Abstract

This document illustrates how the JMS API can be used as transport mechanism supporting the MTOSI application level requirements in terms of communication and operation exchange.

It gives an introduction to the JMS concepts and then it gives the bindings rules and recommendations to use JMS to support MTOSI operations.

Table of Contents

1. Introduction

MTOSI requirements call for a transport independent API. Details of the communication concepts requested at the application level and the different styles supported by MTOSI are presented in [5]. Different Application OSs communicate between each other through the CCV abstraction (Common Communication Vehicule).

A service consumer interacts with a service provider through the invocation of operations to achieve a business goal. The operations involve exchanges of XML messages. The messages are exchanged using a particular communication style. In addition, a communication pattern identifies the sequence and cardinality of messages sent and/or received as well as whom they are sent to or received from.

Two communication styles are promoted by MTOSI, Messaging and RPC, and for each of them there are four communication patterns considered: `SimpleResponse`, `MultipleBatchResponse`, `BulkResponse` and `Notification`.

However, MTOSI phase 1 supports only the Messaging communication style (`MSG`), associated with the JMS transport. The RPC communication style (`RPC`) is best suited for other kind of transport technologies (such as HTTP/S or CORBA). While JMS can also support RPC communication style, MTOSI recommends that JMS transport be used in conjunction with the MSG communication style.

This document illustrates how the JMS API can be used to implement the CCV as transport mechanism supporting the MTOSI application level requirements in terms of communication and operation exchange.

Throughout this document, we present different facets of JMS and show how they fit with the MTOSI requirements.

Companion documents illustrate those concepts using the support of a concrete example, demonstrating the usage of the JMS API using code snippets.

There are many reasons why JMS is perfectly appropriate to support the MTOSI communication requirements. While this document focuses mainly on the technical ones, it is worth mentioning some others:

- The JMS API is available in many middleware products
- It is complete enough to avoid unnecessary proprietary extensions
- It is available in version 1.1 as part of the J2EE 1.4 framework.

The document has the following main sections:

- Section 2 presents the essential concept of the JMS framework for readers not familiar with JMS. Experts in JMS should skip this section.
- Section 3 presents MTOSI rules (denoted as **Rxx**)
  and recommendations (denoted as **Oxx**) concerning how JMS should be used as a transport mechanism for MTOSI.

2. Overview of the JMS Framework

2.1 Essential Abstractions

Java Message Service (JMS) is available as part of the J2EE architecture as the preferred mechanism for communication between enterprise applications. It allows for reliable and flexible exchanges of information between Java applications running on the same system or on independent systems.

JMS simplifies interoperability and eliminates the need for proprietary intercommunication protocols.

JMS fits at the middleware level. Based on the concept of messages, it supports point-to-point and multi-point to multi-point interaction. Compared to the different alternative approaches to inter applications communication, JMS compromises by minimizing the set of concepts that a programmer will have to learn while offering enough expressive power to support sophisticated communication between applications and maximizing portability. It provides different levels of message delivery assurance, and also does not have any dependencies on wire or wireless transport protocol underneath.

JMS is a Java API. However different vendors provide connection libraries for other languages like C/C++/C#/Perl/PHP, facilitating the usage of JMS for non Java applications.

First published in August 1998, the latest version of JMS is Version 1.1, which was released in April 2002. The specification document [1] is available at http://java.sun.com/products/jms/.

While focusing on the essential to offer a simple and powerful API, JMS deliberately does not address the following features (see [1]):

- Load Balancing/Fault Tolerance
  The JMS API does not specify how applications cooperate to appear to be a single, unified service.

- Standard Error/Advisory Notification
  Most messaging products define system messages that provide asynchronous notification of problems or system events to clients. JMS does not attempt to standardize these messages. By following the guidelines defined by JMS, clients can avoid using these messages and thus prevent the portability problems their use introduces.

  This does not mean that JMS cannot be used to support MTOSI notifications. To the contrary, while not imposing any fixed format, JMS allows the application designer to use his own messages to support notification service between MTOSI applications.

- Administration
  JMS does not define an API for administering messaging products. Vendor product offer specific tools for this purpose.

- Security
  JMS does not specify an API for controlling the privacy and integrity of messages.

- Message Type Repository
  JMS does not define a repository for storing message type definitions and it does not define a language for creating message type definitions.

Applications communicate in JMS by exchanging *messages*, representing business actions such as requests, responses, reports or events at the application level. Messages are not exchanged directly

between the applications, but through an intermediate broker, the *JMS Provider*: Applications produce or consume messages to and from a common *JMS Provider*.

This message exchange principle and the mechanisms associated to it are powerful enough to support the different communication styles and patterns promoted in MTOSI.

Here are the essential abstractions of the Java Messaging Architecture (JMS V1.1).

- JMS Provider: a messaging system that implements the JMS interfaces and provides administrative and control features. The JMS provider encapsulates the physical JMS queues and topics.
- JMS Clients: programs or components that produce or consume messages.
    - Message Producers: A message producer is an object that is created by a JMS session and used for sending messages to a JMS Destination. It implements the `MessageProducer` interface.
    - Message Consumers: A message consumer is an object that is created by a JMS session and used for receiving messages sent to a JMS Destination. It implements the `MessageConsumer` interface.
- JMS Messages: objects that communicate information between JMS Clients.
  Administrative objects are preconfigured JMS objects created by an administrator for the use of clients. The two kinds of administered objects are destinations and connection factories.
- JMS Connections: A JMS Connection encapsulates a connection between one client application and a JMS Provider. It typically represents an open TCP/IP socket between a client and a JMS Provider's service daemon. A JMS connection is created programmatically using an administratively defined connection factory.
  A JMS Connection can be used to create one or more JMS Sessions.
- JMS Sessions: A JMS Session is a single-threaded context for producing and consuming messages. You use JMS Sessions to create message producers, message consumers.
- JMS Destinations: A JMS Destination is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the point-to-point messaging domain, JMS Destinations are called queues. In the publish/subscribe messaging domain, JMS Destinations are called topics.
  Most of the time JMS Destination resources are permanent and are created administratively. It is also possible to use temporary JMS Destinations. Temporary JMS Destinations are created programmatically; they receive a system specific name which cannot be published externally through a JNDI interface.
- Message Listener: A message listener is an object that acts as an asynchronous event handler for messages. This object implements the `MessageListener` interface, which contains one method, `onMessage`. In the `onMessage` method, you define the actions to be taken when a message arrives.

2.2 The two messaging paradigms

JMS supports two messaging paradigms: point-to-point and publish/subscribe.

- Point-to-point messaging:

  A point-to-point (P2P) interaction is built around the concept of queues of messages. Each message is addressed to a specific queue, and receiving clients extract messages from the queue(s) established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire.

  - Each message has only one consumer.
  - A sender and a receiver of a message have no timing dependencies. The receiver can fetch a message whether or not it was running when the sender sent it.
  - The receiver acknowledges the successful processing of a message.

  Use P2P messaging when every message you send must be processed successfully by one consumer only.

  Strictly speaking, the JMS specification does not prevent the creation of more than one consumer (within the same application or in different applications) for a given queue. But, in the case more than one consumer is created, then JMS does not define how messages will be distributed between them, knowing that each message will be delivered only once.

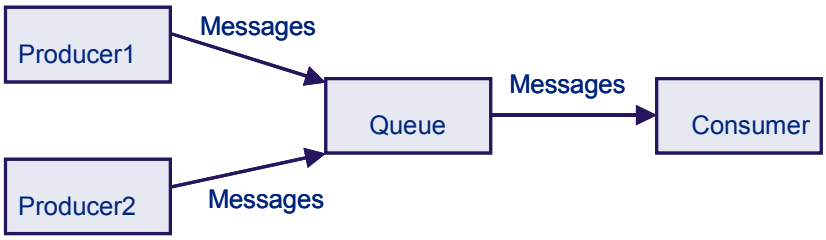

Figure 1. Point to Point Messaging

- Publish/Subscribe messaging:

  In a publish/subscribe (pub/sub) scenario, clients address messages to a topic. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.

  Pub/sub messaging has the following characteristics.

  - Each message can have multiple consumers
  - Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

The JMS API relaxes this timing dependency to some extent by allowing subscribers to create durable subscriptions, which receive messages sent while the subscribers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients.

Use pub/sub messaging when each message can be processed by zero, one, or many consumers.

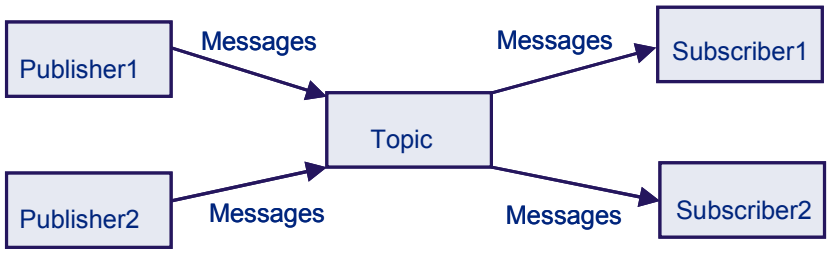

Figure 2. Publish and Subscribe Messaging

2.3 JMS and Naming of administered objects

Typically, JMS Clients look up configured JMS objects using the Java Naming Directory Interface (JNDI) API. JMS administrators use provider-specific facilities for creating and configuring these objects.

This JMS architecture maximizes the portability of clients across different JMS middleware vendors by delegating provider-specific work to the administrator. It also leads to more administrable applications because clients do not need to embed administrative values in their code.

2.4 Delivery Modes

As we have already seen, the communication between producers and consumers is always asynchronous within JMS, meaning that, once a message has been produced, a consumer may consume it without any further dependencies on the producer. This being said, the communication between JMS Clients (consumers or producers) and the JMS Provider handling queues and topics (destinations) can have different styles, which we present below.

- Consumer

  A JMS Client uses a `MessageConsumer` to receive messages from a destination.
  A `MessageConsumer` is created by passing a queue or topic to a session's `createConsumer` method.
  A JMS Client may either synchronously receive consumer's messages or have the JMS Provider asynchronously delivering them as they arrive.

  - Synchronous delivery
    A JMS Client can request the next message from a `MessageConsumer` using one of its `receive` methods. There are several variations of `receive` that allow a client to poll or wait for the next message.

```
// blocks forever until a message is received
Message m = consumer.receive();

// blocks 1000 ms waiting for the reception of a message
Message m = consumer.receive(1000);
```

- Asynchronous delivery
  A JMS Client can register an object that implements the JMS `MessageListener` interface with a `MessageConsumer`. As messages arrive for the consumer, the provider delivers them by calling the listener's `onMessage` method.

- Producer

  A JMS Client uses a `MessageProducer` to send messages to a destination. A `MessageProducer` is created by passing a queue or topic to a session's `createProducer` method.
  A producer can specify a default delivery mode, priority, and time-to-live for messages it sends to a destination. It can also specify delivery mode, priority, and time-to-live per message. Those default values are left to the responsibility of the implementer.

  There are two delivery modes that a producer can use: PERSISTENT and NON_PERSISTENT:

  A JMS Provider must deliver a NON_PERSISTENT message *at-most-once.* This means that it may lose the message, but it must not deliver it twice.

  A JMS Provider must deliver a PERSISTENT message *once-and-only-once*. This means a JMS Provider failure must not cause it to be lost, and it must not deliver it twice.

  PERSISTENT (once-and-only-once) and NON_PERSISTENT (at-most-once) message delivery are a way for a JMS Client to select between delivery techniques that may lose a messages if a JMS provider dies and those which take extra effort to insure that messages can survive such a failure.

  A JMS Client producer can specify a time-to-live value in milliseconds for each message it sends.

2.5 Structure of a JMS Message

A JMS Message is composed of three parts:

- Header - All messages support the same set of header fields. Header fields contain values used by both JMS Clients and by the JMS Provider to identify and route messages.

- Properties - In addition to the standard header fields, messages provide a built-in facility for adding optional header fields to a message.

  - Application-specific properties - This provides a mechanism for adding application-specific header fields to a message.
  - Standard properties - JMS defines some standard properties that are, in effect, optional header fields.

- Provider-specific properties - Integrating a JMS Client with a JMS Provider native client may require the use of provider-specific properties. JMS defines a naming convention for these.

- Body - JMS defines several types of message body which cover the majority of messaging styles currently in use. There are five different message types defined in JMS:
  - StreamMessage - a message whose body contains a stream of Java primitive values. It is filled and read sequentially.
  - MapMessage - a message whose body contains a set of name-value pairs where names are *Strings* and values are Java primitive types.
  - TextMessage - a message whose body contains a *java.lang.String*. This type is perfectly appropriate for content message using XML.
  - ObjectMessage - a message that contains a Serializable Java object.
  - BytesMessage - a message that contains a stream of uninterpreted bytes.

The JMS header fields are shown in the table below:

Header Field	Type	Description
`JMSMessageID`	String	Unique ID for each message sent by the JMS provider. Set by JMS Provider during send process, it can be modified by the *receiving* application.
`JMSDestination`	Destination	Message destination. Set by the sender application on a per message basis.
`JMSDeliveryMode`	Int	Delivery mode (persistent or nonpersistent). Set by the sender application, as a parameter of the `MessageProducer` resource. Can be changed at any time. Delivery mode is set to `PERSISTENT` by default
`JMSTimestamp`	Long	Time message is handed to provider for sending. Set by JMS Provider during send process, it can be modified by the *receiving* application.
`JMSExpiration`	Long	Expiration time. Set by JMS Provider during send process, it can be modified by the *receiving* application. The `setTimeToLive` method can also be used by the sender application as a parameter of the `MessageProducer` resource (sets the default length of time in milliseconds from its dispatch time that a produced message should be retained by the message system). Can be changed at any time. Time to live is set to zero by default.
`JMSPriority`	Int	Message priority (10 possible values). Set by the sender application as a parameter of the `MessageProducer` resource. Can be changed at any time. Priority is set to 4 by default.
`JMSCorrelationID`	String	Links one message with another. A typical use is to link a response message with its request message.

Header Field	Type	Description
		Set by the sender application on a per message basis.
`JMSReplyTo`	Destination	Destination where a reply to the message should be sent. Set by the sender application on a per message basis. The `JMSReplyTo` header field contains the destination where a reply to the current message should be sent. *If it is null, no reply is expected* (typical usage case will be for notifications). Messages with a `JMSReplyTo` value typically expect a response. A response is optional; it is up to the receiving application to decide.
`JMSType`	String	Message type identifier. Set by the sender application on a per message basis.
`JMSRedelivered`	Boolean	Message was probably delivered earlier, but client did not acknowledge

Table 1. The JMS header fields

2.6 Message Selector

JMS offers a mechanism to support the filtering of messages, through the concept of JMS Message Selector.

A JMS Message Selector is a conditional expression, subset of SQL92, which has a string representation, to be used by a JMS Client when it creates a `MessageConsumer` associated to a given JMS destination. Doing so, a JMS Client delegates to the JMS Provider the selection of the messages it is interested to receive. Messages not matching the condition expression specified in the Message Selector will not be forwarded by the JMS Provider to the corresponding JMS client.

Message Selectors cannot reference message body values, meaning that Message Selectors may only reference JMS headers and properties values.

Message Selector expressions may contain:

- Literals: string, numeric and boolean
- Identifiers, representing JMS header fields or JMS property fields.

- Operators: logical operators, comparison operators and arithmetic operators

Here is an example of a string representation of a Message Selector:

```
MTOSI_EventType = 'OBJECTCREATION'    OR    MTOSI_EventType = 'OBJECTDELETION'
```

2.7 Procedures for Creating and Using the JMS Interface

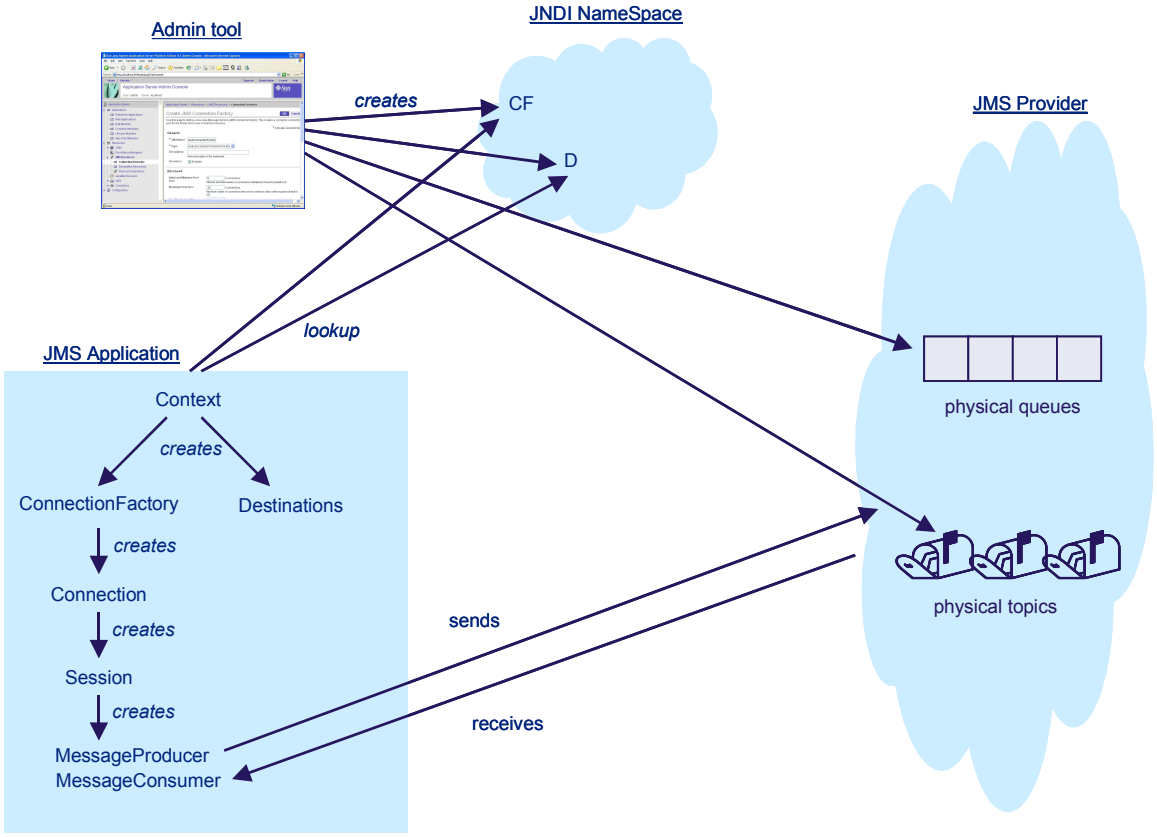

Figure 3. The JMS API Programming Model

A typical JMS system (e.g. a set of OS applications supporting the MTOSI) goes through the following steps to begin producing and consuming messages:

- Set-up the Administrative Environment

  Create the `ConnectionFactory` and the permanent JMS Destination Resources

  Connection factories and permanent destination resources need to be registered in a directory service to be further looked up using the JNDI API. They are many different standards and implementations of directory services (e.g. DNS, LDAP, NDS, NIS to name a few).

How to actually create those JMS entities depends on the JMS environment used. For instance, Sun Java™ System Application Server Platform offers a specific tool with a graphic interface and a set of CLI commands.

- Look up a `ConnectionFactory` and `Destinations` through JNDI

  What is common to all popular naming services is the support of the JNDI interface. It is through this interface that MTOSI applications look up for the JMS resources they need, using their published names.

  ```
  Context ctx = new InitialContext();
  ```

  (Assuming that the client is linked with the JNDI libraries specific of a JNDI vendor and also has the proper configuration to access the JNDI server.)

  ```
  ConnectionFactory c1 = (ConnectionFactory)
        ctx.lookup("jms/QueueConnectionFactory");


  Destination myDest = (Destination) ctx.lookup("jms/RtoNMSQA");
  ```

  In the example above, note the usage of the type `Destination` from the JMS common interface, which allows handling queues and topics in the same abstraction.

- Use the `ConnectionFactory` to create a JMS Connection.

  A connection factory is used by an application to create a connection to the JMS Provider.

  ```
  Connection connection = c1.createConnection();
  ```

- Use the connection to create one or more sessions.

  A connection can support one or many sessions, which can be transacted or not.

  ```
  Session session = connection.createSession
                          (false,
                           Session.AUTO_ACKNOWLEDGE);
  ```

  The first argument specifies whether the session is transacted or not, and the second argument specifies whether the application requests the session to automatically acknowledge the messages upon reception.

  Using transacted sessions allows to group a series of send and receive messages into an atomic unit of work. Transactions are rolled back if they fail at any time.

- Use a JMS Session and a JMS Destination to create the required `MessageProducers` and `MessageConsumers`.

  The message producers and consumers are created associated within a given session, using appropriate `Destination` objects.

  ```
  MessageProducer producer = session.createProducer(myDest1);

  MessageConsumer consumer = session.createConsumer(myDest2);
  ```

- Start the JMS Connection

  Starting the connection allows to start the message delivery mechanism. It is not possible to send or to receive messages before having requested the start of the connection.
  This step may be done after or before having created the producers and consumers.

  ```
  connection.start ();
  ```

- Send and receive messages

  At this stage the dialogue between applications through the JMS Provider can start.

- Stop/Close the session. Clean-up

As mentioned already, in MTOSI phase 1, the way destinations will be named and organized for registration in a directory to be looked up at a later stage by JMS Client applications through the JNDI interface has not been finalized. It will be further investigated as part of the phase 2. The names used in the remaining parts of this document are used just as supporting examples, without precluding any syntax or style that will be proposed in MTOSI phase 2.

3 MTOSI Rules and Recommendations concerning how to use JMS

This section gives the rules and recommendations on how to use JMS to support MTOSI message exchange:

- Rules are mandatory: noted **R x**
- Recommendations are optional: noted **O x**.

MTOSI specifies the definitions of service operations and the binding to SOAP using WSDL format (see [7]). In turn, the SOAP binding makes references to the XML specifications. Those WSDL definitions are normative.

However, due to WSDL v1.1 limitations with respect to the definition of standard extensions for the SOAP bindings with JMS, it is necessary to provide all required specifications about these binding definitions.

This is the purpose of this section which is organized into the following subsections:

- “General Aspects”: about general JMS configuration definitions and MTOSI settings of JMS parameters.
- “Mapping SOAP Addresses to JMS Destinations”
- “Mapping WSDL Operations to JMS Message Exchanges”
- “Mapping SOAP Messages to JMS Messages”

3.1 General Aspects

R 1. MTOSI mandates that JMS version 1.1 (or later version when available) be used. The current version is 1.1.

O 1. MTOSI recommends the usage of JMS Text message only, since the messages exchanged will contain XML data only.

O 2. Both kinds of domain specific JMS Destinations (JMS topics or JMS queues) may be used for any of the MTOSI communication patterns. However, MTOSI recommends to use JMS Topics for the MTOSI Notification pattern and JMS Queues for the others.

O 3. MTOSI recommends the usage of the JNDI interface to access the administratively created JMS Destinations. MTOSI, in phase 1, does not suggest or recommend any style or technology. Note JNDI will hide the implementation technology by offering a common API to the clients.

O 4. MTOSI recommends the usage of the version domain-independent common API as available in JMS 1.1.

O 5. MTOSI does not mandate any style for consumer delivery: consumers may use synchronous (receive) or asynchronous delivery (listener) styles.

O 6. MTOSI does not mandate any style for producer delivery. Both PERSISTENT and NON_PERSISTENT styles can be used.
However, the default style will be NON_PERSISTENT:

DEFAULT_DELIVERY_MODE = NON_PERSISTENT

O 7. Based on MTNM agreements concerning usage of the OMG Notification Service, MTOSI recommends the following default values:

DEFAULT_TIME_TO_LIVE =
24 hours for life cycle events messages
30 minutes for any other message

DEFAULT_PRIORITY = 4

3.2 Mapping SOAP Addresses to JMS Destinations

Planning for the deployment of an OS application that supports MTOSI shall start with the following two important steps:

- The MTOSI profile of the OS application (Refer to SD2-1)
- Identify the access points where the service operations supported by this application will be available.

Using JSM terminology those access points are materialized as administratively defined JMS Destinations (JMS Queues or JMS Topics). A JMS Destination is known through its name.
Using WSDL terminology, a service is accessible through the port of the server that supports it. Since MTOSI uses SOAP, this port is known through its SOAP Address.

When the designer has decided on the name of the JMS Destinations, he needs to report it in the SOAP Address of the WSDL specification for this application.

3.3 Mapping WSDL Operations to JMS Message Exchanges

The four MTOSI services are specified in terms of WSDL.

Manager Interfaces and the corresponding WSDL Ports as summarized in the tables below:

- File *DiscoveryServiceJMS.wsdl:*

Discovery Service	
Client (RequestorOS)	**Server (SupplierOS)**
NA	Discovery

- File *ConfigurationServiceJMS.wsdl*:

Configuration Service	
Client (RequestorOS)	**Server (SupplierOS)**
NA	EquipmentInventoryMgr

	InventoryRetrieval
	ManagedElementMgr
	MultiLayerSubnetworkMgr
	OperationsSystemMgr
	ProtectionMgr
	TransmissionDescriptorMgr

- File *FaultServiceJMS.wsdl:*

Fault Service	
Client (RequestorOS)	**Server (SupplierOS)**
NA	AlarmRetrieval

- File *NotificationServiceJMS.wsdl:*

  For this service, the JMS Provider actually plays the role of the Notification Broker, and as such it exposes an interface to collect the notifications sent by PublisherOS applications (JMS Clients).

  SubscriberOS applications (JMS Clients), in turn, expose an interface to collect the notifications sent by the JMS Provider.

Notification Service		
Client (SubscriberOS)	**JMS Provider**	**Server (PublisherOS)**
NotificationConsumerInterface	NotificationBrokerInterface	NA

MTOSI supports three different communication patterns: *SimpleResponse*, *MultipleBatchResponse and Notification.*

WSDL does not propose a separate operation type pattern to differentiate the case of a single response from the case of multiple responses. After considerations of the different existing patterns available in WSDL V1.1, V1.2 and those ones in preparation in V2, and different discussions with experts from W3C, the following decisions have been taken:

- the WSDL Request-Response MEP is used to specify operations that can use *SimpleResponse*, *MultipleBatchResponse* MTOSI patterns.
- the WSDL One-Way MEP is used to specify the “Notify” operation.

O 8. As a consequence, all operations, to the exception of the “Notify” operation, are specified using the WSDL Request-Response MEP. The decision to choose the *SimpleResponse* or *MultipleBatchResponse* for a given operation can be taken at run time. Two different invocations of the same operation may use the *SimpleResponse* or the *MultipleBatchResponse* patterns *ad libidum*. For instance, the “`getInventory`” operation may use either pattern depending on the value of the filter passed as parameter which will drive the size of the result.

R 2. A temporary destination is system specific and can be consumed only by the JMS Connection that created it. There is no naming mechanism proposed by the JMS specifications for temporary destinations, preventing the registration of temporary destination into a naming service.

The JMS Destinations must respect the following rules:

- All interfaces must be supported by administratively defined JMS Destinations
- Temporary JMS Destinations may be used only on the Client side. However, for the Client side also, MTOSI recommends the use of administratively defined JMS Destinations.

O 9. A JMS Destination may be used for all or any combinations of Server or Client port types bundled in the same MTOSI service.
In case a given OS application supports more than one WSDL Service interface, it is a designer decision to use one specific JMS Destination for each interface supported or to use only one JMS Destination to support all the Service interfaces, or any combination in between.
The decision is taken on a per application basis. It means that different decisions may be considered at the Client side and at the Server side.

For instance, assuming we have the following applications:

- a SupplierOS supporting the `EquipmentInventoryMgr` and the `InventoryRetrieval` interfaces
- and RequestorOS willing to invoke operations throughout those interfaces, and using administratively defined JMS Destinations

Then the figures below show different valid configurations:

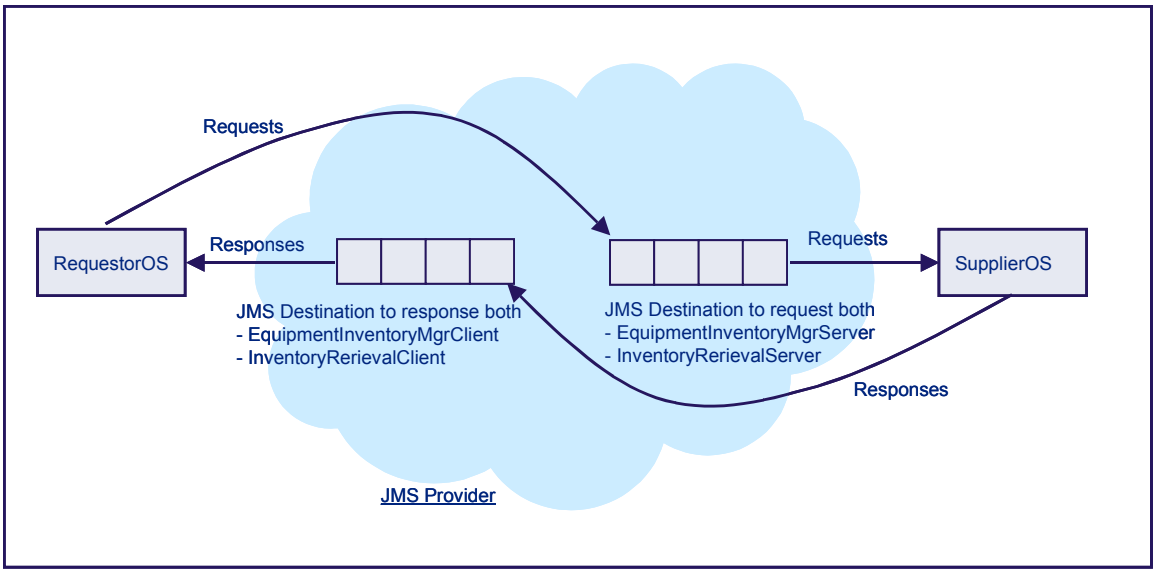

Figure 4. SupplierOS and RequestorOS have one single JMSDestination each

Figure 4 shows a configuration where:

- the SupplierOS exposes both
  the `EquipmentInventoryMgr`
  and the `InventoryRetrieval` interfaces
  at the same JMSDestination
- the RequestorOS receives responses from both interfaces
  at the same JMSDestination

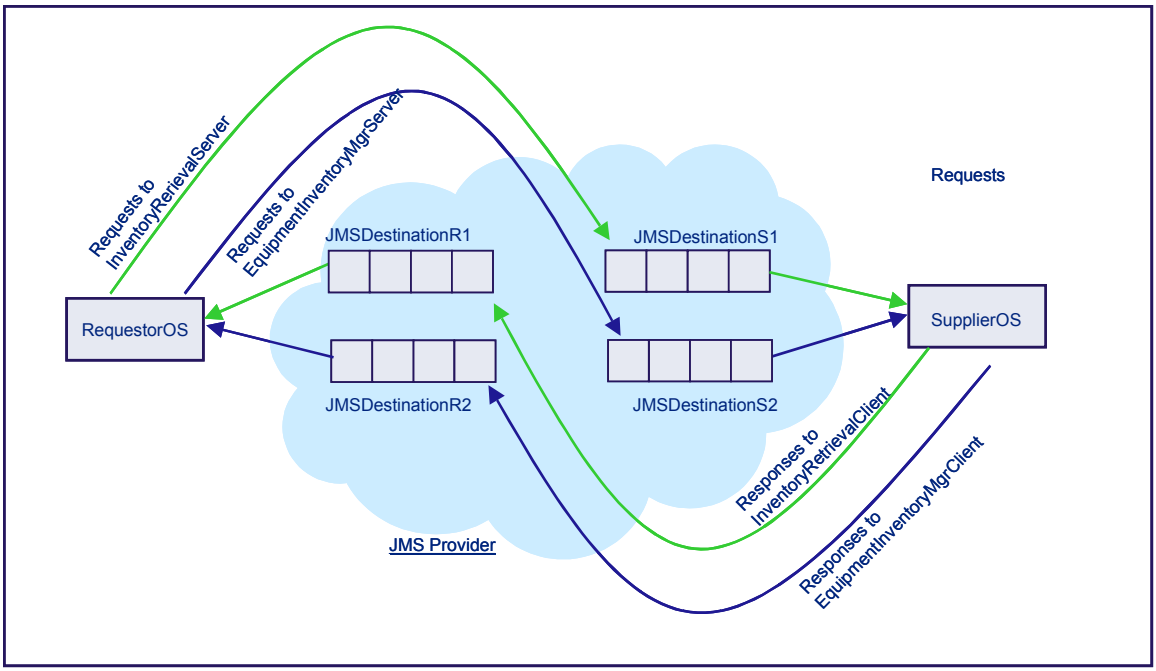

Figure 5. SupplierOS and RequestorOS have two distinct JMSDestinations each

Figure 5 shows a configuration where:

- the SupplierOS uses
  - JMSDestinationS1 to expose the `InventoryRetrieval` interface
  - JMSDestinationS2 to expose the `EquipmentInventoryMgr` interface
- the RequestorOS uses
  - JMSDestinationR1 to receive responses from the `InventoryRetrieval` interface
  - JMSDestinationR2 to receive responses from the `EquipmentInventoryMgr` interface.

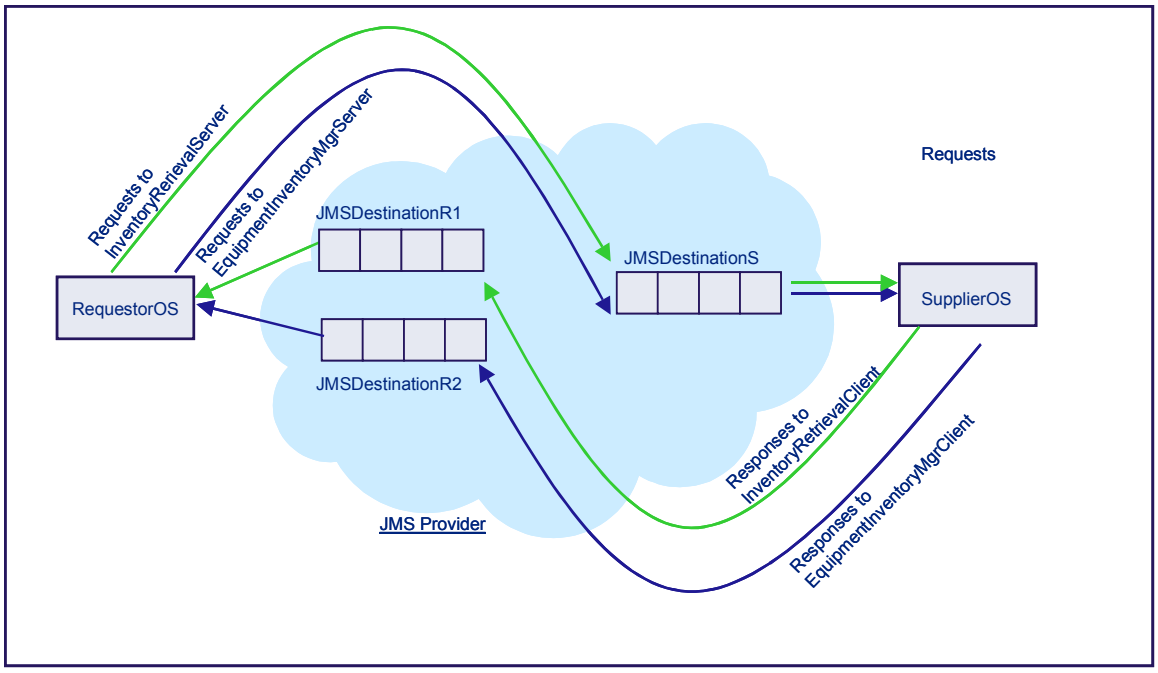

Figure 6. SupplierOS has one JMSDestination and RequestorOS has two

Figure 6 shows a configuration where:

- the SupplierOS exposes both the `EquipmentInventoryMgr` and the `InventoryRetrieval` interfaces at the same JMSDestination
- the RequestorOS uses
  - JMSDestinationR1 to receive responses from the `InventoryRetrieval` interface
  - JMSDestinationR2 to receive responses from the `EquipmentInventoryMgr` interface.

The following sub-sections highlight some specific aspects of the JMS bindings for each of the three MTOSI communication patterns.

3.3.1 *SimpleResponse* MTOSI Communication Pattern

This pattern is used in a point to point request-reply dialogue, between a RequestorOS and a SupplierOS, when a single response is requested.
It is mapped to the WSDL Request-Response WSDL operation type.

The typical configuration of JMS destinations is illustrated on Figure 7 below.

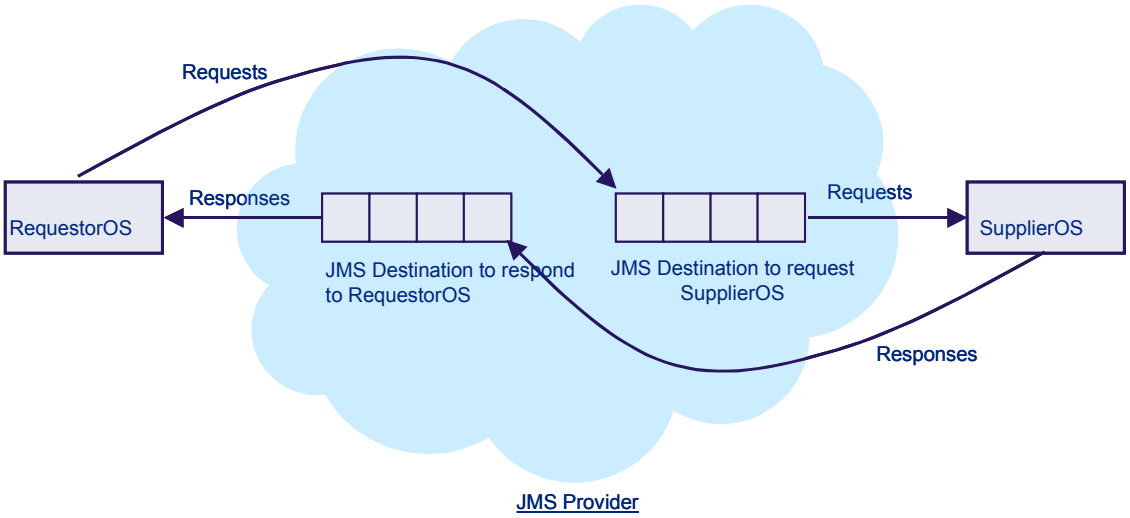

Figure 7. An example of JMS destinations configuration for the *SimpleResponse* communication pattern

The JMS Destination to collect requests at the SupplierOS side must be permanent (name published and accessible through JNDI interface).

The JMS Destination to collect responses at the RequestorOS side may be either permanent or temporary. This is a deployment issue.

In the case when the RequestorOS wants to receive failure messages separated from the normal responses, a second specific destination needs to be created for that purpose.

- Control flow on the SupplierOS side:

  After processing incoming requests, the SupplierOS sends responses to the JMS Destination identified by the JMSReplyTo JMS header field present in the request.

- Control flow on the RequestorOS side:

  The application level conceptual control flow is presented on Figure 8.

R 3. A RequestorOS may send several requests at the same time to the same or different SupplierOS. The MTOSI header field `correlationID` must be provided in the request; the SupplierOS copies it back into the response.
This field is NOT mapped at the JMS level (see section 3.5).
It is this field that will allow correlating each response with its corresponding request, at reception.
This approach is totally thread safe.

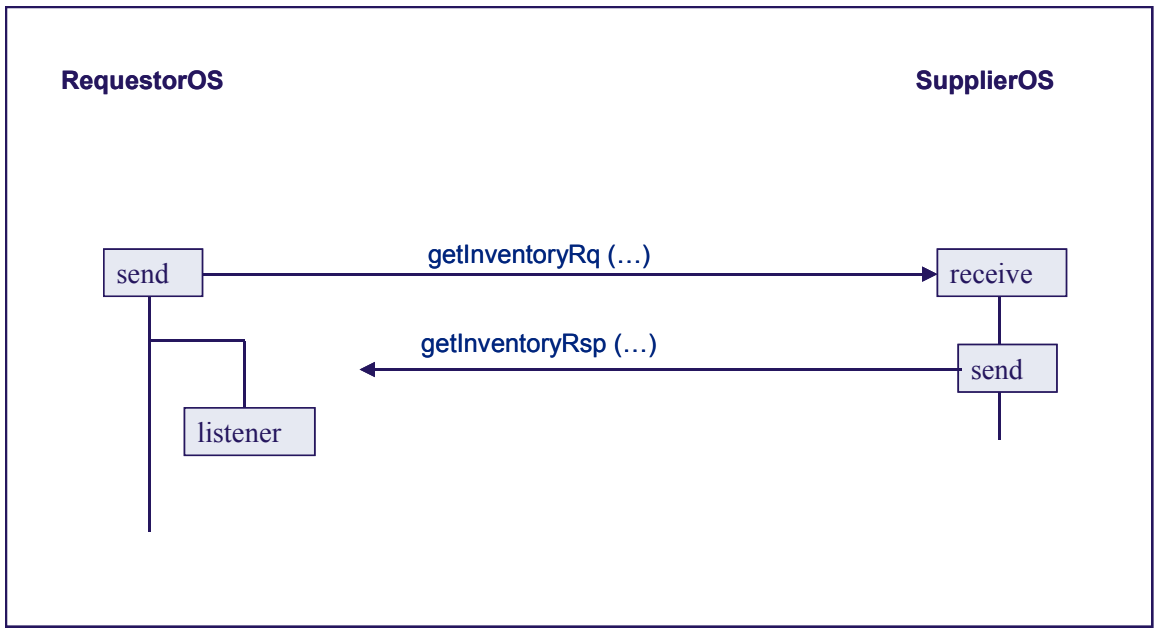

Figure 8. Application level control flow, *SimpleResponse* communication pattern, *MSG* communication style

3.3.2 *MultipleBatchResponse* MTOSI Communication Pattern

This pattern is used in a point to point request-reply dialogue, between a RequestorOS and a SupplierOS, when more than one response is expected.
Like the *SimpleResponse* pattern, it is mapped to the WSDL Request-Response WSDL operation type.

R 4. When the `MultipleBatchResponse` communication pattern is requested, the `requestedBatchSize` parameter must be set in the MTOSI header part of the request. This parameter represents the maximum number of managed object instances, as specified in the filter, expected in partial responses (e.g. maximum number of MEs).

R 5. The responses must contain the `batchSequenceNumber` and `batchSequenceEndOfReply` MTOSI header fields.
These fields must be mapped into the `MTOSI_batchSequenceNumber` and `MTOSI_batchSequenceEndOfReply` JMS property fields (as shown in Table 2).

R 6. The vale of the MTOSI header field `correlationID` (copied from the original request) is the same in all the partial responses. It must be provided.
This field is NOT mapped at the JMS level (see section 3.5).

The typical configuration of JMS destinations is the same as show on Figure 7.

Upon reception of the request, the SupplierOS considers the `requestedBatchSize` parameter to prepare the responses.

Partial responses are constructed and sent to the `JMSReplyTo` destination as soon as they are available, without waiting for any further solicitation from the RequestorOS. The sequencing of the partial responses may not be guaranteed by the JMS provider and the RequestorOS is responsible to ensure that all messages have been received and put in the correct sequence.

There is no information context registered at the SupplierOS side.

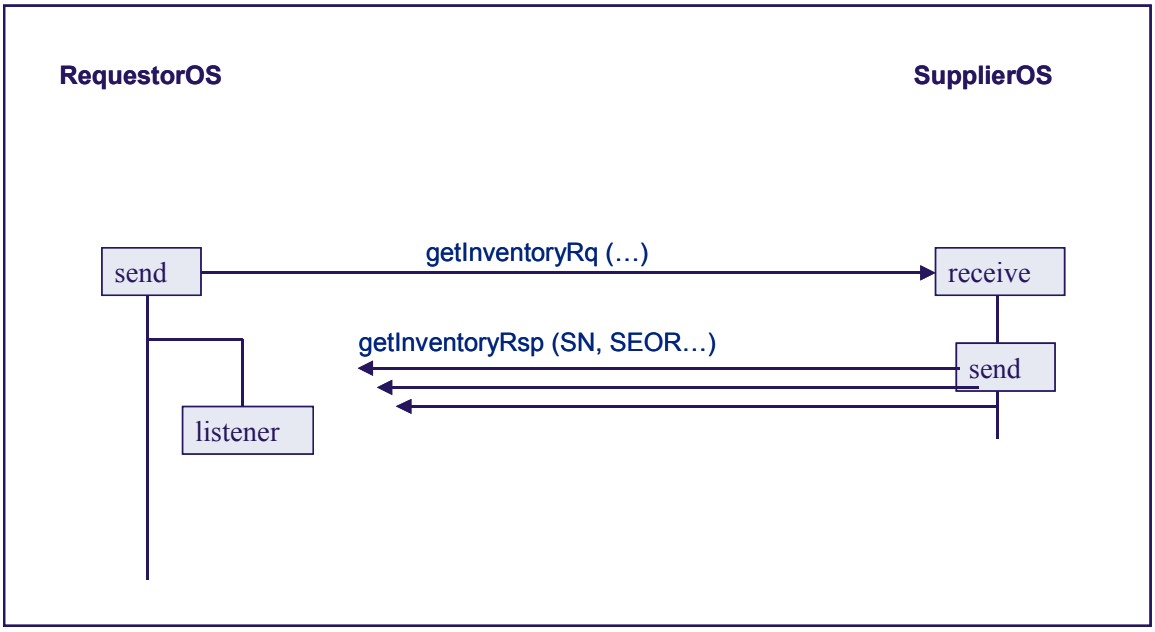

Figure 9 Application level control flow, *MultipleResponse* communication pattern, *MSG* communication style

3.3.3 *Notification* MTOSI Communication Pattern

This pattern is designed to disseminate MTOSI notifications to a set of recipients possibly greater than one.

O 10. In this current phase, MTOSI has identified three different categories of notifications, known as MTOSI Notification Topics:

- `Inventory` MTOSI Notification Topic
- `Fault` MTOSI Notification Topic
- `Protection` MTOSI Notification Topic

MTOSI recommends that when an application wants to publish notifications, it uses a specific permanent administratively declared JMS topic for each category of MTOSI Notification Topics it wants to publish.
For example, if an application wants to publish notifications from the Inventory MTOSI Notification Topic and the Fault MTOSI Notification Topic, it uses two independent specific JMS topics for this purpose. Each of them will be used for a specific (unique) category of MTOSI Notification Topic to the exclusion of any other.

O 11. It is perfectly acceptable that several independent OS applications share the same JMS topic to publish notifications belonging to a common MTOSI Notification Topic.
This is a deployment decision.

As specified in the *NotificationServiceJMS.wsdl* file, the MTOSI Notification service specifies the NotificationBroker binding definition with the following operations:

Subscribe	as a Request-Response WSDL operation type
Unsubscribe	as a Request-Response WSDL operation type
Notify	as a One-Way WSDL operation type

We use the JMS Provider with the role of a Notification Broker. However, there is no direct binding at the JMS level for the Subscribe and Unsubscribe operations. Strictly speaking JSM Clients willing either to publish notifications or to subscribe to the reception of notifications do not exchange any Subscribe message with the JMS Provider.

R 7. Instead, an application willing to publish notifications simply creates, at the API level, a `MessageProducer` associated with a specific JMS Destination, previously created administratively. This JMS Destination is typically looked up from its published name, using the JNDI API.

R 8. Instead, an application willing to receive notifications simply creates, at the API level, a `MessageConsumer` associated with a specific JMS Destination, previously created administratively. This JMS Destination is typically looked up from its published name, using the JNDI API.

O 12. While it is more “natural” to use JMS topics for the MTOSI `Notification` communication pattern, MTOSI does not mandate their usage.
JMS queues and JMS topics have a different scalability and performance behaviour and should be used in an architecture topology (MTOSI domain) according to the deployment requirements.
For instance if we know in advance that an MTOSI Notification Topic will be dedicated to convey information from an NMS to a single OS it may be appropriate to use a queue.
On the other hand, in a more generic loosely coupled architecture, an Inventory OS may decide to externalise the `ObjectCreation` event through the use of a JMS topic in order to efficiently reach all the federated OSS interested on the state changes. It is essential to document this deployment decision to allow connectivity and interoperability in the context of a MTOSI domain.

R 9. When publishing an MTOSI notification, a publisher may map filterable parameters into JMS application-specific property fields, as shown in Table 4.
The decision about which parameters to map is left to the implementer. However MTOSI mandates that the selection of which parameters to map is decided at configuration time and is uniform for all publishers in the same system. In other words, several independent publishers publishing the same type of MTOSI notifications (e.g. `ObjectCreation`) must use the same set of parameters mapped into JMS property fields.

R 10. Subscriber applications will be able to make use of the JMS Message Selector mechanism supported by JMS Providers, referencing the filterable parameters mapped into JMS property fields.

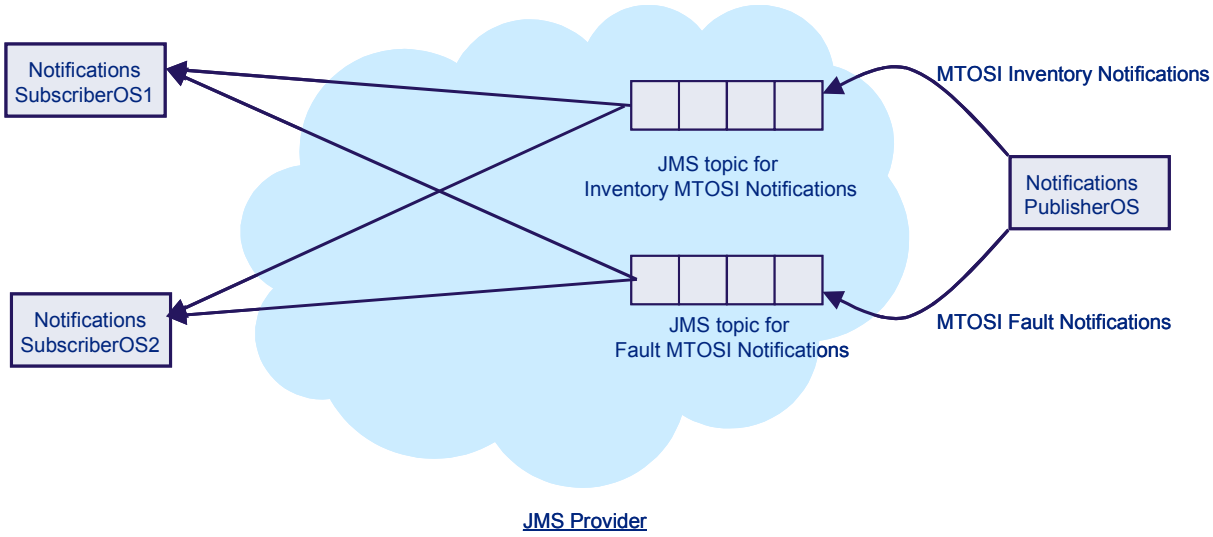

Figure 10. An example of JMS topics for the Notification MTOSI pattern

Figure 10 shows a configuration where two JMS topics are used to disseminate separately

- Inventory MTOSI notifications (event notification types: `ObjectCreation`, `ObjectDeletion`, `AttributeValueChange`, and `ObjectDiscovery`)
- and Fault MTOSI notifications (event notification type: `StateChange`, and `AlarmInformation`).

3.4 *Mapping SOAP Messages to JMS Messages: General Case*

The MTOSI layer architecture stipulates the use of the standard W3C SOAP envelope in the JMS Message body. The SOAP message body contains the MTOSI message.

In MTOSI, the dialogue mechanism between the communicating OS applications is controlled by a set of parameters that can be integrated either in the header or in the body part of the MTOSI XML message (see [7]). Some of those parameters can be used to control the communication at the underlying transport level (in our case JMS). For this reason, they are passed to the JMS layer and are mapped into JMS header fields or JMS properties fields, as shown on Figure 11. The process is as follows:

- In a producer JMS Client, the application layer encapsulates the MTOSI XML body into a SOAP body and the MTOSI XML header into a SOAP header, and then wraps them into a SOAP envelope (upper row in the figure). This SOAP envelope is opaque to the JMS layer.
- When the application layer invokes the JMS transport layer it passes it the SOAP envelope and it also transmits the values of the relevant MTOSI XML header fields (middle row).
- The JMS layer uses those data to prepare the JMS Message (lower row).:
  - the relevant MTOSI XML header fields are used as input to the JMS header fields or JMS properties.
  - the SOAP envelope will constitute the JMS Message body.

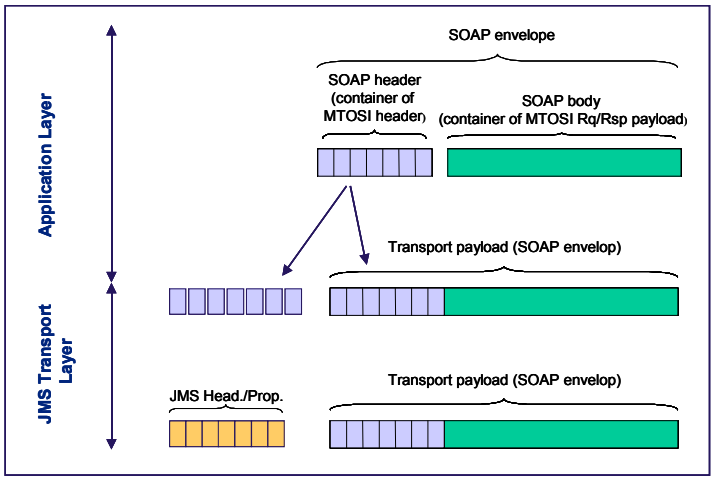

Figure 11. JMS header fields and JMS properties

The two snippets in Figure 12 and Figure 13 show an example of a SOAP envelope containing a MTOSI header and MTOSI body corresponding to a “getInventory” request.

```
<?xml version="1.0" ?>
 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
     <q: MTOSIHDR.v1.Header xmlns:q="mtosi_namespace_URI"
       env:role=http://www.w3.org/2003/05/soap-
envelope/role/ultimateReceiver
    env:mustUnderstand="true">

   <Header xmlns="tmf854"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation="..." version="1.0">
         <activityName>BulkInventoryRetrieval<activityName/>
         <msgName>getInventory</msgName>
         <msgType>REQUEST</msgType>
         <payloadVersion>1.0</payloadVersion>
         <destinationURI>jms/RtoNMSQA</destinationURI>
         <senderURI>jms/RtoInventoryOSQA</senderURI>
         <failureReplytoURI>jms/RtoInventoryOSQA</failureReplytoURI>
         <correlationId>1</correlationId>
         <priority>4</priority>
         <communicationPattern>SimpleResponse</communicationPattern>
         <communicationStyle>MSG</communicationStyle>
   </Header>

     </q: MTOSIHDR.v1.Header>
   </env:Header>
```

Figure 12. MTOSI header of a "getInventory" request (MSG communication style)

```
 <env:Body >

<getInventory xmlns="tmf854"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema- instance"
 xsi:schemaLocation="..." version="1.0">

 <compressionType>NO_COMPRESSION</compressionType>
 <packingType>NO_PACKING</packingType>
 <filter version="1.0">
       <baseInstance version="1.0">
             <mdNm>N1/XdrEMS/Server1</mdNm>
       </baseInstance>
       <includedObjectType>
             <objectType>ME</objectType>
             <granularity>FULL</granularity>
       </includedObjectType>
       <includedObjectType>
             <objectType>EH</objectType>
             <granularity>FULL</granularity>
       </includedObjectType>
       <includedObjectType>
             <objectType>EQ</objectType>
             <granularity>FULL</granularity>
       </includedObjectType>
       <includedObjectType>
             <objectType>CC</objectType>
             <granularity>FULL</granularity>
       </includedObjectType>
 </filter>
   </getInventory>

  </env:Body>
</env:Envelope>
```

Figure 13. MTOSI body of a "getInventory" request

3.5 MTOSI Header Fields Mapping

This section presents the normative rules to follow when mapping MTOSI XML header fields to JMS header fields or JMS properties:

R 11. The candidate MTOSI XML header fields that may be mapped to JMS header fields or JMS application specific properties are shown in Table 2.

R 12. Some MTOSI XML header fields may not always be present in the MTOSI message (e.g. fields “priority”, “replyToURI”). Indeed, in this case, no mapping is done.

R 13. When a field is present in the MTOSI XML header, MTOSI mandates the mapping only in some cases (see “M” in the table). In other cases, the decision to create or not create the corresponding JMS header field or JMS application specific property is left to the implementer.

R 14. The names of the JMS header fields and JMS application specific properties shown in the table are prescriptive.

R 15. The value type of the JMS header fields and JMS application specific properties, shown in the table, is a String in most cases, with the following exceptions:

- *JMSReplyTo* conveys Destination Java class values
- *JMSPriority, MTOSI_requestedBatchSize* and *MTOSI_batchSequenceNumber* convey Integer values
- *MTOSI_batchSequenceEndOfReply* conveys Boolean values

R 16. *senderURI* and *replyToURI*:

- when the *senderURI* MTOSI header field is present but not the *replyToURI* one, then the *senderURI* MTOSI header field is mapped to the *JMSReplyTo* header field, and the *MTOSI_senderURI* is not used.

- when the *senderURI* and the *replyToURI* MTOSI header fields are both present, then the *replyToURI* MTOSI header field is mapped to the *JMSReplyTo* JMS header field, and the *senderURI* MTOSI header field is mapped to *MTOSI_senderURI* JMS property.

The value conveyed in the *JMSReplyTo* JMS header field is the JMS Destination corresponding to the *senderURI* or *replyToURI* and not the String value of those URI.

This way, the *JMSReplyTo* JMS header field always contains the JMS destination where the response should be sent.

R 17. *JMSPriority*:
When the *priority* MTOSI header field is not present, the value taken by this field is left to the implementer.

R 18. *languageCode* and *countryCode*:
These two MTOSI header fields are not shown in the table. They should never be mapped to a JMS header field or JMS property.

O 13. *correlationID:*
It is not recommended to map the *correlationID* MTOSI header field at the JMS level. The reason is that it is not needed:

- the *correlationID* MTOSI header field present in a SOAP request is reused unchanged by the supplier when building the SOAP response message, allowing the SOAP requestor to correlate the received response to the SOAP request sent earlier.
- the responsibility of the JMS middleware is to ensure that JMS response messages are transmitted to the proper *JMSReplyTo* Destination. The need for correlating JMS response messages with JMS requests message is handled by the JMS middleware, by use of the *JMSCorrelationID*, which can be used as it is recommended in [1]:
  - after having sent a request, the requestor, at the JMS level, collects the corresponding *messageID* as it has been assigned by the JMS middleware during the send operation
  - the supplier, at the JMS level, collects this *messageID* from the incoming JMS message, and copies it into the *JMSCorrelationID* of the response message
  - when receiving the JMS response message, the requestor can compare this *JMSCorrelationID* value with the *messageID* values of the possibly many JMS requests messages sent earlier, to find the corresponding request.

MTOSI is not prescriptive in this recommended approach. Other schemes may be considered as well.

MTOSI header	JMS header	JMS application property	mapping is: - mandatory (M) or - optional (O)
destinationURI		MTOSI_destinationURI	O
senderURI	JMSReplyTo (if replyToURI not present)	MTOSI_senderURI (if replyToURI is present)	M
activityName		MTOSI_activityName	O
msgType		MTOSI_msgType	M
msgName		MTOSI_msgName	M
replyToURI	JMSReplyTo		M
failureReplytoURI		MTOSI_ failureReplytoURI	M
activityStatus		MTOSI_activityStatus	M
payloadVersion		MTOSI_payloadVersion	O
vendorExtensions		MTOSI_vendorExtension	O
security		MTOSI_security	O
securityType		MTOSI_securityType	O
priority	JMSPriority		M
communicationPattern		MTOSI_communicationPattern	M
communicationStyle		MTOSI_communicationStyle	M
requestedBatchSize		MTOSI_requestedBatchSize	M
batchSequenceNumber		MTOSI_batchSequenceNumber	M
batchSequenceEndOfReply		MTOSI_batchSequenceEndOfReply	M

Table 2. Mapping MTOSI header fields to JMS header field or JMS application-specific properties

As an example, from the MTOSI header fields shown in Figure 12 above, the rules specified above will result in the creation of the following values of the JMS header fields and JMS application properties:

MTOSI_destinationURI=”jms/RtoNMSQA”
MTOSI_senderURI=”jms/RtoInventoryOSQA”
MTOSI_activityName=”BulkInventoryRetrieval”
MTOSI_msgType=”REQUEST”
MTOSI_msgName=”getInventory”
JMSReplyTo= The JMS destination (of type Destination) whose JNDI name is ”jms/RtoInventoryOSQA”
MTOSI_ failureReplytoURI=”jms/RtoInventoryOSQA”
MTOSI_payloadVersion=”1.0”
JMSPriority=4
MTOSI_communicationPattern=”SimpleResponse”
MTOSI_communicationStyle=”MSG”

Table 3. Example of mapped values from the MTOSI header fields ("getInventory" request, MSG communication style)

3.6 Mapping SOAP Messages to JMS Messages: MTOSI notifications

MTOSI encourages the usage of JMS Message Selectors for the filtering of messages of interest by consumer JMS Client.

While this mechanism can be used for any purpose, it is relevant mainly in the case of unsolicited JMS Messages, that is to say messages containing MTOSI notifications.

Because JMS Message Selectors cannot reference the JMS Message body, MTOSI mandates the following approach:

R 19. A JMS Client publishing a JMS Message containing an MTOSI notification may transmit to the JMS layer a list of parameters which will be mapped into JMS application-specific property fields in the corresponding message (in addition to the ones presented in the previous section for the mapping of MTOSI header fields).

R 20. The possible candidate parameters correspond to the list of the event filterable attributes as presented in the column named “Filterable” from table 1 “MTOSI Notification Event Attributes” of reference [6].
Decision to use any of those candidate parameters for filtering is left to the implementer. MTOSI mandate that only those candidate parameters may be considered for filtering purpose to the exclusion of any other.

R 21. When mapped into JMS application-specific property fields, the following identifiers should be used as shown below:

Filterable MTOSI Notification Event Attributes	JMS application specific property fields
EventType	MTOSI_EventType
objectName	MTOSI_objectName
objectType	MTOSI_objectType
osTime	MTOSI_osTime
neTime	MTOSI_neTime
edgePointRelated	MTOSI_edgePointRelated
layerRate	MTOSI_layerRate
aliasNameList	MTOSI_aliasNameList
probableCause	MTOSI_probableCause
probableCauseQualifier	MTOSI_probableCauseQualifier
nativeProbableCause	MTOSI_nativeProbableCause
perceivedSeverity	MTOSI_perceivedSeverity
rcaiIndicator	MTOSI_rcaiIndicator
acknowledgeIndication	MTOSI_acknowledgeIndication
transferStatus	MTOSI_transferStatus

Table 4. Filterable MTOSI Notification Event Attributes

4 Future Work

This document will be extended during MTOSI Phase 2. Topics of interest already identified are:

- usage of JNDI in association with naming policies of the MTOSI interfaces (e.g. LDAP)
- Running JMS on different systems
- Performance aspects: handling massive amount of information in few messages (e.g. Inventory retrieval) of massive amount of messages of small size (e.g. notifications).

5 References

[1] Specification document: Java Message Service, Version 1.1, April 12, 2002

[2] J2EE 1.4 and JMS 1.1 Tutorial, http://java.sun.com/j2ee/1.4/docs/tutorial/doc/J2EETutorial.pdf

[3] JSR 144 OSS Common API, Final Draft, http://jcp.org/en/jsr/detail?id=144

[4] OSS/J Design Guidelines, http://www.ossj.org/downloads/design_guidelines.shtml

[5] MTOSI Communication Styles, Supporting Document SD2-5 of the TM Forum MTOSI Release 1.0 Deliverables, 2005.

[6] MTOSI Notification Service, Supporting Document SD2-8 of the TM Forum MTOSI Release 1.0 Deliverables, 2005.

[7] MTOSI XML Implementation User Guide, Supporting Document SD2-2 of the TM Forum MTOSI Release 1.0 Deliverables, 2005.

Revision History

Version	Date	Description of Change
1.0	May 2005	This is the first version of this document and as such, there are no revisions to report.
1.1	Dec 2005	Replaced “originatorURI” by “senderURI” Replaced “operationStatus” by “activityStatus” Removed “Domain”

Acknowledgements

Scott Toborg Cramer

Graham Glendinning Cramer

How to comment on the document

Comments and requests for information must be in written form and addressed to the contact identified below:

Michel Besson Cramer

Phone: +44 7717 692 178

Fax:

e-mail: Michel.Besson@cramer.com

Please be specific, since your comments will be dealt with by the team evaluating numerous inputs and trying to produce a single text. Thus we appreciate significant specific input. We are looking for more input than wordsmith” items, however editing and structural help are greatly appreciated where better clarity is the result.