JMS - CORBA® Notification Service Interworking

A White Paper

March 2003

PrismTech Corporation

PRISMTECH

INTRODUCTION

Asynchronous messaging is a proven communication model for developing large-scale, distributed enterprise integration solutions. Compared with regular request/reply-based communications in traditional client-server systems, messaging provides more flexibility and scalability because senders and receivers of messages are decoupled and are no longer required to execute in lockstep.

In addition to the traditional requirements for a messaging system, most application integration solutions also have a basic requirement for interoperability of messaging systems, since components often execute in a heterogeneous environment. Although the Java® Message Service (JMS) is a powerful API for pure Java environments, many integration solutions being developed today require the JMS server to be capable of communicating with:

- Components written in languages other than Java.
- CORBA Components.
- Other proprietary MOM (Message Oriented Middleware) products for which a CORBA gateway exists.

A typical interworking use case occurs when Telecom Service Providers or Network Equipment Vendors built their Element Management Layer using CORBA and the Notification Service to implement their software and interfaces such as Integration Reference Points defined by the 3rd Generation Partnership Project (3GPP), and they require to achieve end-to-end business integration with EJB Network Management Layer based applications (e.g. Service Activation).

In response to this demonstrated need to integrate CORBA Notification Service and JMS, the OMG (Object Management Group) is currently working on standardizing interworking. Vendors such as PrismTech are also working ahead of the standards efforts to produce commercial solutions.

This white paper introduces JMS and CORBA Notification Service and describes the challenges being addressed by the OMG in providing an interworking solution.

The key design goal of PrismTech's interworking solution is to provide within the OpenFusion product suite a Notification Service-JMS Bridge that can link Enterprise Java Beans (EJBs) deployed into any application server, whether it is IIOP-compliant or uses other object transport protocols. This goal ensures that our solution is generic. The OpenFusion NS-JMS Bridge is compliant with the latest OMG Notification/Java Message Service Interworking Draft Revised Submission.

The approach that is taken by the OpenFusion NS-JMS Bridge contrasts with the some proprietary solutions that make EJBs CORBA Notification Service aware. These approaches require either the implementation of some Notification Service APIs at the EJB level or the implementation of the JMS APIs at the CORBA application level.

The interworking solution PrismTech provides with OpenFusion Notification-JMS Bridge is completely transparent to Application programs. The EJBs can still use their native Java Messaging Service APIs and the CORBA application still use the Notification Service APIs. This approach is particularly appropriate to integrating legacy applications, making it possible to protect and leverage existing investments.

CORBA NOTIFICATION SERVICE

The CORBA Notification Service is an OMG standard that allows multiple event suppliers to send events to multiple event consumers. It is a mature standard that has been one of the OMG's success stories. It has been widely used in a range of scenarios, such as the integrating mechanism for disparate telecom equipment, within large- scale routers on the internet backbone, part of e-commerce frameworks for major banks, and as the messaging infrastructure to link satellites and their ground stations.

The CORBA Notification Service differs from JMS in that its specification covers both the client interface and the messaging engine. Suppliers are de-coupled from consumers by means of an event channel, which takes care of client registration and de-registration and dissemination of events to multiple consumers. The channel also accommodates slow or unavailable consumers.

The CORBA Notification Service supports both push and pull models, and the architecture allows event channels to be federated without the use of intermediators. It extends the event service to include event filtering and a comprehensive "quality of service" (QoS) framework. Some of the more important QoS properties include:

- Persistent events and connections to support guaranteed delivery.
- Queue management to support ordering and discard policies.
- Event start and stop times to delay and time out events.

The CORBA Notification Service also supports the concept of a structured event, which qualifies as a "proper" message in terms of message middleware systems. The main interfaces in the CORBA Notification Service are depicted in Figure 1, which also shows that the CORBA Notification Service supports un-typed, structured, and sequence client types for sending and receiving events.

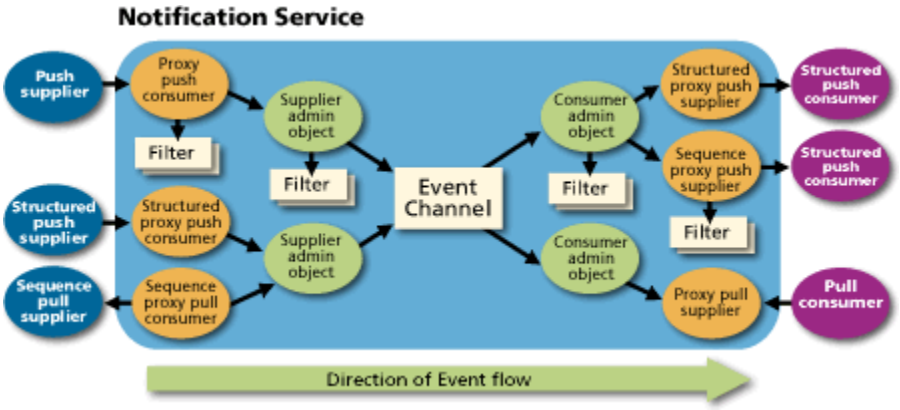

Figure 1: Main Interfaces of the CORBA Notification Service

A developer will typically perform the following steps in CORBA Notification Service client applications:

1. Obtain a reference to an event channel, e.g., create one using the event channel factory.
2. Get or create an administration object; a supplier must use a "supplier administration object," while a consumer must use a "consumer administration object."

3. Create a proxy object, which is a connection end point. Suppliers use proxies to send events, while consumers use proxies to receive events.
4. Attach filters to administration or proxy objects if necessary.

One of the more important architectural features of the CORBA Notification Service is that it supports channel federation without the use of intermediators that forward events from one channel to another; a proxy supplier can be connected directly to a proxy consumer. This feature implies that routing with the CORBA Notification Service can be achieved very easily for improved reliability, scalability, and performance. Since the CORBA Notification Service is based on the CORBA architecture, it also supports integration with applications not written in Java.

JMS

The Java Message Service (JMS) defines a standard API that allows Java developers to easily build enterprise integration solutions. JMS is important in its own right because it provides a simplified and common way for Java clients to access message-oriented middleware (MOM). For example, JMS interfaces have been produced to both IBM and Tibco's messaging products. More importantly, with the introduction of message-driven beans (MDB), JMS has become even more tightly integrated into J2EE. This provides an asynchronous manner for Enterprise Java Beans (EJBs) to communicate with other elements in a distributed architecture. In a relatively short space of time, JMS has become an enormously popular messaging paradigm and enjoys support from all of the major messaging vendors.

It should also be recognized that JMS was designed as an abstraction over existing (and new) messaging products. This has a number of benefits, including the ability to replace message systems with no or few client modifications. This abstraction has also resulted in the following characteristics:

- JMS is purely an interface. In order to transport and route messages, some form of messaging engine is required. The JMS specification has nothing to say about the engine, its architecture, or the transport. This has led to a wide range of different solutions from different vendors.
- The JMS specification does not facilitate interoperability between different implementations. This is a natural consequence of the previous point. If the specification does not mandate a transport protocol and wire format, then there will never be interoperability.
- Compared to the proprietary MOM products, JMS has a relatively simple set of five message formats.

JMS supports a point-to-point (or queue) model and a publish/subscribe model, and it defines a number of message types that publishers and subscribers can exchange. Messages support properties that define how they should be treated by the message system. Clients can be transient or durable, and messages can be sent reliably.

Although the publish/subscribe and point-to-point communication models are different from a conceptual point of view, the authors of JMS realized that the models have a lot in common. JMS is therefore centered on a generic messaging model, and publish/subscribe and point-to-point are derived (in the sense of interface inheritance) from the generic model.

JMS supports five different kinds of messages, which are used to carry different types of payload. The header of a message is the same regardless of the payload, which means that filtering is the same for all five message types. A message supports a number of properties to set priority, reliability, and other QoS properties, which will be interpreted and handled by the JMS server.

INTEGRATION POINTS

Although there is a comfortable overlap between the JMS and CORBA Notification Service communication models and capabilities, there are three areas where the integration must necessarily be defined:

- Automatic federation between the Notification Service channel concept and the JMS topic/queue concept.
- QoS mapping is important to guarantee end-to-end QoS for mission critical applications. Both JMS and the Notification Service define a rich set of message delivery and queuing related QoSs. Fortunately, the Notification Service and JMS QoS management frameworks are flexible enough to support all QoSs defined by the two messaging systems.
- The CORBA Notification Service supports different type of messages, called generic, structured and typed events. JMS supports five different message formats with different message bodies. The JMS message payload type can be a text, stream, map, byte or object. It is therefore necessary to translate a CORBA message into a JMS message.

We will describe how these issues are being addressed by the OMG standards work and by commercial implementations such as PrismTech's ***OpenFusion NS–JMS Bridge***.

BRIDGE ARCHITECTURE OVERVIEW

The bridge automates the federation of JMS destinations and Notification Service Event Channels at configuration and deployment time. The Bridge also creates and manages bridge instances that perform automatic mapping and forwarding of messages and events.

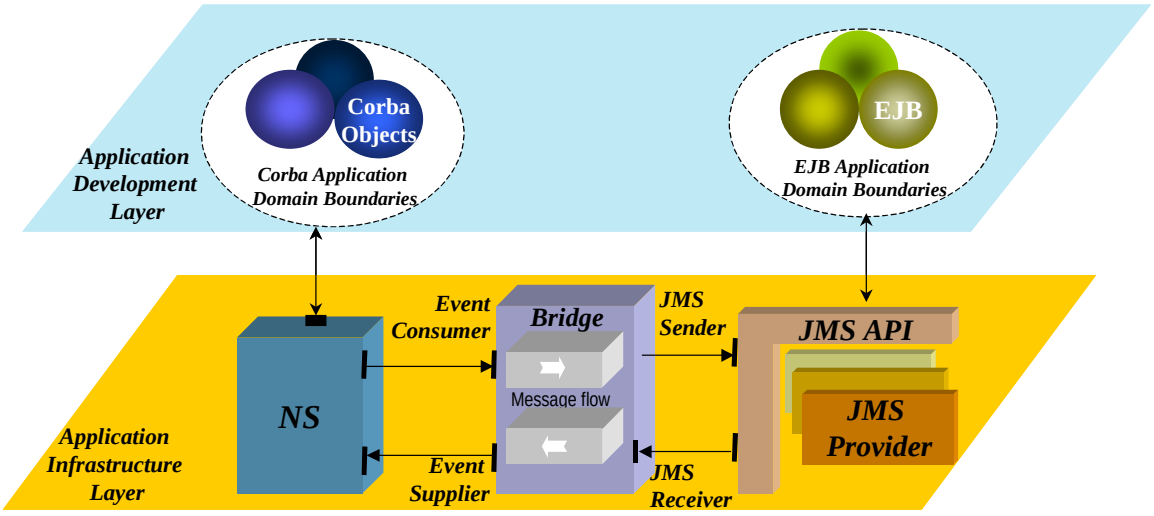

Figure 2: Architecture Overview

Figure 2 shows the different relationships between the bridge, the Notification Service and the JMS. The figure depicts the transparency of the bridge for the application developers. At the development level the end user applications still use their native underlying messaging APIs.

The bridge can connect OpenFusion Notification Service with any JMS system independently from the JMS engine and the object transport protocol used by the Application Server.

To improve performance, the bridge can be configured to group and forward events using a minimal number of remote procedure calls. This is achieved through the use of the event batching mechanism.

Bi-directional QoS mapping

Today it is widely accepted that maintaining QoSs is fundamental for mission critical applications. This, for example, is the case when sending a taxation event conveying network resource time consumption from a CORBA -based network management system to an EJB based billing system. In this case, the lost data is prejudicial for the telecom operator and therefore it is essential to maintain the Reliability QoS for both the CORBA and J22E domains.

Event Reliability

The Notification Service QoS property `EventReliability` is used to define the level of reliability at the event level. The usage of this QoS avoids the loss of events. It is mapped to JMS QoS property `JMSDeliveryMode`. Each value of the properties is mapped as follows:

EventReliability	**JMSDeliveryMode**
BestEffort	NON_PERSISTENT
Persistent	PERSISTENT

Table 1: Event Reliability Mapping

Connection Reliability

The Notification Service's QoS property `ConnectionReliability` is used to define the level of reliability at the connection level. This requires that the entry point objects at the Notification Service (e.g Proxy) and JMS (e.g TopicSubscriber) are Persistent. This QoS maps to `TopicSubscriber` object in the JMS Publish/Subscribe model. Each value of the property is mapped as follows:

ConnectionReliability at object level	**JMS Publish/Subscribe model at the object level**
BestEffort Proxy	TopicSubscriber
Persistent Proxy	Durable TopicSubscriber

Table 2: Connection Reliability Mapping

Order Policy

The Order policies manage the queuing order policies. The Notification Service's QoS property `OrderPolicy` are mapped to JMS's order policy as follows:

OrderPolicy	**JMS's order policy**
PriorityOrder	JMSPriority
DeadlineOrder	JMSExpiration
FifoOrder	Supported

Table 3: Priority Mapping

Expiry times

The Notification Service `StopTime` QoS defines the event time-to-live. When the date conveyed by this QoS expire the event is deleted from the event channel. This QoS maps to the JMS QoS property `Timeout`.

Message mapping

The JMS messages derive common functionality from the base message interface. The Message interface is a base interface for all JMS message. A JMS message consists of a header, a set of properties and a body. The body part is different for each of the five different JMS message types. The header and properties are the same for all message types. Some properties a pre-defined, others can be extended by the application users.

The Notification Service specification made event grouping possible through structured event sequences only. Event-grouping is crucial to improve interworking performance. This makes structured events centric in the JMS –NS message mapping.

Structured Events provide a well-defined data structure which is comprised of two main components: a header and a body. The header can be further decomposed into a fixed portion and a variable portion. The message to event mapping is bi-directional and it is performed without information loss.

JMS Message to Event

The mapping of the JMS Message header and properties part is independent from the message type. The body mapping depends on the message types enumerated above.

The JMS Message header is made up by several fields for setting various Quality of Service (QoS) such as `JMSDeliveryMode`, `JMSExpiration` and `JMSPriority`. Those fields have well-defined meanings in the structured event. They are mapped as follows:

- `JMSDeliveryMode` maps to The `EventReliability` QoS in the variable header of a structured event.
- `JMSExpiration` maps to the `Timeout QoS` field in the variable header of a structured event.
- `JMSPriority` maps to the `Priority QoS` in the variable header of structured events.

The rest of the JMS header fields are mapped to the structured event optional header fields.

Metadata such as the JMS Topic name or the JMS message type map to the structured event fixed header fields. The message body mapping depends on the message types, it is performed as follow:

Text Message

A `TextMessage` provides a body, which is a Java String. The body is inserted into a `remainder_of_body` of the structured event by simply inserting the string into the `Any`.

Stream Message

A `StreamMessage` provides a body which contains a stream of Java primitive values. The values on the stream stack are written onto the `remainder_of_body` of the structured event using the `AnySeq` data type. The elements in this sequence are mapped using the standard Java to IDL mapping.

Map Message

A `MapMessage` provides a body of name-value pairs where names are `Strings`, and values are Java primitives. The body can be inserted in the `remainder_of_body` field, of a structured event using the `PropertySeq` data type.

Byte Message

A bytes message supports a body with un-interpreted data. The message supports the methods of the `DataInputStream` and `DataOutputStream` interfaces from the Java I/O package. As the body is an array of bytes it is written to the remainder of the body field of a structured event using an IDL octet sequence. The `OctetSeq` data type is defined in the notification service IDL extension module.

Object Message

An object message provides a body that can contain any Java object that supports the Serializable interface. This type of message is serialized onto a byte sequence and written onto the any in the remainder of the body using the same `OctetSeq` data type described above. On the receiver side the byte sequence is converted to an object input stream where the object is read from.

Event to JMS Message

Since the Event fixed header fields don't have direct equivalent fields in the JMS message structure, those fields are mapped to the JMS user defined header fields. This guarantees that the information conveyed by the event is not lost during the mapping.

Those fields with a well-defined structure such as `EventReliability`, `Timeout` or `Priority` fields map directly and respectively to `JMSDelivery`, `JMSpriority` and `JMSTimetolive`.

JMS fields with no equivalence in structured event are filled in automatically by the JMS Bridge This, for example, is the case for the `JMSMessageID` or the `JMSDestination` fields.

The mapping of structured event body to given JMS message type body depends on the complexity of the data wrapped into `remainder_of_body` field.

When remainder_of_body typed CORBA::Any involves:

- IDL basic type elements, each element maps to a Java primitive type using standard IDL to Java mapping. The set of elements obtained are enterer in JMS StreamMessage body.
- `String` type element only, it maps to a Java string type and is placed in JMS message body.
- Sequence of Properties (`PropertySeq`), it maps to a body of name-value where names are strings and values are java primitives.
- Octet Sequence or other type such as user constructed types, it maps to a body of `BytesMessage`.

The mapping of complex data structures is supported. For example, if the structured event contains the pair `<name, value>= <Fd_name1, CORBA::Any A>`, and the A value wraps the structure named `Alarm { string Al_name; int Severity }` then this field will be transformed in to two JMS user defined properties (fields) : `<$Fd_name1.Al_name, string>` and `<$Fd-name1.Severity, integer>`.

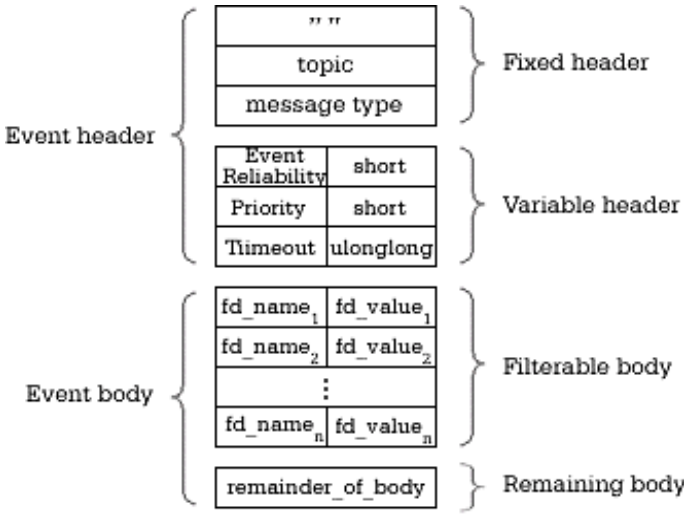

Figure 3: Mapping between JMS message and structured event

Figure 3 summaries the mapping between JMS messages and structured events.

CONCLUSION

The asynchronous communication paradigm is recognized as the most flexible and reliable way to communicate between distributed applications. JMS is the native API for Message Driven EJBs. Notification Serve is the native API for event Driven CORBA applications. The OpenFusion NS-JMS bridge is a simple and seamless way to integrate JMS and Notification Service. Since no changes are required for either the EJB side or for the CORBA application side, the OpenFusion NS-JMS Bridge enables users to protect and leverage their existing software investments.

CONTACTS

PrismTech can be contacted at the following address, phone number, fax and e-mail contact points for information and technical support.

Corporate Headquarters

PrismTech Corporation
6 Lincoln Knoll Lane
Suite 100
Burlington, MA
01803
USA

Tel: +1 781 270 1177
Fax: +1 781 238 1700

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Team Valley
Gateshead, NE11 0NG
UK

Tel: +44 (0)191 497 9900
Fax: +44 (0)191 497 9901

Web: *http://www.prismtechnologies.com*
General Enquiries: *info@prismtechnologies.com*

NOTICES