of the event structure must be flattened into property lists of name value pairs. The disadvantages associated with this practice are enumerated in the previous section, “Interworking via a JMS Bridge”.

The other approach is for an EJB to implement the CosNotification event consumer interface and be registered directly with the Notification channel so that events can be delivered directly to the EJB. This requires that an EJB possess the operations of the Structured Consumer interface, and requires that either the Notification Channel or some intermediary proxy box events before being delivered to the EJB. With this approach, the event does not need to be flattened, and because no content translation is involved, the EJB code required to navigate and parse the event data type is very similar (if not identical) to that found in existing CORBA consumers. The benefit of this is that code reuse is more likely, or at least the design can be reused, thus leading to more confidence in the code quality.

For these reasons, the second approach to interworking EJBs with the Notification Service is considered more suitable for interfacing EJBs with the CosNotification service.

## Appendix 1: A simplified IDL mapping

As discussed in Case 1 “A modifiable CORBA-based system” mapping the Home and Remote (RMI) interfaces of an EJB to non-Java languages (e.g., for use by non-Java clients of the EJB) can result in complex and complicated interfaces. Much of this complexity arises from trying to accommodate as many of the RMI semantics as possible in the resulting IDL and ultimately in the target implementation language. However, if the EJB’s Home and Remote interfaces are relatively simple, then such a comprehensive mapping (as specified by CORBA) is not necessary. In such cases, a trade-off can be made whereby some semantics of the RMI interface are not fully translated into the IDL and target language in order to simplify the resulting IDL, resulting in a simplified mapping of Java to IDL.

Such a simplified mapping is not standardized by the OMG but is commercially available with the Borland Enterprise Server from Borland Software Corporation. Details of the assumptions made are explained in an example [3] included with the product.

As provided by that product, the simplified IDL mapping is able to map Java constructs to simpler IDL forms as compared to the regular Java-to-IDL mapping because it makes certain assumptions about what are acceptable semantics. For example, SIDL maps the data type `java.util.Date` to a structure containing an integer value representing milliseconds since Jan. 1, 1970 GMT. Java Collections, Enumerations, Lists, and Vectors are all mapped directly to sequences of Any; and `java.lang.Class` is mapped to an IDL struct containing two strings – one being the name of the associated Java type and the other being the name of the associated IDL type. Such mappings convey the majority of the original data type’s meaning for most uses and are implemented by an accompanying library. The simplified IDL generated for EJB2 is shown below.

```
#include <sidl.idl>

typedef sequence<wstring> Sequence_of_Wstring;

module ossj {
  module example2 {
    module myEJBapp {
      interface EJB2 : ::sidl::javax::ejb::EJBObject {
        ::sidl::java::util::Date
          echoDate(in ::sidl::java::util::Date arg0);
        wstring echoString(in wstring arg0);
        ::Sequence_of_Wstring echoStrings(in ::Sequence_of_Wstring arg0);
      };
      ...
    };
  };
};
```

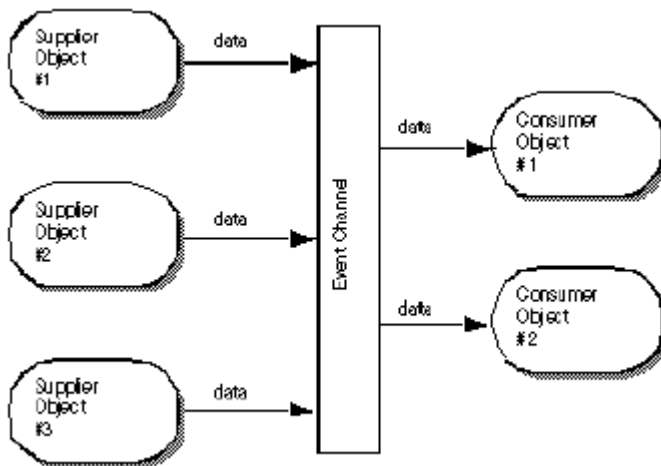
The SIDL scheme thus presents a simplified IDL interface of an EJB. The scheme does not require that the EJB be modified or adapted, and it is deployed to the Borland Enterprise Server's EJB container in the same way. Once deployed, the EJB container offers the usual RMI-IIOP interface to the EJB based on the CORBA standard mapping of Java to IDL, and also (if configured for an EJB) offers another RMI-IIOP interface based on the simplified IDL mapping of the EJB. Then at runtime, when a method of the EJB's simplified interface is invoked, the Borland Enterprise Server EJB container translates the simplified types used into the appropriate Java types required by the EJB before dispatching the invocation to the EJB, and performs the inverse translation for invocation replies. Because the EJB container performs the translation to and from simplified constructs, the EJB itself need not be altered to participate in this environment, with the caveat that the EJB's interface is expressible using simplified IDL. Similarly, the client operates as a standard CORBA client invoking a CORBA-compliant IDL, albeit a simplified one.

Thus, this approach to interworking non-Java callers with EJBs takes much of the complexity that previously would have existed in the client application and places it within the EJB container.

## Appendix 2: The OMG Notification Service

The Notification Service is a standardized service of the CORBA platform defined by the OMG [8]. It provides applications the ability to use decoupled communication between their elements instead of traditional direct/synchronous communication. The Notification Service supercedes a similar but less-sophisticated service—the Event Service [9]—by offering more features and maintaining backward 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 8**.



**Figure 8:** *The Notification/Event Channel*

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. This and other features, such as Quality of Service (QoS) parameters, can be used to tailor the behavior of the Channel. However, these facilities are mentioned here only 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.

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

- Supplier elements can deliver events at different rates than that at which consumer elements process them. 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.

## The four kinds of Notification Channel

The Notification Service specifies four kinds of Notification Channel:

- Untyped Event Channel
- Structured Event Channel
- Sequenced Event Channel
- Typed Event Channel

All four types of channel propagate events from suppliers to consumers, but each represents the event using a different form. The choice of representation allows applications to choose a channel most suited to their needs.

The Untyped Event Notification Channel uses the CORBA: :Any data type to represent events. To deliver such data types, 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 define their own data structure for storing event information and then package that into the *Any* data type. This allows applications to continue using their own event data type 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 data types to represent events. For the Structured Channel, the data type 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 data type is found in the Notification Service specification (Section 2.2, [8]) and an abbreviated version (applicable to the discussions in this document) is provided below.

```
// Definition of a Structured Event from the CosNotification.idl file

struct EventType {
    string domain_name;
    string type_name;
};

struct FixedEventHeader {
    EventType event_type;
    string event_name;
};

struct Property {
    PropertyName name;
    PropertyValue value;
};
typedef sequence<Property> PropertySeq;
typedef PropertySeq OptionalHeaderFields;

struct EventHeader {
    FixedEventHeader fixed_header;
    OptionalHeaderFields variable_header;
}; // continued over page

typedef PropertySeq FilterableEventBody;

// Define the Structured Event structure

struct StructuredEvent {
    EventHeader header;
    FilterableEventBody filterable_data;
    any remainder_of_body;
};
```

The Sequenced Event Channel's event data type is a sequence of the Structured Event Channel's event data type. 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 unmarshaling 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 that are to be used to transfer events must specify a void return type and not use any out or in/out 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.

A 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 data type when publishing or consuming events. In contrast, when using a Structured, Sequenced, or Untyped Channel, an application must use a generic interface, which requires that events be formatted into an explicit event structure, delivered using an infrastructure-level operation, and extracted from the event data type. Nontyped channels therefore force clients to use a contract that is semantically weaker, less object-oriented, and less type-safe than that offered by the Typed Event Channel.

Although not covered here, 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.

## References

1. **Examples of Integrating EJBs with CORBA systems.** Java source code and documentation by Brenton Camac.  
<http://codecentral.borland.com/codecentral/ccWeb.exe/listing?id=20882>.
2. **Java Language Mapping to OMG IDL.** Version 1.3 (formal/03-09-04). The Object Management Group.
3. **A Simplified Java to IDL mapping for EJBs.** Java source code and documentation. Borland Software Corporation.  
<http://codecentral.borland.com/codecentral/ccWeb.exe/listing?id=20716>
4. **IDL-to-Java Language Mapping.** Version 1.2 (formal/2002-08-05). The Object Management Group.
5. **The Java Remote Method Invocation Specification.** Version 1.3. Sun Microsystems.  
<ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf>
6. **Notification / JMS Interworking Specification.** Version 1.0 (dtc/03-06-01). The Object Management Group.
7. **The JMS Specification.** Version 1.0.2. Sun Microsystems.  
<http://docs.sun.com/db/doc/816-5904-10>
8. **Notification Service Specification.** Version 1.0 (formal/2000-06-20). The Object Management Group.
9. **Event Service Specification.** Version 1.1 (formal/2001-03-01). The Object Management Group.

**Made in Borland®** Copyright © 2004 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. Microsoft, Windows, and other Microsoft product names are trademarks or registered trademarks of Microsoft Corporation 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](http://www.borland.com) • Offices in: Australia, Brazil, Canada, China, Czech Republic, Finland, France, Germany, Hong Kong, Hungary, India, Ireland, Italy, Japan, Korea, Mexico, the Netherlands, New Zealand, Russia, Singapore, Spain, Sweden, Taiwan, the United Kingdom, and the United States. • 21802