

*Convenience Document with Proposed Changes for
Notification-JMS Interworking FTF*

DTC 04-xx-xx

January 09, 2003

OMG TC Document telecom/2002-01-06

Table of Contents

1	OVERVIEW	5
2	ARCHITECTURAL FEATURES	7
2.1	BRIDGE ARCHITECTURE OVERVIEW.....	7
2.2	BRIDGE FACTORY.....	8
2.3	BRIDGE INSTANCE.....	9
2.4	MESSAGE MAPPING.....	12
2.4.1	<i>JMS Message to Event</i>	13
2.4.1.1	JMS message body mapping	16
2.4.1.1.1	Text Message	16
2.4.1.1.2	Stream Message	16
2.4.1.1.3	Map Message	16
2.4.1.1.4	Bytes Message.....	16
2.4.1.1.5	Object Message.....	17
2.4.2	<i>Event to JMS Message</i>	17
2.5	QOS PROPERTIES MAPPING.....	21
2.5.1	<i>Event Reliability</i>	21
2.5.2	<i>Connection Reliability</i>	21
2.5.3	<i>Delivery Reliability</i>	22
2.5.4	<i>Priority</i>	22
2.5.5	<i>Expiry times</i>	23
2.5.6	<i>Order Policy</i>	23
2.6	ACKNOWLEDGMENT MAPPING.....	25
2.6.1	<i>Overview of Reliable Event Delivery with Event Acknowledgment</i>	25
2.6.1.1	Event Acknowledgment on Push Model	26
2.6.1.2	Event Acknowledgment on Pull Model	27
2.6.2	<i>Scope of Event Acknowledgment</i>	28
2.6.3	<i>Sequence Number Header Field</i>	29
2.6.4	<i>Notify Ack Interface</i>	29
2.6.4.1	Acknowledge	30
2.6.5	<i>Reliable Delivery Sequence</i>	31
2.6.5.1	Reliable Delivery Sequence for Push Model.....	31
2.6.5.2	Reliable Delivery Sequence for Pull Model	32
2.6.5.3	Recovery in Failure of Retries	33
2.6.6	<i>QoS Properties for Reliable Event Delivery</i>	34
2.6.6.1	DeliveryReliability.....	34

2.6.6.2	Retries	35
2.6.6.3	RetryInterval.....	35
2.6.6.4	Supported level of the QoS Properties.....	35
2.6.7	<i>Mapping between Event Acknowledgment and JMS Message Acknowledgment</i>	35
2.7	TRANSACTIONAL SUPPORT.....	38
2.7.1	<i>Asynchronous Transactional Model</i>	38
2.7.2	<i>Supported configurations</i>	40
2.7.3	<i>Notification Service Transactional Role</i>	40
2.7.3.1	Data Flowing from Notification Service to JMS	41
2.7.3.2	Data flowing from JMS to Notification Service.....	41
2.7.4	<i>Bridge Transactional Roles</i>	42
2.7.5	<i>JMS service provider Transactional Role</i>	42
2.7.5.1	Data Flowing from Notification Service to JMS	42
2.7.5.2	Data Flowing from JMS to Notification Service	43
2.7.6	<i>Bridge Transactional Monitoring</i>	43
2.7.6.1	Notification Service QoS and Admin Property Extensions.....	43
2.7.6.2	Bridge Transaction Management Interface	46
2.8	CONFORMANCE.....	47
3	BRIDGE INTERFACES	48
3.1	CO_SBRIDGEADMIN MODULE	48
3.1.1	<i>ExternalEndPoint</i>	51
3.1.1.1	ExternalEndpointRole.....	51
3.1.1.2	ExternalEndpointConnector.....	51
3.1.1.3	MessageType.....	52
3.1.2	<i>EndPointSender interface</i>	52
3.1.3	<i>EndpointReceiver interface</i>	52
3.1.4	<i>Bridge Interface</i>	52
3.1.4.1	start_bridge	52
3.1.4.2	stop_bridge.....	52
3.1.4.3	get_status.....	53
3.1.4.4	destroy	53
3.1.5	<i>Bridge Factory Interface</i>	53
3.1.5.1	create_bridge.....	53
3.1.5.2	get_all_bridges	54
3.1.5.3	get_bridge_with_id.....	54
3.2	BRIDGETRANSACTIONMGMTMODULE	54
3.2.1	<i>TransactionManagement interface</i>	54
3.2.1.1	enable_transaction	55
3.2.1.2	disable_transaction	55

1 Overview

Asynchronous messaging is a proven communication model for developing large-scale, distributed enterprise applications. In order to support flexible and end-to-end business integration, it is becoming necessary to provide messaging interworking between CORBA applications and Java / EJB.

The CORBA Notification Service is the OMG mature standard that allows Corba objects, named suppliers, to send event asynchronously to other Corba object named consumers. Suppliers are de-coupled from consumers by means of event channel concept, which takes care of dissemination of events to them.

The Java Message Service (JMS) defines a standard API that provides a simplified and common way for Java clients to access message oriented middleware. More importantly, JMS is tightly integrated into J2EE and is the messaging standard for Enterprise Java Beans (EJB). Application publishing a message are de-coupled from application receiving them by the mean of Queue or topic concept.

The Notification Service differs from JMS in that its specification covers both the client interfaces and the messaging engine whereas JMS was designed as an abstraction over existing and new messaging products. The JMS messaging engine implementation may differ from one vender to another.

This document specifies architecture and interfaces for managing Notification Service interworking with Java Message Service. The interworking involves several aspects such as:

- Event -Message mapping,
- QoS mapping,
- Automatic federation between Notification Service channel concept and topic/queue concepts

- Transaction support.

2 Architectural Features

2.1 Bridge Architecture Overview

The bridge defined by this specification is designed to manage and interconnect event channel with JMS destination. A clear goal of this specification is to define the capability to manage an inter-related channel and JMS destination that can be created via existing implementations of those services. The guiding principles which drove the definition of the Bridge IDL interfaces were to preserve backward compatibility with both JMS and Notification Service programming models. Extensions to current Notification Service can be required only in advanced use cases where acknowledgment data delivery QoS is required.

The Figure 2-1 depicts bridge general architecture. The IDL module names of the interfaces defined by the Notification Service and the Bridge are abbreviated in the above diagram. NCA stands for CosNotifyChannelAdmin, while NC stands for CosNotifyComm, finally BA stands for CosBridgeAdmin.

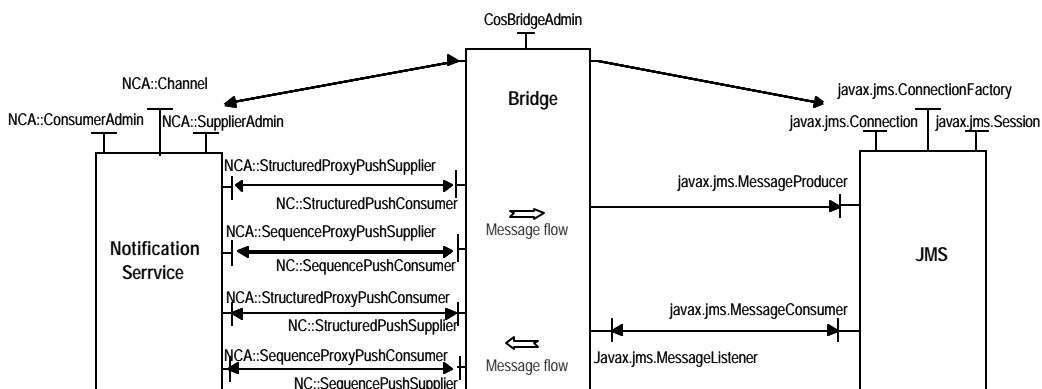


Figure 2-1 General Architecture of the Bridge

The Bridge is used to create and manage bridges instances that perform automatic mapping and forwarding of messages and events. The Figure 2-1 shows the different relationships between the bridge, the notification service and the JMS. For the sake of clarity only the push communication style is considered in the Figure 2-1.

To preserve the Notification service and JMS interfaces, the Bridge behaves as event Consumer and as JMS sender when forwarding an event from event channel to JMS destination. In addition, it behaves as JMS receiver and event Producer when forwarding a message in the other way.

Event-grouping is crucial to improve interworking performance. This makes structured events centric in the JMS-NS message mapping, consequently, when the push communication style is selected the Bridge supports the standard `StructuredPushConsumer`, `SequencePushConsumer` and `StructuredPushSupplier` and `SequencePushSupplier` interfaces. It supports also the `StructuredPullConsumer`, `SequencePullConsumer` and `StructuredPullSupplier` and `SequencePullSupplier` interfaces.

To receive messages from JMS, the bridge offers the standard JMS `javax.jms.MessageListener` interface and makes use of JMS `javax.jms.MessageProducer` and `javax.jms.MessageConsumer` interfaces.

The bridge is also used to automate the connection setups between channel and JMS destination. It performs on behalf of JMS the necessary steps to create and configure entry points in the event channel. These steps involve `StructuredProxyPush` creation and default QoS setting. Similarly, it performs on behalf of the notification service steps needed to create and configure connection and session with JMS provider.

Note-. Issues related to detection of event or message duplication in complex topologies is out of the scope of the RFP and are not taken into account. Alternatively, techniques such as those used in Event Domain Management can be used.

2.2 Bridge Factory

A `BridgeFactory` is responsible for the creation of Bridge objects based on initial parameters. In order to create a bridge, it is necessary to have provide information on an existing Notification Service event channel and JMS destination. Channel and JMS destination information are abstracted using external end point connector concept.

External end point connector may provide or suck up data flow. It can be either a source or sink of data. When External end point connector describes JMS destination

it should indicate the type and the name of the destination should it be a topic or queue. When it describes channel information it should indicate whether data will be sent using sequence or single structured events. Finally, the BridgeFactory user should indicate the communication style it wishes to use. The communication style can be either Push or Pull.

Since the JMS specification supports Pull communication style on the application receiver side only, the BridgeFactory should check the consistency of the end-to-end¹ communication model in use before creating a bridge. For example, when forwarding data from event channel to JMS destination, the Pull communication style can't be set at the sink external endpoint. This scenario is summarized in Table 2-1

Some other consistency checking of the end-to-end communication model is implementation dependent. For example, bridge implementation with storage capabilities may support PUSH communication style at JMS side acting as a source and PULL communication style at event channel side acting as a sink only and only if the bridge uses buffer that desynchronize data transmission from its reception by event channel.

Controls performed by Bridgefactory are summarized in Table 2-1.

		Sink : JMS					Sink : Channel	
		PUSH	PULL				PUSH	PULL
Source: Channel	PUSH	yes	Not Allowed by JMS specification	Source: JMS	PUSH	Yes	Implementation dependent	
	PULL	yes	Not Allowed by JMS specification		PULL	Yes ²	yes	

Table 2-1: Communication consistency checking

A compliant bridge implementation is not required to support all communication models described in Table 2-1. However, vendors are encouraged to provide several communication styles to increase bridge flexibility.

2.3 Bridge Instance

To propagate unidirectional data flow a bridge instance connects a single source endpoint to a single sink endpoint. This object offers two interfaces that fit the source

¹ End-to-end portion concerns the source to sink endpoints.

² To be able to pull the JMS and to push toward event channel, a kind of scheduling can be passed to the bridge using the CoSNotification::PacingInterval QoS defined at each external endpoint side.

and sink natures and requirements. A source is connected to the bridge instance through the Endpoint receiver and the sink is connected through the Endpoint sender. When bi-directional interworking between the Notification Service and JMS is required two bridge instances can be created separately. The Figure 2-2 summaries the bridge architecture abstract view.

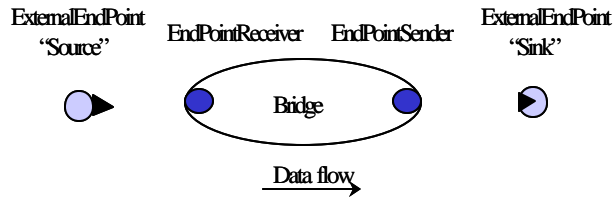


Figure 2-2: Bridge abstract Architecture

Depending on the source endpoint nature it is connected to, the endpoint receiver can be a:

- JMS Message Listener, if it receives messages from JMS destination. At abstract level, the JMS Message Listener behaves as an event Push Consumer, in that it offers *onMessage* operation which, from functional viewpoint, can be compared to push operation defined in Notification Service.
- JMS Message Consumer, when it retrieves messages from JMS destination. It can be compared to event pull consumer.
- structured push consumer, if it connects an event channel pushing single structured event,
- sequence push consumer, if it connects an event channel pushing sequence of structured events,
- structured pull consumer, if it connects an event channel offering pull structured event operations,
- structured pull consumer, if it connects an event channel offering pull structured event operations.

The endpoint sender can behave as a:

- JMS Message producer, when sending events to JMS destination,
- Structured push supplier, when sending event to event channel,
- Sequence push supplier, when sending events to event channel,

- Structured pull supplier, when pulling event from event channel,
- Sequence pull supplier, when pulling event from events channel.

When creating a bridge, regular steps to connect event channel and JMS destination are performed. They consist of:

1. Obtaining administration object references from event channel and JMS. On event channel side, those object references are `SupplierAdmin` or `ConsumerAdmin` and on the JMS side those objects references correspond to `Connection` and `Session`.
2. Creating on event channel side `proxySupplier` or `proxyConsumer` objects and creating on JMS side `MessageProducer` or `MessageConsumer` objects that fit communication style and data grouping policy selected by user.

A bridge is a stateful object that reflects the connection states with both proxy and JMS destination entry³. Consequently, the bridge state is an aggregation of Proxy states and the JMS destination entry state it is connected to. The AND logical operator semantic should be applied to obtain the bridge state.

When it is created the bridge status should be set to Inactive. When starting the bridge, the endpoint receiver and sender should activate simultaneously the connections with the proxy and JMS destination entry. If these two steps are successfully achieved the bridge state become Active.

The bridge evolve to Inactive state when at least one connexion with the proxy or JMS destination is lost or suspended.

The bridge interface offers the following operations: `start_bridge`, `stop_bridge`, `get_status` and `destroy`.

The `connect_xxx` operations class provided in `CosNotifyChannelAdmin` module are used to establish connection with event channel proxy object. The JMS

`javax.jms.Connection.start` operation can be used to activate connexion with JMS provider. The invocation of the `start_bridge` operation on bridge inactivated by the use of `stop_bridge` operation resumes the `connections` with the proxy and JMS destination entry. The `resume_connection` operations class provided in `CosNotifyChannelAdmin` module and the `javax.jms.Connection.start` operation can be respectively used. When

Deleted: start

Deleted: stop

Deleted: start

Deleted: start_bridge

Inserted: start_bridge

Deleted: start

Deleted: stop

Deleted: connexions

³ A JMS destination entry object can be either a Message Producer or MessageConsumer.

the bridge invokes successfully *connect_xxx* or *resume_connection* and *javax.jms.Connection.start* its state become Active.

Note – exception behavior of the Bridge Interface operations will be described in section 3

Deleted: *stop*

The *stop_bridge* operation stops the connections with the proxy and JMS destination entry. The *suspend_connection* operations class provided in *CosNotifyChannelAdmin* module and the JMS *javax.jms.Connection.stop* operation can be respectively used to suspend the connection with event channel proxy object and the connexion with JMS provider. When the connections with the proxy and JMS destination entry are successfully suspended the bridge state becomes Inactive.

The *get_status* operation is intended to describe the status of end-to-end connection stating from source to sink endpoints. To return up-to-date status, the bridge can use the JMS *exceptionListener* interface to detect Connexion problems with JMS provider. Since Notification Service does not provide tools to get the connexion status of proxy, the bridge implementation may use *connect_structured_xxx* operations to deduce the connection status. The use of this operations class is idempotent. When receiving *AlreadyConnected* exception, the connection status is active. Otherwise, the status is considered inactive.

The *destroy* operation destroys the bridge object, invalidating its object reference. To liberate resources on the event channel and JMS sides the *disconnect_xxx* class provided in *CosNotifyChannelAdmin* module and the *javax.jms.Connection.close*⁴ operations should be respectively invoked before destroying the bridge object.

Beside the configuration steps described above, the bridge perform data mapping from structured event format to JMS message formats and vice versa. The bi-directional mapping is described in section 2.4.

2.4 Message Mapping

The JMS specification defines five different messages that all derive common functionality from the base *Message* interface. The Notification Service specification defines three event types and associates them well defined translation rules making the consumption of events produced in different formats possible.

The Notification Service specification made event grouping possible through structured event sequences only. Event-grouping is crucial to improve interworking performance.

⁴ Note that in JMS there is no need to close the sessions, producers, and consumers of a closed connection.

This makes structured events centric in the JMS–NS message mapping. This section describes the mapping between structured events and the different JMS message types, namely:

- TextMessage,
- StreamMessage,
- BytesMessage,
- MapMessage*,
- ObjectMessage*.

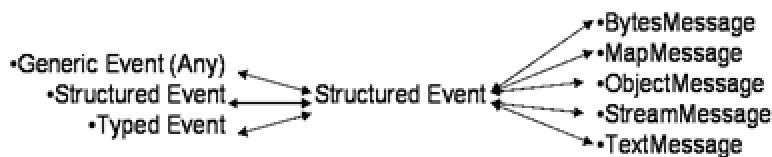


Figure 2-3 Structured event role in data mapping

A JMS message consists of a header, a set of properties and a body. The header and properties are the same for all message types. The body part is different for each of the five different JMS message types.

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 current version of the specification addresses bidirectional mapping without information loss. Future version may be enhanced by customizable mapping interfaces that discard irrelevant data for the application receiver and event consumer.

2.4.1 JMS Message to Event

This section describes the mapping of message sent by JMS client toward Notification Service Consumer. 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.

JMS Header and properties mapping

When possible, the mapping from JMS to structured event should follow the general naming conventions adopted by the Notification Service specification when translating generic event (Any) or typed event types to structured event. Structured event fields

such as `domain_name`, `type_name` and `event_name` should be compliant with the notation adopted in section 2.1.4 of the Notification Service specification.

- The `domain_name` data member should be set to empty string.
- The `type_name` data member should start with the “%” character and indicate the JMS message source type, namely, the `TextMessage`, `MapMessage`, `StreamMessage`, `BytesMessage` or `ObjectMessage`. For example, the `type_name` data member would be set to the value “%TextMessage” if the JMS source message type is `textMessage`.
- The semantics associated with `event_name` data member is used by end-users only, it is not interpreted by any component Notification Service. This field can be optionally set to the `Topic` or the queue name through which JMS message was published or sent. The extra-information delivered within `event_name` field may be used by a JMS-aware event consumer.

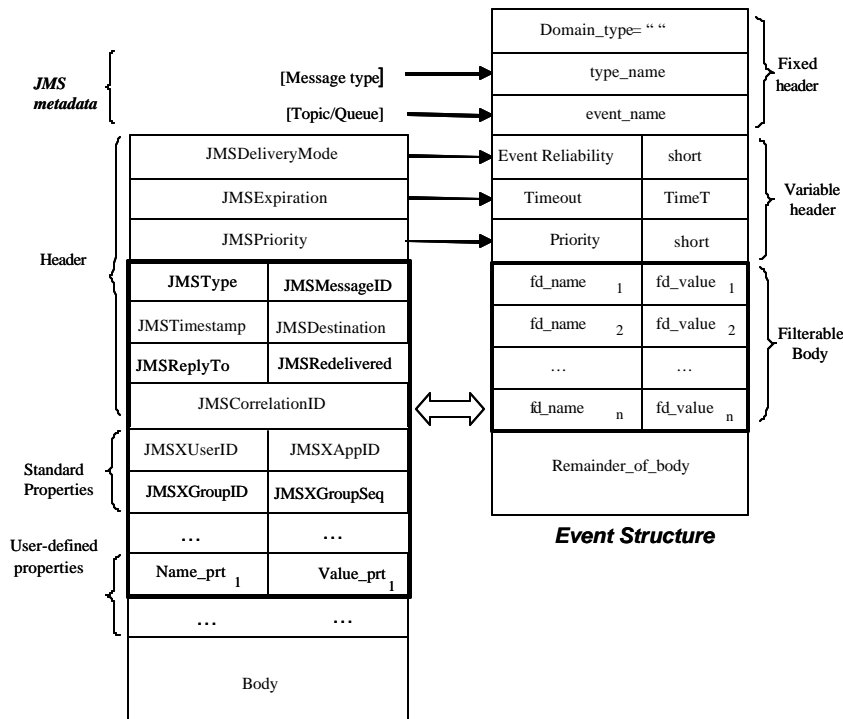


Figure 2-4 JMS message to Structured Event mapping

The JMS Message header is made up by several fields for setting various Quality of Service (QoS) s such as `JMSDeliveryMode`, `JMSExpiration` and `JMSPriority`.

Those fields have well-defined meanings in the structured event and they must be mapped as follow:

- `JMSDeliveryMode` maps to The `EventReliability QoS` in the variable header of a structured event. It is set to `Persistent` when the delivery mode is `PERSISTENT`. Otherwise, the `EventReliability QoS` is set to `BestEffort`.
- `JMSExpiration` maps to the `Timeout` in the variable header of a structured event. JMS expiration time value is expressed in milliseconds. This value is converted to units of 100 nanoseconds as this is the base unit of time in CORBA. Expired messages are not visible to clients.
- `JMSPriority` maps to the JMS message priority. It is mapped to the `Priority QoS` in the variable header of structured events. Priority delivery mode is used to ensure that messages with higher priority are delivered before messages with lower priority values.

The rest of the JMS header fields can't be mapped directly to standard structured event fields of the variable header portion; however those fields can be viewed as optional information that may be useful for JMS-aware event consumer. For example, a `JMSReplyTo` field with a valid value can be used by event consumer that would like to react, after receiving the JMS message converted into a structured event, by producing a new event which can be seen as reply message. The reply message will be sent through channel linked up with the destination specified into the `JMSReplyTo` field. In this case the `JMSCorrelationID` may also be reused by event consumer in the reply message to allow the JMS sender to tie up with the initial message it sent.

To increase filtering capability on the event channel side, the `JMSType`, `JMSMessageID`, `JMSTimestamp`, `JMSReplyTo`, `JMSCorrelationID`, `JMSDestination`, `JMSRedilivred` fields are mapped, by default, to the filterable date member of the structured event. Each of them is inserted using the name-value pair, i.e using the `PropertySeq` data type defined in the `CosNotificfation` module.

To decrease the structured event length and increase performance, JMS header fields with nil values can be omitted during the mapping process.

The JMS property fields, prefixed by `JMSX`, are optional part of the JMS message structure. Some of them are standard and well defined by JMS specification, as those enumerated in the Figure 2-4, others are defined by JMS end-user. In the current version of the specification, the default behavior is to map all `JMSX` fields. Future version may restrict the mapping to `JMSX` fields that are relevant to the event consumer. This will improve performance and mapping pertinence.

All JMSX Property values are java primitives data type, they can be `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and `String`. They are all entered into the filterable body of the structured event using the name-value pair, i.e using the `PropertySeq` data type defined in the `CosNotification` module and they are mapped using the standard Java to IDL mapping.

The order of JMS properties is not defined in JMS. The Notification-JMS specification doesn't mandate any particular order when implementing the mapping.

2.4.1.1 JMS message body mapping

It is expected that the text, map, and stream messages will be the three specific message types intensively used in an environment that consists of both JMS and non-JMS clients.

2.4.1.1.1 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`.

2.4.1.1.2 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.

2.4.1.1.3 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.

2.4.1.1.4 Bytes 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.

2.4.1.1.5 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.

2.4.2 Event to JMS Message

When a message is sent from a Notification Service event channel to JMS destination, the construction of the JMS message is performed as follows:

1. When defined, the standard optional part of the event variable header is mapped to corresponding fields in JMS header. If the event supplier does not define those fields, then JMS header fields are populated using the default values specified in the JMS specification.
2. The data in the event fixed header, the rest of the optional header fields as well as the event filterable body are placed in the JMS properties fields. They will be seen by the JMS receiver application as user-defined fields.
3. Meta data created by JMS-NS bridge is used to define complete JMS missing fields such as `JMSDestination` or `JMSMessageID`.
4. The `remainder_of_body` section is mapped to the JMS message body depending on the complexity of the data format wrapped in the Any.

Structured Event Header and filterable body mapping

When defined in structured event, the `EventReliability`, `Timeout` or `Priority` fields are respectively mapped to `JMSDelivery`, `JMSPriority` and `JMSTimetolive`. If `EventReliabilty` is not defined `JMSDelivery` is set to `PERSISTENT`. If the `Timeout` is not defined `JMSTimetolive` to `Unlimeted`. If the `Priority` is not defined the `JMSPriority` is set to 4.

The User-defined property fields are pair of name-value. The structured event fixed header fields are mapped to the JMS message User-defined property fields as follow:

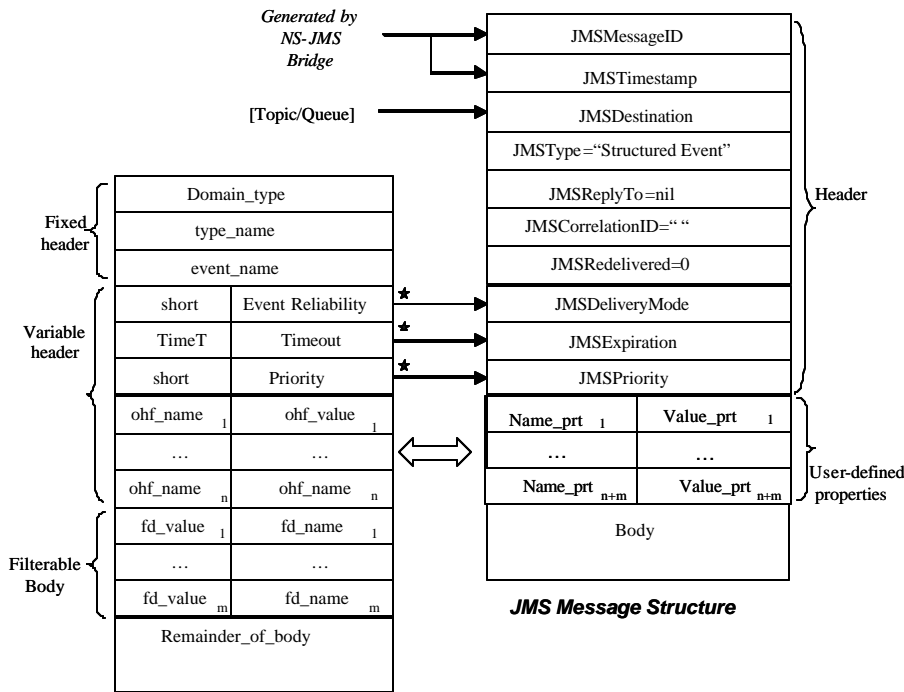
- The `domain_type`: A new property name labeled `$domain_type` is created. It must obey the rules for a message selector identifier⁵ specified in Section 3.8.1.1 of JMS specification. The content of the `domain_type` field in the event is converted to java String.
- The `type_name`: A new property name labeled `$type_name` is created. It must obey the rules for a message selector identifier specified in Section 3.8.1.1 of JMS specification. The content of the `type_name` field in the event is converted to java String.
- The `event_name`: A new property name labeled `$event_name` is created. It must obey the rules for a message selector identifier specified in Section 3.8.1.1 of JMS specification. The content of the `event_name` field in the event is converted to java String.

Inputs of `JMSMessageID`, `JMSTimestamp`, `JMSDestination` and `JMSType` fields are fulfilled by JMS-NS bridge.

- `JMSMessageID` is a String value which should be a unique key, prefixed by 'ID'. The exact scope of uniqueness is provider defined.
- `JMSTimestamp` field contains the time a message was handed off to JMS to be sent. It is in the format of a normal Java millis time value.
- `JMSDestination` contains the topic or the Queue name to which the message is being sent.
- `JMSType` is a String value that should be set to 'Structured Event'.

Complete specification those header fields are defined in the JMS specification.

⁵ 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 '\$'.



Event Structure

Figure 2-5 Structured Event to JMS message mapping

For each optional header (ohf_*) or filterable data (fd_*) field a new property name labeled \$ohf_* or \$fd_* is created. It must obey the rules for a message selector identifier specified in Section 3.8.1.1 of JMS specification. The content of the ohf_* or fd_* field is converted to java data type primitives.

If the optional header or filterable data field has the struct IDL type then multiple JMS properties are created, one for each primitive element of the complex type. The structure is linearized as follow:

- The new \$ohf_* or \$fd_* field name is concatenated with the structure and the member of the structure names. The structure member operator '.' is used to delimitate each name. This process is repeated if the structure contains nested data structures meaning that struct types are expanded if and only if the structure contains only primitive types.
- The content of the linearized field is converted to java data type primitives.

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>.

If the optional header or filterable data field has other complex data types it is mapped to bytes stream, the JMS client that would receive a `bytes` stream have to use the appropriate CORBA Helper classes to unmarshal the user data.

Structured Event Remainder of body mapping

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 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 entered in JMS `StreamMessage` body.
- `String` type element only, it maps to 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`. To reconstruct the IDL type, the JMS client that would receive a `BytesMessage` have to use the appropriate CORBA Helper class.

2.5 QoS Properties Mapping

The Notification Service and the JMS have each specific QoS properties. In the Notification Service, QoS properties are specified in the header of Structured Events, Proxy, Admin and Channel object. In the JMS, QoS Properties are specified in the header of JMS messages and some objects in JMS Provider. This section describes the bi-directional mapping of QoS properties between the Notification Service and the JMS.

2.5.1 Event Reliability

The Notification Service's QoS property `EventReliability` is mapped to JMS QoS property `JMSDeliveryMode`. Each value of the properties are mapped as follows:

<code>EventReliability</code>	<code>JMSDeliveryMode</code>
<code>BestEffort</code>	<code>NON_PERSISTENT</code>
<code>Persistent</code>	<code>PERSISTENT</code>

Table 2-2 Event Reliability

2.5.2 Connection Reliability

The Notification Service's QoS property `ConnectionReliability` is mapped to `QueueReceiver` object in the JMS Point-to-Point model or `TopicSubscriber` object in the JMS Publish/Subscribe model. Each value of the property is mapped as follows:

<code>ConnectionReliability</code>	JMS Point-to-Point model	JMS Publish/Subscribe model
<code>BestEffort</code>	[no supported]	<code>TopicSubscriber</code>
<code>Persistent</code>	<code>QueueReceiver</code>	<code>Durable TopicSubscriber</code>

Table 2-3 Connection Reliability mapping

The JMS Publish/Subscribe model has both durable and not durable subscriber objects. Each subscriber is mapped to the corresponding value of `ConnectionReliability`. The JMS Point-to-point model has only the durable receiver object `QueueReceiver`. It is mapped to the `Persistent` value of `ConnectionReliability`. Since the JMS Point-to-point model does not have not durable subscriber objects, `BestEffort` of `ConnectionReliability` can't be used in interworking with JMS Point-to-Point model.

2.5.3 Delivery Reliability

The Notification Service's QoS property `DeliveryReliability` is an additional property of this specification (see Section 2.6.6, "QoS Properties for Reliable Event Delivery"). This QoS property is mapped to JMS reliable messaging functions. Each value of the property is mapped as follows:

<code>DeliveryReliability</code>	JMS reliable messaging functions
None	[no use of any reliable messaging functions]
Acknowledgment	Message Acknowledgment

Table 2-4 Delivery Reliability mapping

2.5.4 Priority

The Notification Service's QoS property `Priority` is mapped to the JMS QoS property `JMSPriority`. The value of the Notification Service's `Priority` is represented by short integer, where $-32,767$ is the lowest priority and $32,767$ the highest. The `JMSPriority` is represented by ten values, where 0 is the lowest priority and 9 the highest. Since the range of the value is very different between Notification Service and JMS, this specification defines following priority mapping as default mapping which must be supported. Other mappings may be supported in addition to the default mapping if necessary.

When the value of `JMSPriority` is converted to the value of Notification Service's `Priority`, same value on `JMSPriority` is used as the value of Notification Service's `Priority` as follows:

JMSPriority	Priority
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9

Table 2-5, Priority mapping from JMS to Notification

When the value of Notification Service's `Priority` is converted to the value of `JMSPriority`, some values on Notification Service's `Priority` are integrated with value 0 or 9 on `JMSPriority` as follows:

Priority	JMSPriority
-32,767 ... 0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9 ... 32,767	9

Table 2-6 Priority mapping from Notification to JMS

2.5.5 Expiry times

The Notification Service's QoS property `StopTime` is mapped to the JMS QoS property `Timeout`.

2.5.6 Order Policy

The JMS has only one order policy: messages must be received in the order they were sent. However messages of higher priority may jump lower priority messages. Thus JMS's order policy can be mapped to one of two values in the Notification Service's QoS property `OrderPolicy` as follows:

OrderPolicy	JMS's order policy
AnyOrder	[not supported]
FifoOrder	Supported
PriorityOrder	
DeadlineOrder	[not supported]

Table 2-7 Priority mapping

The selection of the mapping depends on implementation or system configuration. However when the `PriorityOrder` value is selected, the order of same priority events must be preserved.

Even if well-defined translation is executed on event channel between some Structured Events on a logical connection and an event batch (sequence of Structured Event), event order must be preserved for all events in `FifoOrder` value or for same priority events in `PriorityOrder` value Figure 2-6 and Figure 2-7. The first event on the logical connection is translated from/to the top of the event batch, and the last event on the logical connection is translated from/to the bottom of the event batch.

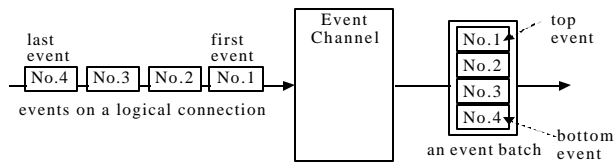


Figure 2-6 Preservation of event order in translation from Structured Events to event batch

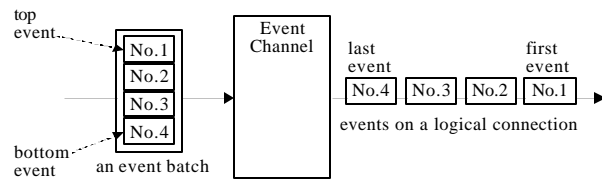


Figure2-7 Preservation of event order in translation from event batch to Structured Events

2.6 Acknowledgment Mapping

The JMS QoS API has two kinds of function for reliable messaging on the JMS MessageConsumer side:

- one using transactions, and;
- the other using Message Acknowledgment.

The use of acknowledgments or transactions provide different form of reliability. These two forms represent different use models: The use of acknowledgments provides a model where each message can be guaranteed to be delivered. If the message is not delivered, the message queue or topic can take steps to redeliver it. While this could be done within a transaction, the transactional model is fairly heavyweight for a single event, such as assuring the delivery of a single message. However, messages can also be part of a more complex set of actions, and in that context it makes a great deal of sense to include a message in a distributed transaction. For example, there may be a sequence of events where upon the receipt of a message, a database must be updated, and then a message sent to trigger some additional processing. The application designer might choose to bracket these actions within a transaction, so that if the message is not successfully delivered, all of the additional actions can be rolled back. If a message is part of a transaction, then the acknowledgment semantics are not used. The two models exist because there are a range of possible applications, not all of which would require transactions.

This section describes the mapping between the JMS Message Acknowledgment and the Notification Service.

2.6.1 Overview of Reliable Event Delivery with Event Acknowledgment

According to the JMS specification, reliable messaging with JMS Message Acknowledgment shall satisfy the following conditions:

- No lost messages
- No duplicate delivery of messages
- Preserve message order

However the Notification Service lacks the required acknowledgment functions to satisfy these conditions. This specification adds the required acknowledgment functions, called Event Acknowledgment, to the Notification Service so that the JMS Message Acknowledgment can be mapped to the Notification Service.

To realize the Event Acknowledgment satisfying the conditions above, this specification defines `DeliveryReliability` QoS property, `AckNotify` interface, `SequenceNumber` header field and Reliable Delivery Sequence.

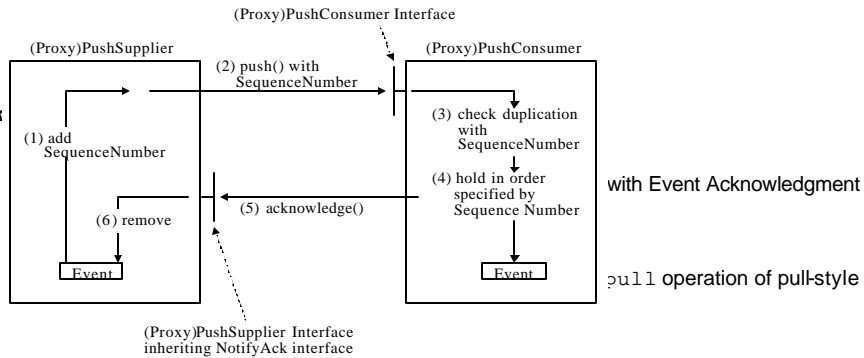
- `DeliveryReliability` QoS property specifies what reliable event delivery mechanism is used. When the value `Acknowledgment` is specified, the Event Acknowledgment is applied to event delivery.
- `AckNotify` interface defines an `acknowledge` operation. It is inherited by `StructuredPushSupplier`, `StructuredPullSupplier`, `SequencePushSupplier` and `SequencePullSupplier` interface to add the `acknowledge` operation to their interfaces.
- `SequenceNumber` header field indicates a serial number on each event. It is used to check event order and duplication of same event.
- Reliable Delivery Sequence defines a sequence of reliable event delivery with the `acknowledge` operation and the `SequenceNumber` header field. The Reliable Delivery Sequence is applied to event delivery between supplier (or proxy supplier) and consumer (or proxy consumer) when the value of `DeliveryReliability` property is `Acknowledgment`.

2.6.1.1 Event Acknowledgment on Push Model

Figure 28 shows an overview of reliable event delivery with Event Acknowledgment using the push model.

1. Push-style supplier (or proxy supplier) adds `SequenceNumber` header field to an event.
2. Supplier sends the event to push-style consumer (or proxy consumer) invoking `push` operation.
3. Consumer checks duplication of the received event using the `SequenceNumber`.
4. Consumer holds the event in order specified by the `SequenceNumber`.
5. Consumer invokes `acknowledge` operation
6. Supplier removes the sent and acknowledged event.

2.6.1.2 Event Ack



2. Supplier adds `SequenceNumber` header field to an event.
3. Supplier sends the event to consumer as return value of invoked `pull` operation.
4. Consumer checks duplication of the received event using the `SequenceNumber`.
5. Consumer holds the event in order specified by the `SequenceNumber`.
6. Consumer invokes `acknowledge` operation.
7. Supplier removes the sent and acknowledged event.

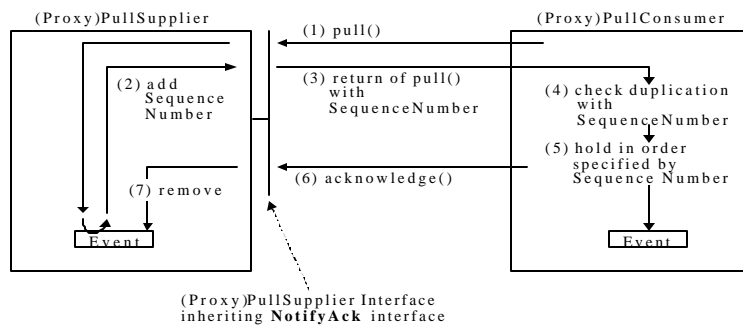


Figure 2-9 Event Acknowledgment on push model

2.6.2 Scope of Event Acknowledgment

Event Acknowledgment supports delivery of Structured Events and delivery of event batches (sequence of Structured Events) for both the push model and the pull model.

Note: The notification channel mechanisms for translation of Typed events and untyped events (i.e., syntax Any) from a proxy consumer to a proxy supplier, can be used to convert such notifications to structured event syntax. Thus, it is sufficient to specify event acknowledgment extensions only for Structured events and sequence of structured events.

Event Acknowledgment can be applied to any logical connection between a supplier (or proxy supplier) and a consumer (or proxy consumer), even if the supplier or the consumer is a Bridge for interworking with JMS (see Figure 2-10).

Applying Event Acknowledgment to all the logical connections at same time on an event domain can realize end-to-end reliability between supplier and consumer. Each Event Acknowledgment on a logical connection is managed independently. Figure 2-10 shows an example of Reliable Delivery Sequence with end-to-end reliability on an event domain using the push model. Since the Reliable Delivery Sequence on the logical connection 1 is managed independently from the next logical connection 2, when “(1) push” operation is invoked, the event channel A may invoke “(2) acknowledge” operation soon thereafter. The event channel A is not required to invoke “(3) push” operation or wait for “(4) acknowledge” operation on the next logical connection before invocation of “(2) acknowledge”.

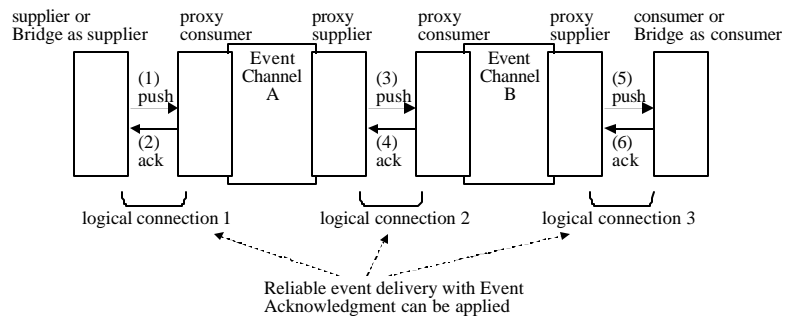


Figure 2-10 Event Acknowledgment on event domain in push model

Note: When the Event Acknowledgment is used on a logical connection in an event domain, the supplier (not proxy supplier) can create and add `SequenceNumber` header field (see Section 2.6.3, “Sequence Number Header Field”) to events for improvement of performance. Because insertion of `SequenceNumber` header field by proxy supplier would require a lot of overhead.

2.6.3 Sequence Number Header Field

The following definition is added to the `CosNotification` module for `SequenceNumber` header field:

```
const string SequenceNumber = "SequenceNumber";  
// SequenceNumber takes a value of type long.
```

The `SequenceNumber` header field is an event identifier defined as a standard optional header field. The type of its associated value is `long`. When the Event Acknowledgment is applied to event delivery, the supplier (or proxy supplier) adds the header field to the variable header in the Structured Event before sending the event to the consumer (or proxy consumer). In the case of delivery of an event batch, the supplier adds the header field to only the first Structured Event in the sequence of Structured Events. If the `SequenceNumber` header field was already added for previous event delivery, the event channel overrides the `SequenceNumber` header field with a new value. The consumer uses the `SequenceNumber` header field to:

- Detect and remove duplicate events
- Detect and correct invalid order of events

The `SequenceNumber` is an integer value which takes a value in the range $0..2^{31}-1$. It is created and managed per each logical connection between supplier and consumer. In the first event or event batch within the logical connection, `SequenceNumber` takes the value 0. It is incremented (ex. 0, 1, 2, ..) for each event (in the case of delivery of Structured Event) or for each event batch (in the case of delivery of sequence of Structured Events) sent by the supplier within the logical connection. The next value of $2^{31}-1$ in the increment is 0. Only when the logical connection is disconnected explicitly by invocation of `disconnect` operations, `SequenceNumber` for the logical connection is reset to 0. Otherwise `SequenceNumber` is never reset.

Note – The associated value of the `SequenceNumber` header field takes a positive value or 0. However `long` type is applied to the value rather than `unsigned long` for natural mapping with Java. It is a design policy of the Notification Service specification.

2.6.4 Notify Ack Interface

The `JMSMessageConsumer` can explicitly acknowledge receipt of messages in both synchronous and asynchronous messaging using the `acknowledge` method of `JMS MessageObject`. However the Notification Service lacks an operation which can be mapped with the `acknowledge` method of the `JMS MessageObject`. This specification defines an additional interface, `AckNotify`, which has an

acknowledge operation to be mapped with the acknowledge method of the JMS MessageObject.

This specification maps JMS messages with Notification Service's Structured Events. The acknowledge operation is therefore added to the StructuredPushSupplier, StructuredPullSupplier, SequencePushSupplier, SequencePullSupplier and their proxy interfaces through inheritance of the AckNotify interface.

The following types and interface are added to the CosNotifyComm module.

```
typedef sequence<long> SequenceNumbers;
interface NotifyAck {
    void acknowledge(SequenceNumbers sequence_numbers);
};
```

The StructuredPushSupplier, StructuredPullSupplier, SequencePushSupplier and SequencePullSupplier interfaces in CosNotifyComm module are changed (creating new versions) as follows, to inherit the NotifyAck interface. By this change, the NotifyAck interface is also inherited by the StructuredProxyPushSupplier, StructuredProxyPullSupplier, SequenceProxyPushSupplier and SequenceProxyPullSupplier interfaces in CosNotifyChannelAdmin module.

```
interface StructuredPushSupplier : NotifySubscribe, NotifyAck {
    ...
};

interface StructuredPullSupplier : NotifySubscribe, NotifyAck {
    ...
};

interface SequencePushSupplier : NotifySubscribe, NotifyAck {
    ...
};

interface SequencePullSupplier : NotifySubscribe, NotifyAck {
    ...
};
```

2.6.4.1 Acknowledge

The acknowledge operation in the NotifyAck interface causes the supplier (or proxy supplier) to acknowledge that the consumer (or proxy consumer) received the events which the supplier sent previously. When the operation is invoked, the supplier may remove sent events indicated by the SequenceNumbers input parameter, which

specifies values of the `SequenceNumber` header field in the received events by the consumer.

A Consumer does not always need to invoke the `acknowledge` operation after each invocation of `push` or `pull` operation. A Consumer may convey multiple `SequenceNumber` values of received events to the supplier at once by one invocation of `acknowledge` operation after some invocations of `push` or `pull` operations.

2.6.5 Reliable Delivery Sequence

To realize reliable event delivery with Event Acknowledgment, the Notification Service supplier (or proxy supplier) and consumer (or proxy consumer) must support the Reliable Delivery Sequence, which detects a lost event (or event batch) at system failure or communication error and recovers it.

2.6.5.1 *Reliable Delivery Sequence for Push Model*

The Reliable Delivery Sequence using the push model consists of the following steps:

1. The supplier sends events to the consumer by invocations of the `push` operation of the consumer.
2. The supplier detects possibility, in the invocations of `push` operation, that the events were lost at system failure or communication error using exceptions or timeout of acknowledgment. The timeout means that the supplier's `acknowledge` operation is not invoked after the invocations of `push` operation for a time which is specified by QoS parameter `RetryInterval`. If the supplier does not detect the possibility of lost events, it jumps to step (4).
3. The supplier retries the invocations in the step (1) to re-send the events to supplier.
When the invocations have failed again due to system failure or communication error, the supplier repeats the same invocations until succeeds in the invocations or the total number of retries specified by QoS parameter `Retries` is satisfied. The interval between original invocations and first retry, or between retries is specified by QoS parameter `RetryInterval`.
4. The consumer checks the received events in duplication and order of event using `SequenceNumber` header field.
If it is not received events previously, the consumer checks the order of the events and stores them in persistent storage. If the order is invalid, the

consumer corrects the order at passing the events to next event delivery or application.

If they are received events previously, the consumer ignores the events.

5. The consumer invokes `acknowledge` operation of the supplier to notify the supplier of successful of the event delivery. The event channel removes the events specified by the `SequenceNumbers` parameter of the `acknowledge` operation from persistent storage. Even if the consumer detects possibility in the invocations of `acknowledge` operation that the acknowledgment was lost at system failure or communication error using exceptions, the consumer does not need to retry the `acknowledge` operation in this step. Because if the acknowledgment was lost, the supplier retries the `push` operation (see step (2)). As the result, the consumer will execute this step again.

The retry count is managed per each logical connection. When invocation of `acknowledge` operation is successful or the logical connection is disconnected explicitly by invocation of `disconnect` operations, the retry count is reset to 0.

2.6.5.2 Reliable Delivery Sequence for Pull Model

The Reliable Delivery Sequence using the pull model consists of the following steps:

1. The consumer receives events from the supplier by invocations of the `pull` operation of the supplier.
2. The consumer detects possibility, in the invocations of the `pull` operation, that the events were lost at system failure or communication error using exceptions.
If the supplier does not detect the possibility of lost events, it jumps to step (4).
3. The consumer retries the invocations in the step (1) to re-receive the events from the supplier.
When the invocations have failed again due to system failure or communication error, the consumer repeats the same invocations until succeeds in the invocations or the total number of retries specified by QoS parameter `Retries` is satisfied. The interval between original invocations and first retry, or between retries is specified by QoS parameter `RetryInterval`.
4. The consumer checks the received events in duplication and order of event using `SequenceNumber` header field.
If it has not received events previously, the consumer checks the order of the

events and stores them in persistent storage. If the order is invalid, the consumer corrects the order at passing the events to next event delivery or application.

If they are received events previously, the consumer ignores the events.

5. The consumer invokes `acknowledge` operation of the supplier to notify the supplier of successful of the event delivery. The event channel removes the events specified by the `SequenceNumber s` parameter of the `acknowledge` operation from persistent storage.
6. If the consumer detects possibility in the invocations of `acknowledge` operation that the acknowledgment was lost at system failure or communication error using exceptions, the consumer retries the invocation. When the invocation has failed again due to system failure or communication error, the consumer repeats the same invocation until succeeds in the invocation or the total number of retries specified by QoS parameter `Retries` is satisfied. The count of retries for `acknowledge` operation is individual from the count of retry for `pull` operation in step (3). The interval between original invocation and first retry, or between retries is specified by the QoS parameter `RetryInterval`.

The retry count is managed per each logical connection. When invocation of `acknowledge` operation is successful or the logical connection is disconnected explicitly by invocation of `disconnect` operations, the retry count is reset to 0.

2.6.5.3 Recovery in Failure of Retries

When the supplier (or proxy supplier) or the consumer (or proxy consumer) fails in all the retries, the supplier or the consumer stops the event delivery on the logical connection, and reports the unrecoverable failure to system administrators.

How to report and recover the failure is out of scope of the specification. The following recovery schemes are shown for example:

The system administrators resolve the failure by hand and then:

1. Reset the retry count and restart the event delivery on the logical connection,
or
2. Invoke `disconnect` operation for the logical connection and then invoke `connect` operation to restart the event delivery (the `SequenceNumber` and retry count are reset by the `disconnect` operation).

In the first scheme, non duplication and event order semantics are preserved between before the failure and after it. But in the second scheme, the semantics might be lost between before the failure and after it.

2.6.6 QoS Properties for Reliable Event Delivery

The Event Acknowledgment uses three additional QoS properties, `DeliveryReliability`, `Retries` and `RetryInterval`.

2.6.6.1 *DeliveryReliability*

The Notification Service has no way to specify what mechanism is used for reliable event delivery. The specification defines additional QoS property `DeliveryReliability` to provide this way. Following definition is added to `CosNotification` module to define this QoS property.

```
const string DeliveryReliability = "DeliveryReliability"
const short None = 0;
const short Acknowledgment = 1;
```

The QoS property specifies a mechanism used by a given supplier (or proxy supplier) and consumer (or proxy consumer) to realize reliable event delivery. Constant values to represent the following setting are defined:

- None – Any reliable delivery mechanism is not applied to event delivery
- Acknowledgment – The Event Acknowledgment described in the specification is applied to event delivery

Table 2-8 shows possible combinations of related QoS properties when the `DeliveryReliability` property is set to `Acknowledgment`.

	Event Reliability	Connection Reliability	Delivery Reliability	Description
combination 1	BestEffort	BestEffort	Acknowledgment	Implementations may support these combinations. But they can't realize complete reliability.
combination 2	BestEffort	Persistent	Acknowledgment	
combination 3	Persistent	BestEffort	---	This combination has no meaning and need not be supported (according to the Notification Service specification).
combination 4	Persistent	Persistent	Acknowledgment	Implementations must support this combination for complete reliability.

Table 2-8 Combination of related properties for Event Acknowledgment

2.6.6.2 Retries

The following definition is added to `CosNotification` module for `Retries` QoS property:

```
const string Retries = "Retries"  
// Retries takes on a vlaue of type long
```

The QoS property `Retries` specifies minimum number of retries in the Reliable Delivery Sequence. The type of associated value is `long`.

2.6.6.3 RetryInterval

The following definition is added to `CosNotification` module for `RetryInterval` QoS property:

```
const string RetryInterval = "RetryInterval"  
// RetryInterval takes on a vlaue of TimeBase::TimeT
```

The QoS property `RetryInterval` specifies interval between original invocation and first retry or between retries. The type of associated value is `TimeBase::TimeT`.

2.6.6.4 Supported level of the QoS Properties

Supported level of the QoS property is described in following table.

Property	Per-Message	Per-Proxy	Per-Admin	Per-Channel
DeliveryReliability		X	X	X
Retries		X	X	X
RetryInterval		X	X	X

Table 2-9 Levels at which setting the QoS properties for Reliable Event Delivery is supported

The admin level setting overrides the channel level setting, and proxy level setting overrides the admin level or channel level setting. Note that their properties have no meaning if set on a per-message basis.

2.6.7 Mapping between Event Acknowledgment and JMS Message Acknowledgment

The JMS Message Acknowledgment is mapped to the Notification Service's Event Acknowledgment. On one hand the `JMS MessageConsumer` has sending functions of acknowledgment, on the other hand the `JMS MessageProducer` does not have received functions of acknowledgment. Thus the acknowledgment functions between JMS and Notification Service is mapped on only consumer side Bridge.

The JMS `MessageConsumer` has two kinds of message delivery mode: asynchronous delivery with `onMessage` method and synchronous delivery with `receive` or `receiveNoWait` method. The asynchronous delivery is mapped to the Notification Service's push model as follows:

push model (XXXXXX: Structured or Sequence)	JMS asynchronous delivery
push operation in XXXXXPushConsumer interface	<code>onMessage</code> method without Message Acknowledgment
push operation in XXXXXPushConsumer interface and acknowledge operation in XXXXXProxyPushSupplier interface	<code>onMessage</code> method with <code>DUPS_OK_ACKNOWLEDGE</code> <hr/> <code>onMessage</code> method with <code>AUTO_ACKNOWLEDGE</code>
push operation in StructuredPushConsumer interface and acknowledge operation in StructuredProxyPushSupplier interface	<code>onMessage</code> method and acknowledge method of <code>CLIENT_ACKNOWLEDGE</code>

Table 2-10 Asynchronous delivery mapping

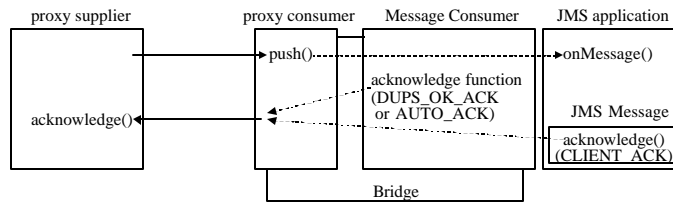


Figure 2-11 Asynchronous delivery mapping

The synchronous delivery is mapped to the Notification Service's pull model as follows:

waiting policy	pull model (XXXXXX: Structured or Sequence)	JMS synchronous delivery
waiting for message	pull operation in XXXXXProxyPullSupplier interface	<code>receive</code> method without Message Acknowledgment
	pull and acknowledge operations in XXXXXProxyPullSupplier interface	<code>receive</code> method with <code>DUPS_OK_ACKNOWLEDGE</code> <hr/> <code>receive</code> method with <code>AUTO_ACKNOWLEDGE</code>
	pull and acknowledge operations in StructuredProxyPullSupplier interface	<code>receive</code> method and acknowledge method of <code>CLIENT_ACKNOWLEDGE</code>

no waiting	try_pull operation in XXXXXProxyPullSupplier interface	receiveNoWait method without Message Acknowledgment
	try_pull and acknowledge operations in XXXXXProxyPullSupplier interface	receiveNoWait method with DUPS_OK_ACKNOWLEDGE receiveNoWait method with AUTO_ACKNOWLEDGE
	try_pull and acknowledge operations in StructuredProxyPullSupplier interface	receiveNoWait method and acknowledge method of CLIENT_ACKNOWLEDGE

Table 2-11, Synchronous delivery mapping

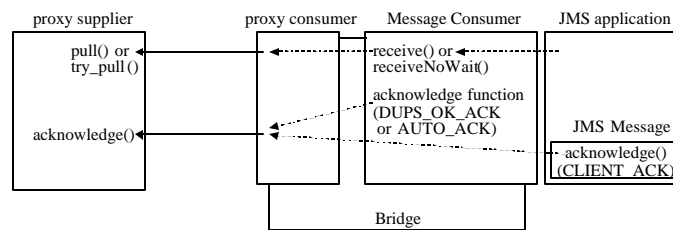


Figure 2-12 Synchronous delivery mapping

When the JMS MessageConsumer works with DUPS_OK_ACKNOWLEDGE or AUTO_ACKNOWLEDGE mode, the Bridge as Notification Service consumer sets the value of SequenceNumber header field of received event (or event batch) to the SequenceNumbers input parameter and invokes acknowledge operation whenever an event is received. When DUPS_OK_ACKNOWLEDGE is specified, the Bridge may omit the check of duplicated events.

When the JMS MessageConsumer works with CLIENT_ACKNOWLEDGE mode, the Bridge as Notification Service consumer sets one or more values of SequenceNumber header field of received events (or event batches) to the SequenceNumbers input parameter and invokes acknowledge operation only when the JMS application calls the acknowledge method of the JMS MessageObject.

When Notification Service's Event Acknowledgment is applied for mapping of JMS Message Acknowledgment, following conditions must also be satisfied to apply combination 4 of the related QoS properties (see "2.6.6.1 DeliveryReliability"):

- The EventReliability QoS property is set to Persistent, and
- The ConnectionReliability QoS property is set to Persistent.

Note – When the Notification Service push model is mapped with the JMS DUPS_OK_ACKNOWLEDGE or AUTO_ACKNOWLEDGE mode, invoking acknowledge

operation is actually not needed. Because the supplier can know that sent event reached the consumer by the end of push operation. But the specification forces consumer to invoke acknowledge operation to realize consistent design of operations among mapping with DUPS_OK_ACKNOWLEDGE, AUTO_ACKNOWLEDGE and CLIENT_ACKNOWLEDGE mode, and consistent design of operations among push and pull model.

2.7 Transactional Support

2.7.1 Asynchronous Transactional Model

To maintain the transaction semantic from a Notification Service client to a JMS client, at least three different transactions are needed:

1. One transaction (T1) that involves a notification client, its transactional resources (e.g a database) and the notification service.
2. One transaction (T2) that involves the Notification Service , the bridge and the JMS implementation. This transaction will be named Routing transaction.
3. One transaction (T3) that involves a JMS client, its transactional resources (e.g a database) and the JMS implementation.

These transactions are tied by precedence rules:

When Sending an event from an event supplier to a JMS consumer the precedence rules imply that :

- The transaction T3 will start only and only if T2 commits,
- T2 will start only and only if T1 commits.
- Each time one of these transactions commits the data sent in its scope is moved toward its next destination.
- If one of those transactions rolls back, the data sent in its scope will be put back into its initial destination. Subsequent trials will take place to send the data to its next destination later.

These rules imply that once data is sent from the supplier, it will be conveyed to its final destination that is the consumer, guaranteeing the transaction semantic from end-to-end.

When Sending a JMS message from a JMS sender (or publisher) to an event consumer the precedence rules are reversed, meaning that T1 will not take place only and only if T2 commits and T2 will not take place only and only if T3 commits.

Although the usage of end-to-end transaction semantic is recommended this specification does not mandate it. This specification covers the routing transaction (T2) only which implies that a notification service client can send or receive events in a non-transactional context, likewise the JMS client can send or receive JMS messages in a non-transactional context, but the data sent between notification service and JMS may be sent in the routing transaction context.

The rationale behind the use of the routing transaction without necessarily using transactions at the Notification Service and JMS client sides is that the Notification Service to JMS communication portion is hidden and transparent to the application client developers preventing them from performing any recovery action if a failure, that will lead to data loss, occurs.

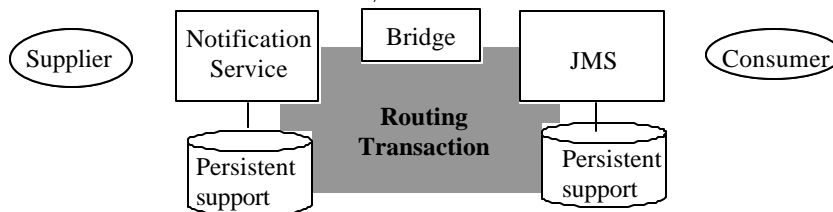


Figure 2-13 Routing Transaction Scope

To guarantee the ACID transactional properties of the Routing Transaction it is expected that the notification service and the JMS implementation rely on a transactional persistent support.

Applying end-to-end transaction semantic and using transactional persistent support in an effective way will guarantee the exactly-once delivery QoS.

The Figure 2-13 denotes the Routing transaction scope. This transaction involves the bridge, the Notification Service, the JMS service provider and the Persistent Supports used by them. All those components should be coordinated by a single root coordinator which is hosted by a Transaction Service.

It is also expected that the interposition schema is applied between OTS (Object Transaction Service) on the CORBA side and JTS (Java Transaction Service) on the Java side. The interoperability between OTS and JTS is guaranteed by the fact that JTS is mapped from OTS and its usage of IIOP as an underlying transport protocol to propagate transaction context between OTS and JTS.

Both JMS and Notification specifications provide a model that outlines how a messaging system should behave in a transactional environment. The transactional roles of the JMS, the Bridge and the Notification Service are driven by the following considerations:

- The Notification Service specification allows a channel and its related proxy objects to initiate a transaction and assumes the transaction client role.
- The Notification Service specification allows a channel and its related proxy queue objects to assume the Resource object role.
- The JMS service provider doesn't support the transactional client role.
- The Bridge should be as light as possible, meaning that it should not be assigned complex transactional behaviors. This assertion will promote the bridge adoption by the industry.

2.7.2 Supported configurations

By combining the communication consistency checking defined in Table 2-1 and the state of the art of the transactional role of JMS and Notification, the only case where events can be sent from Notification service to JMS in transaction scope is the case where the Channel pushes the data to the bridge and the bridge pushes it to JMS. Likewise, the only case where data can be sent from JMS to Notification service in transaction scope is the case where the Channel pulls data from the bridge that will synchronously pull it from JMS.

In both cases the Notification Service assume the Routing Transaction client role.

The Figure 2-14 summarizes the supported transactional configurations. The Transactional Roles of different components are detailed in the next sections.

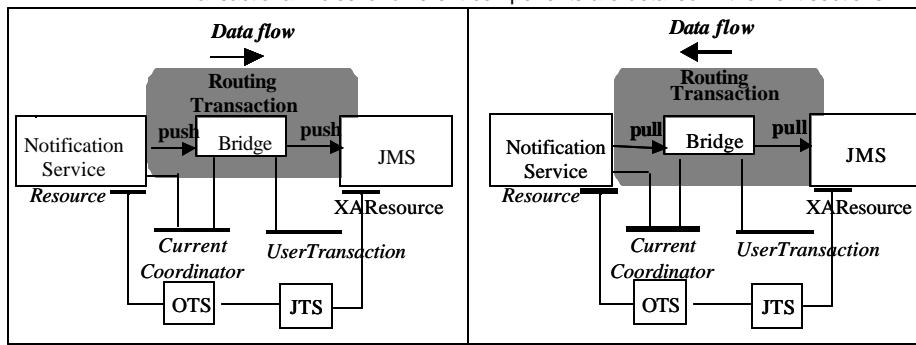


Figure 2-14 Supported transactional configuration

2.7.3 Notification Service Transactional Role

When events are sent in the routing transaction scope, the Notification Service assumes always the transaction client and recoverable server roles, and as such the

Notification Service: Starts a transaction may use the `OTS Current` interface; Enlist a Resource object that wraps events queues using the `OTS Coordinator` interface.

The notification service may also use direct transactional context management, by using the `OTS Control` interface, to manage the routing transaction.

The Notification Service queue managers participate into the two-phase commit completion and the recovery protocols by implementing the Resource Object interface and the transactional recovery protocol as it is specified in the OTS specification. The behavior of the Notification Service Resource Objects depends on whether the Notification Service is the source or the sink of the data.

2.7.3.1 Data Flowing from Notification Service to JMS

When the Notification Service sends one or several events in to the routing transaction scope, the OTS transaction identifier is propagated implicitly to JTS in the Propagation context.

If the events are sent successfully the Notification Service asks the Transaction Service to commit the routing transaction. When the root transactional coordinator decides definitively to commit the routing transaction the events associated to it are removed from Notification Service events queue.

If the events were not correctly sent or if the root transactional coordinator rolls back the routing transaction for any other reason, the events sequences associated with it will remain in events queue, these events will be sent later when the notification service will start a new transaction.

2.7.3.2 Data flowing from JMS to Notification Service

When the Notification Service receives one or several events in to the routing transaction scope, the OTS transaction identifier is propagated implicitly to JTS in the Propagation context.

If all the events are received and stored into the transactional persistent support successfully the Notification Service asks the Transaction Service to commits the routing transaction. When the root transactional coordinator decides definitively to commit the routing transaction the events associated to it are durably added in to the Notification Service events queue.

If the events were not correctly received or if the root transactional coordinator rolls back the routing transaction for any other reason, the events that are potentially received in its scope will be deleted from the events queue.

2.7.4 Bridge Transactional Roles

In order to keep the bridge simple, one design principle was to forbid attributing to it the transaction client role. Furthermore, this specification does not define any recoverable state and does not implement any transactional change at the Bridge level.

The bridge does not participate to the transaction completion protocol, but it can force the transaction roll back. A typical case is when the bridge is unable to carry the data to the JMS or to the Notification Service. To rollback the transaction the bridge may use either the JTA/JTS or OTS interfaces. Therefore, the bridge is assuming the transactional object role.

2.7.5 JMS service provider Transactional Role

The JMS specification does not attribute any transaction client role, meaning that the JMS service provider is not allowed to initiate or start any transaction.

The JMS specification assigns to JMS a Resource Manager role, meaning that it can integrate the sphere of control of the routing transaction by enlisting its transactional queue manager using the `javax.transaction.Transaction` interface and implements the transactional semantic on the message queues. The JMS queue managers participate into the two-phase commit completion and the recovery protocols by implementing the `javax.transaction.xa.XAResource`, `javax.jms.XAConnection` and the `javax.jms.XASession` interfaces and the transactional recovery behavior as it is specified in the JTA specification.

The behavior of JMS `XAResource` objects depends on whether the JMS Service is the source or the sink of the data.

2.7.5.1 Data Flowing from Notification Service to JMS

Due to the considerations described in section 2.7.2 only the notification service is the component that is allowed to initiate routing transactions. When the Notification Service sends successfully events that are translated to messages to JMS and commits the routing transaction all the messages are durably stored in the JMS message queues and the transactional persistent support.

If the messages were not correctly sent or if the root transactional coordinator rolls back the routing transaction for any other reason the messages are potentially sent in the routing transaction scope will be removed from the JMS message queue and the transactional persistent support. The messages will be received later when the notification service will start a new transaction.

2.7.5.2 Data Flowing from JMS to Notification Service

When the bridge pulls JMS in to the scope of the routing transaction the OTS transaction identifier is propagated implicitly to JTS in the Propagation context. The propagation context will be in turn propagated to the JMS `XAResource` Object that encapsulates the message queues.

If the JMS returns successfully the messages in to the routing transaction scope and if the notification service receives them successfully the Notification Service asks the Transaction Service to commits the routing transaction. When the root transactional coordinator decides definitively to commit the routing transaction the events associated to the routing transactions are durably removed from the JMS message queue.

If the translated messages to events were not correctly received into the notification service or if the root transactional coordinator rolls back the routing transaction for any other reason, the messages that are potentially sent in the routing transaction scope will remain in the JMS message queue.

2.7.6 Bridge Transactional Monitoring

2.7.6.1 Notification Service QoS and Admin Property Extensions

The notification service specification states that in order to support transactional event transmission, an implementation of the Notification Service should support implementations of the various proxy interfaces that are POA objects that support TransactionPolicy. Unfortunately the specification was not precise enough to define the way an application program will dynamically control the transactionality of its proxy in the notification service. Furthermore, the specification did not offer to the developer the way to specify the number of events that are sent or retrieved by the event channel in the scope of a transaction. To make up for those gaps we propose to extend the Notification Service quality of services and administration properties Framework by a new QoS and three administrative properties.

Fortunately, the Notification Service QoS and Admin frameworks are flexible enough to add new QoS and AdminProperty values without changing the Notification Service interfaces. Therefore the new QoS and AdminProperties should be seen as an extension rather than a modification of the Notification Service Interfaces. These new QoS and AdminProperties are:

- **EnableTransaction** QoS is a boolean that enables the notification service client to activate or deactivate the support of the transaction at

Notification Service object levels. When this QoS is enabled and applied on the `ProxyPushSupplier`, `ProxyPullConsumer`, `StructuredProxyPushSupplier`, `StructuredProxyPullConsumer` and `TypedProxyPushSupplier`, `TypedProxyPullConsumer`⁶ levels it will allow the later to behave as a transaction client. When this QoS is disabled and applied on those various types of proxy their transactional client behavior is disabled. This is their default behavior. When the `EnableTransaction` QoS is enabled at the `ProxyPushConsumer`, `ProxyPullSupplier`, `StructuredProxyPushConsumer`, `StructuredProxyPullSupplier` `TypedProxyPushConsumer`⁶, and `TypedProxyPullSupplier`⁶, the proxies' implementations will set their `TransactionalPolicy` to `Require_shared`. By default this QoS is disabled, meaning that for the `ProxyPushConsumer` various types and the `ProxyPullSupplier` various types the proxies' POAs `TransactionalPolicy` attributes are set to `Allows_none`.

If this QoS is applied at the `SupplierAdmin`, `ConsumerAdmin`, `TypedSupplierAdmin`⁶ or `TypedConsumerAdmin`⁶ levels each of their proxy child will enable individually this QoS at their level according to their types. If this QoS is applied at the channel, respectively `TypedChannel`⁶ level all the `SupplierAdmin` and the `ConsumerAdmin`, respectively, all the `TypedSupplierAdmin` and the `TypedConsumerAdmin` objects will enable this QoS, subsequently all the proxy objects apply it individually.

Whenever the `EnableTransaction` QoS is enabled the `EventReliability` and the `ConnectionReliability` QoSs will be setup automatically by the Notification Service to "Persistent". Likewise, when this QoS is disabled the `EventReliability` and the `ConnectionReliability` are set to "BestEffort".

- **TransactionEvents** AdminProperty defines the number of separate events sent in the scope of a transaction. The scope of this property is the `ProxyPushSupplier`, `ProxyPullConsumer`, `StructuredProxyPushSupplier`,

⁶ Although this specification is not directly concerned by the generic and Typed Events it specifies the transactional behavior of their proxies. This is because this section is aiming to update the section 2-12 of the Notification Service specification.

StructuredProxyPullConsumer, TypedProxyPushSupplier and TypedProxyPullSupplier.

- **TransactionEventSequences⁷** AdminProperty defines the number of event sequences sent in the scope of a transaction. The scope of this property is the SequenceProxyPushSupplier and SequenceProxyPullConsumer.
- **TransactionTimeout** adminProperty defines the timeout period in number of seconds associated with routing transaction created. If the parameter has a non-zero value n, then the created routing transaction will be subject to being rolled back if they do not complete before n seconds after their creation. If its value is zero, then no application specified time-out is established. This adminProperty is aimed to be mapped on the unsigned long input parameter of the OTS Current.set_timeout() operation. This adminProperty is applied on all the proxies that behave as transaction clients.

When those adminProperty are applied at the SupplierAdmin, ConsumerAdmin or EventChannel level they will affect only the proxies with transaction client behavior. The Table 2-12 summarizes the scope of the new QoS and AdminProperties at the proxy level. It also summarizes proxies' transactional roles. Empty Cells denotes that QoS is not applicable.

	Proxy Types	QoS	AdminProperties		
		Enable Transaction	Transaction Timeout	Transaction EventSequences	Transaction Events
Transaction Client Role	ProxyPushSupplier,	X	X		X
	ProxyPullConsumer,	X	X		X
	StructuredProxyPushSupplier	X	X		X
	StructuredProxyPullConsumer	X	X		X
	SequenceProxyPushSupplier	X	X	X	
	SequenceProxyPullConsumer	X	X	X	
	TypedProxyPushSupplier ⁶ ,	X	X		X
	TypedProxyPullConsumer ⁶	X	X		X
Transaction Server Role	ProxyPushConsumer	X			
	ProxyPullSupplier	X			
	StructuredProxyPushConsumer	X			
	StructuredProxyPullSupplier	X			
	SequenceProxyPushConsumer	X			
	SequenceProxyPullSupplier	X			
	TypedProxyPushConsumer ⁶	X			
	TypedProxyPullSupplier ⁶	X			

Table 2-12 New Notification Service QoS and AdminProperties scope

2.7.6.2 Bridge Transaction Management Interface

The `TransactionManagement` interface provides the bridge application clients the ability to enable and disable automatically the routing transactions. This interface is optionally inherited by the `Bridge` interface.

This interface contains two operations: `enable_transaction()` and `disable_transaction()`.

The invocation of the `enable_transaction` operation will enable the proxies with a transaction client behavior (`StructuredProxyPushSuppliers`, `StructuredProxyPullConsumer`, `SequenceProxyPushSupplier`, `SequenceProxyPullConsumer`) to start routing transactions and manage their events queues as OTS recoverable objects. It also enables the bridge `EndPoint` objects that are connected to the notification service to set their Transactional POA Policies to `Require_shared`. The `EndPoint` objects affected by this operation are:

- `StructuredPushConsumer`,
- `SequencePushConsumer`,
- `StructuredPullSupplier`,
- `SequencePullSupplier`.

This operation takes as an input the number of events sent in the scope of the routing transaction and the routing transaction lifetime.

The invocation of the `disable_transaction` operation will disable subsequent routing transactions.

2.8 Conformance

The Transaction mapping capabilities, specified in Section 2.7, are an optional conformance point for this specification. All other interfaces defined in this specification are required to be implemented for conformance to this specification.

Deleted: <#>1

Formatted: Bullets and Numbering

3 Bridge Interfaces

This section describes the semantic and the behavior of the interfaces which make up the NS-JMS bridge. All the data structures and the interfaces are defined in the CoSBridgeAdmin module.

3.1 CosBridgeAdmin Module

The `CosBridgeAdmin` module defines the `ExternalEndPoint` data type. In addition, this module provides declarations for administrative interfaces which are defined for managing the Bridge Life cycle.

Deleted: , along with `EndPointSender` and `EndPointReceiver` types

Note that the `EndPointSender` and `EndPointReceiver` described in section 2 are implementation objects, therefore they are not modulised in the public IDL files.

```
#ifndef _COS_BRIDGE_ADMIN_
#define _COS_BRIDGE_ADMIN_
#include <orbdefs.idl>
#include <CosNotifyChannelAdmin.idl>
```

Deleted: #include `<CosNotifyComm.idl>`

```
#pragma prefix "omg.org"
```

Deleted: #include `<CosEventDomainAdmin.idl>`

```
module CosBridgeAdmin
{
    enum ExternalEndpointRole
    {
        SOURCE,
        SINK
    };

    enum JMSDestinationType
    {
        QUEUE,
```



```

TOPIC
};

enum MessageType
{
    JMS_MESSAGE,
    STRUCTURED_EVENT,
    SEQUENCE_EVENT
};

struct JMSDestination
{
    JMSDestinationType destination_type;
    string destination_name;
    string factory_name;
};

enum FlowStyle
{
    PUSH,
    PULL
};

union ExternalEndpointConnector switch (MessageType)
{
    case JMS_MESSAGE: JMSDestination destination;
    default: CosNotifyChannelAdmin::ChannelID channel_id;
};

struct ExternalEndpoint
{
    ExternalEndpointRole role;
    ExternalEndpointConnector connector;
    FlowStyle style;
    MessageType type;
};

enum ExternalEndpointErrorCode
{
    INVALID_CHANNELID,
    INVALID_JMSDESTINATION,
    MISMATCH_ENDPOINTROLE_NOTIFSTYLE
};

struct ExternalEndpointError
{
    ExternalEndpointRole role;

```

Deleted: CosEventDomainAdmin::NotificationStyle style;

```

ExternalEndpointErrorCode code;
};

typedef sequence<ExternalEndpointError> ExternalEndpointErrorSeq;

exception InvalidExternalEndPoints
{
    ExternalEndpointErrorSeq error;
};

typedef long BridgeID;
typedef sequence<BridgeID> BridgeIDSeq;

exception BridgeAlreadyStarted {};
exception BridgeInactive {};
exception BridgeNotFound {};

interface BridgeFactory;

interface Bridge
{
    readonly attribute ExternalEndpoint end_point_receiver;
    readonly attribute ExternalEndpoint end_point_sender;

    void start_bridge() raises (BridgeAlreadyStarted, InvalidExternalEndPoints);
    void stop_bridge() raises (BridgeInactive);
    status get_status();
    void destroy ();
};

interface BridgeFactory
{
    Bridge create_bridge (in ExternalEndpoint source, in ExternalEndpoint sink,
out BridgeID id)
        raises (InvalidExternalEndPoints );
    Bridge get_bridge_with_id (in BridgeID id)
        raises (BridgeNotFound);
    BridgeIDSeq get_all_bridges();
};
};

#endif

```

Deleted: enum
EndpointReceiverType¶
{¶
STRUCTURED_PUSH_CONS,¶
SEQUENCE_PUSH_CONS,¶
STRUCTURED_PULL_CONS,¶
SEQUENCE_PULL_CONS,¶
MESSAGE_LISTENER,¶
MESSAGE_CONSUMER¶
};¶

Deleted: union
EndpointReceiver switch
(EndpointReceiverType)¶
{¶
case
STRUCTURED_PUSH_CONS:
CosNotifyComm::Structure
dPushConsumer
structured_push_consumer
;¶
case
SEQUENCE_PUSH_CONS:
CosNotifyComm::SequenceP
ushConsumer
sequence_push_consumer; ¶
case
STRUCTURED_PULL_CONS:
CosNotifyComm::Structure
dPullConsumer
structured_pull_consumer
;¶

[1]

Deleted: enum
EndpointSenderType¶
{¶
STRUCTURED_PUSH_SUPL,¶
SEQUENCE_PUSH_SUPL,¶

[2]

Deleted: union
EndpointSender switch
(EndpointSenderType)¶
{¶
case
STRUCTURED_PUSH_SUPL:Cos
NotifyComm::StructuredPu
shSupplier

[3]

Deleted: readonly
attribute
EndpointReceiver
end_point_receiver; ¶
readonly attribute
EndpointSender
end_point_sender;

Deleted: start

Deleted: stop

3.1.1 ExternalEndPoint

`ExternalEndpoints` are abstract entities that represent the sender and the receiver of the data through the NS-JMS bridge. These entities are represented by a data structure `ExternalEndPoint` which specifies :

1. the role of the external end point which can be either a Source or a Sink of data
2. the nature of the external end point, be it a JMS destination or an `EventChannel`
3. the Notification style used by the `ExternalEndPoint`, which can be either a Push or a Pull.
4. the type of message that it handles. This type can be either a JMS message, a structured event or a sequence of the structured events.

Each of the previous points is described by a `ExternalEndPoint` field in the structure.

The following subsections describe briefly those fields.

3.1.1.1 *ExternalEndpointRole*

The `ExternalEndpointRole` is an enumeration that describes the role of the `ExternalEndPoint`. The enumeration is made by the following values: {SOURCE, SINK};

3.1.1.2 *ExternalEndpointConnector*

The `ExternalEndpointConnector` is an IDL union structure that represents exclusively a `JMSDestination` or an `EventChannel`. If the `ExternalEndpointConnector` handles a `JMS_MESSAGE` message type it refers to a `JMSDestination`. If it handles a structured event or a sequence of structured events it refers to an `EventChannel`.

The `JMSDestinaton` is a data structure that includes :

1. The `JMSDestinatonType`, which is an enumeration that can be either a `Queue` or a `Topic`,
2. a string that specifies the destination name,
3. a string that specifies the JMS factory name.

operation maps on the Proxy's `disconnect_` operation on the Notification Service side and on the `javax.jms.Connection.stop` operation on the JMS side.

When the `stop_bridge` operation is successfully executed the bridge state becomes `stopped`. Stopping an inactive bridge raises the `BridgeInactive` exception.

Deleted: stop

Formatted: Bullets and Numbering

3.1.2.3 `get_status`

The `get_status` operation returns the current state of the bridge which can be either in `stopped` or in a `started` mode. The state of the bridge is not necessarily persistent.

Formatted: Bullets and Numbering

3.1.2.4 `destroy`

The `destroy` operation destroys the Bridge instance. The `EndPointReceiver` and `EndPointSender` implementations invoke the `disconnect` operation on the corresponding Notification Service and invoke the `stop` operation on the JMS side. When those operations are successfully executed both of the end points are destroyed.

Formatted: Bullets and Numbering

3.1.3 Bridge Factory Interface

A `BridgeFactory` is responsible for the creation of Bridge objects based on initial parameters. In order to create a bridge, it is necessary to have the identifier of an existing Notification Service event channel and a JMS destination information.

Formatted: Bullets and Numbering

3.1.3.1 `create_bridge`

The `create_bridge` operation creates new instances of NS-JMS bridge. At creation time, the client must specify two external points. One external point must represent an `EventChannel` Instance the other one must represent a JMS destination. Furthermore, the external points information must be consistent with the communication consistency checking table defined in section 2.2. For example, connecting an external point behaving as an `EventChannel`, assuming the Source role and using the `PUSH` `NotifyStyle` with an external point behaving as a `JMSDestination`, assuming the Sink role and using the `PULL` `NotifyStyle` is not consistent.

The `create_bridge` operation raises the `InvalidExternalEndpoints` exception when an inconsistent external endpoint is passed as input parameters.

If no exception is raised, the `create_bridge` operation will return a reference to a new bridge and will assign to this new bridge a unique numeric identifier. This identifier is returned as an output parameter.

Formatted: Bullets and Numbering

3.1.3.2 get_all_bridges

The `get_all_bridges` operation returns a sequence of all of the unique numeric identifiers corresponding to NS-JMS bridge instances, which have been created by the Bridge factory.

Formatted: Bullets and Numbering

3.1.3.3 get_bridge_with_id

The `get_bridge_with_id` operation returns a reference to the Bridge object identified by the supplied bridge id. If the bridge cannot be found, then the `BridgeNotFound` exception is thrown.

3.2 BridgeTransactionMgmt module

The `BridgeTransactionMgmt` module defines a single `TransactoinManagement` interface.

3.2.1 TransactionManagement interface.

The `TransactoinManagement` interface is optionally inherited by the `Bridge` interface. It manages the activation and the disactivation of the routing transaction. This interface is made up by two operations: `enable_transaction` and `disable_transaction`.

```
module BridgeTransactionMgmt
{
    exception UnsupportedTransaction {};
    exception TransactionAlreadyActive {};
    exception TransactionActive {};

    interface TransactionManagement
    {
        void enable_transaction (in unsigned long events, in unsigned long timeout)
```

```
        raises (UnsupportedTransaction, TransactionAlreadyActive);
    void disable_transaction() raises (TransactionActive);
};
}
```

3.2.1.1 *enable_transaction*

The `enable_transaction` operation configures the proxies, namely the `StructuredProxyPushSuppliers`, `StructuredProxyPullConsumer`, `SequenceProxyPushSupplier`, `SequenceProxyPullConsumer` with a transactional client behavior. It allows also those proxies to manage their events queues as OTS recoverable objects.

This operation sets `EnableTransaction` new Notification Service QoS to true. It configures the bridge `EndPoint` objects, namely the `StructuredPushConsumer`, `SequencePushConsumer`, `StructuredPullSupplier`, `SequencePullSupplier` that are connected to the notification service to set their Transactional POA Policies to `Require_shared`.

This operation takes as an input parameter the number of events sent in the scope of the routing transaction. When the bridge convey event sequences, this input parameter denotes the number of event sequences, meaning that the total number of events sent in the routing transaction scope is obtained by multiplying the number of events within a sequence by the value of the input parameter. The number of events, respectively, the number of event sequences sent in the scope of the routing transaction are mapped on the new notification service `TransactionEvents`, respectively, `TransactionEventSequences` `AdminProperties`.

The `enable_transaction` operation takes also as an input parameter the lifetime of the routing transaction. The lifetime of the routing transaction is mapped on the Notification Service `TransactionTimeout` `AdminProperty`.

This operation raises the *UnsupportedTransaction* exception if the Notification Service or the JMS Implementations does not support distributed transactions.

It raises the Exception *TransactionAlreadyActive* if an active transaction is already associated to the bridge object.

3.2.1.2 *disable_transaction*

The `disable_transaction` operation invocation disables the bridge, the notification service and the JMS transactional behaviors. Invoking this operation on an

in progress transaction will raise an exception *TransactionActive* and it will disable the subsequent routing transaction from taking place. The actual execution of this operation will take place as soon as the active transaction finishes.

The implementation of the `disable_transaction` will configure the bridge EndPoint objects, namely the `StructuredPushConsumer`, `SequencePushConsumer`, `StructuredPullSupplier`, `SequencePullSupplier` to set their Transactional POA Policies to `Allows_none`. It will also set the Notification Service `EnableTransaction QoS` to `False`.


```
union EndpointReceiver switch (EndpointReceiverType)
{
    case STRUCTURED_PUSH_CONS: CosNotifyComm::StructuredPushConsumer
structured_push_consumer;
    case SEQUENCE_PUSH_CONS: CosNotifyComm::SequencePushConsumer
sequence_push_consumer;
    case STRUCTURED_PULL_CONS: CosNotifyComm::StructuredPullConsumer
structured_pull_consumer;
    case SEQUENCE_PULL_CONS: CosNotifyComm::SequencePullConsumer
sequence_pull_consumer;
    case MESSAGE_LISTENER: MessageListener msg_listener;
};
```

```
enum EndpointSenderType
{
    STRUCTURED_PUSH_SUPL,
    SEQUENCE_PUSH_SUPL,
    STRUCTURED_PULL_SUPL,
    SEQUENCE_PULL_SUPL,
    MESSAGE_PRODUCER
};
```

```
union EndpointSender switch (EndpointSenderType)
{
    case STRUCTURED_PUSH_SUPL: CosNotifyComm::StructuredPushSupplier
structured_push_supplier;
    case SEQUENCE_PUSH_SUPL: CosNotifyComm::SequencePushSupplier
sequence_push_supplier;
    case STRUCTURED_PULL_SUPL: CosNotifyComm::StructuredPullSupplier
structured_pull_supplier;
    case SEQUENCE_PULL_SUPL: CosNotifyComm::SequencePullSupplier
sequence_pull_supplier;
};
```

EndPointSender interface

The `EndPointSender` interface connects and fits the `ExternalEndPoint` type when the later behaves as data Consumer. The `EndPointSender` behaves as data supplier.

Depending on the values of the `CosEventDomainAdmin::NotificationStyle` and the `MessageType` fields, the `EndPointSender` can be either :

`CosNotifyComm::StructuredPushSupplier` if the

`CosEventDomainAdmin::NotificationStyle` and the `MessageType` of

the `ExternalEndPoint` values are respectively `PUSH` and `STRUCTURED_EVENT`. This means that the `EndPointSenderType` is `STURCTURED_PUSH_SUPPLIER`.
`CosNotifyComm::SequencePushSupplier` if the `CosEventDomainAdmin::NotificationStyle` and the `MessageType` of the `ExternalEndPoint` values are respectively `PUSH` and `SEQUENCE_EVENT`. This means that the `EndPointSenderType` is `SEQUENCE_PUSH_SUPPLIER`.
`CosNotifyComm::StructuredPullSupplier` if the `CosEventDomainAdmin::NotificationStyle` and the `MessageType` of the `ExternalEndPoint` values are respectively `PULL` and `STRUCTURED_EVENT`. This means that the `EndPointSenderType` is `STURCTURED_PULL_SUPPLIER`.
`CosNotifyComm::SequencePullSupplier` if the `CosEventDomainAdmin::NotificationStyle` and the `MessageType` of the `ExternalEndPoint` values are respectively `PULL` and `SEQUENCE_EVENT`. This means that the `EndPointSenderType` is `SEQUENCE_PULL_SUPPLIER`.
When the `CosEventDomainAdmin::NotificationStyle` value is `PUSH` and the `MessageType` is `JMS_MESSAGE` the `EndPointSender` behaves as JMS producer. According to the JMS specification, the JMS producer does not offer any interface. The `EndPointSenderType` value `MESSAGE_PRODUCER` is specified for a future usage if the JCP community decides to extend the JMS `PushSupplier` with a new interface.

EndpointReceiver interface

The `EndPointReceiver` interface connects and fits the `ExternalEndPoint` type when the later behaves as a data supplier. The `EndPointReceiver` behaves as data consumer. Depending on the values of the `CosEventDomainAdmin::NotificationStyle` and the `MessageType` fields, the `EndPointReceiver` can be either a
`CosNotifyComm::StructuredPushConsumer` if the `CosEventDomainAdmin::NotificationStyle` and the `MessageType` of the `ExternalEndPoint` values are respectively `PUSH` and `STRUCTURED_EVENT`. This means that the `EndPointReceiverType` is `STURCTURED_PUSH_CONSUMER`.
`CosNotifyComm::SequencePushSupplier` if the `CosEventDomainAdmin::NotificationStyle` and the `MessageType` of the `ExternalEndPoint` values are respectively `PUSH` and `SEQUENCE_EVENT`. This means that the `EndPointReceiverType` is `SEQUENCE_PUSH_CONSUMER`.
`CosNotifyComm::StructuredPullSupplier` if the `CosEventDomainAdmin::NotificationStyle` and the `MessageType` of the `ExternalEndPoint` values are respectively `PULL` and `STRUCTURED_EVENT`. This means that the `EndPointReceiverType` is `STURCTURED_PULL_CONSUMER`.
`CosNotifyComm::SequencePullSupplier` if the `CosEventDomainAdmin::NotificationStyle` and the `MessageType` of the `ExternalEndPoint` values are respectively `PULL` and `SEQUENCE_EVENT`. This means that the `EndPointConsumerType` is `SEQUENCE_PULL_CONSUMER`.
`MessageListener` JMS standard interface, if the `EndPointConsumerType` is the `CosEventDomainAdmin::NotificationStyle` and the `MessageType`

values of the `ExternalEndPoint` are respectively `PUSH` and `JMS_MESSAGE`.
`EndPointConsumerType` corresponds to `MESSAGE_LISTENER`.
When the `CosEventDomainAdmin::NotificationStyle` value is `PULL` and the `MessageType` is `JMS_MESSAGE` the `EndPointReceiver` behaves as JMS Consumer.
According to the JMS specification, the JMS consumer that pulls messages does not offer any interface.
The `EndPointReceiverType` value's `MESSAGE_CONSUMER` is specified for a future usage if the JCP community decides to extend the JMS `PullConsumer` with a new interface.