

MTOSI Communication Styles

Abstract

This document outlines the top down approach followed by MTOSI to define the technology neutral abstract interfaces and the various technology specific concrete solutions set.

Table of Content

1	Introduction	2
2	Communication Architecture	2
2.1	Two different communication styles.....	5
2.2	Semantics and processing model of the two communication styles	6
2.2.1	RPC Style sequence description	6
2.2.2	MSG Style sequence description	7
2.3	Implications of the two communication styles in the abstract interface signature.....	7
2.4	Mapping the communication styles to a transport	7
2.4.1	RPC style with a synchronous transport.....	7
2.4.2	RPC style with a asynchronous transport.....	7
2.4.3	MSG style in synchronous transport.....	8
2.4.4	MSG style in asynchronous transport.....	9
2.4.5	Style transport mapping summary	10
3	Message Exchange Patterns.....	10
3.1	Simple Response pattern:(SRR) (ARR).....	11
3.2	Multiple Batch Response Communication Pattern	12
3.2.1	Synchronous Iterator (SIT) MEP	12
3.2.2	Asynchronous batch response (ABR).....	12
3.3	Bulk Response Pattern	14
3.3.1	Synchronous (File) Bulk Response (SFB) MEP.....	14
3.3.2	Asynchronous (File) Bulk Response (AFB) MEP.....	15
3.4	Notifications.....	15
3.4.1	The MTOSI topics	15
3.4.2	Publishing in MTOSI.....	16
3.4.3	Receiving notifications in MTOSI.....	16
3.4.4	The Selector syntax.....	18
4	Summary.....	18
5	Revision History	20
6	Acknowledgements	20
7	How to comment on the document.....	20

1 Introduction

This document describes certain interactions between OS to OS such as an Inventory System and Discovery OS. To aid understanding of these interactions, and the terminology used in the rest of this document, the following high-level conceptual architecture diagram is presented as specific example.

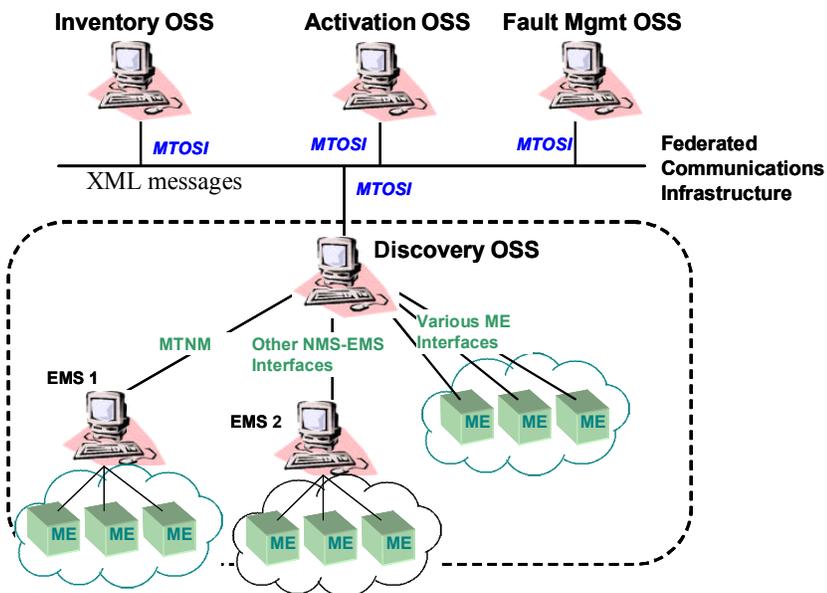


Figure 1: High level architecture

The Inventory OS interacts with the underlying Discovery OS by the sending and receiving XML messages..

2 Communication Architecture

The MTOSI team identified the need to be transport independent. Being transport independent will allow replacing the underlying transport without changing the application code (and the application logic) of both the OS client and OS server. This property is achieved by keeping untouched the MTOSI messages as the specific transport is deployed.

In order to meet this requirement a service oriented façade design pattern is used [GAM]. Similar to the CORBA broker architecture, the MTOSI team has defined an abstract interface that is transport technology agnostic and the encapsulation of the mappings to different transport in generic modules called bindings.

The following diagram (Figure 2) shows the detailed communication architecture.

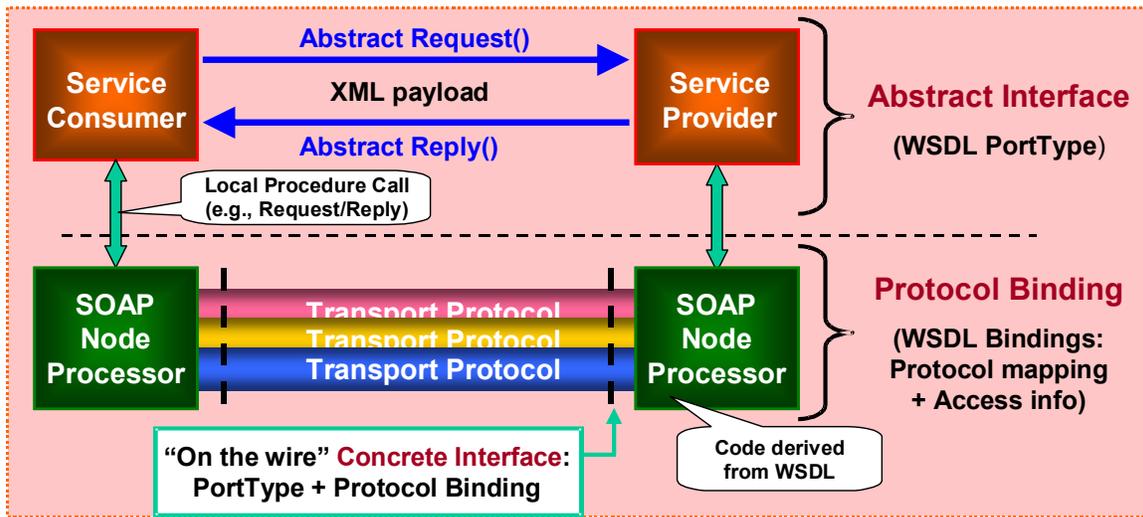


Figure 2 - MTOSI communication architecture

A service consumer interacts with a service provider through the invocation of an operation to execute a *Business Activity* achieving a *business goal*. The operation involves an exchange of XML messages (XML payload). A *communication pattern* (as described further in Section 2.1) identifies the sequence and cardinality of the messages sent and/or received as well as whom they are sent or received from. The messages are exchanged by the application to the *SOAP Node Processor* (usually middleware) according to a *communication style*: RPC or message (MSG) (see Section 2.2). The Soap Node processors implement the bindings for a supported transport and are responsible for the marshalling and un-marshalling of the XML messages and meta information to the wire format protocol.

The combination of a communication pattern and a communication style fully identify the messages and the choreography (sequencing and cardinality) of messages involved in a business activity, which we call a Message Exchange Pattern (MEP) [SOA] [WSD] [WSD2]. The combination of the MEP and the message types (XML Schemas) fully specify an interface at the abstract level. By adding the transport protocol details (Bindings) to the abstract interface we define a the concrete (transport specific) interface.

The following figure (Figure 3) illustrates how a business goal addressed by an abstract operation can be mapped to a communication style, pattern, and protocol.

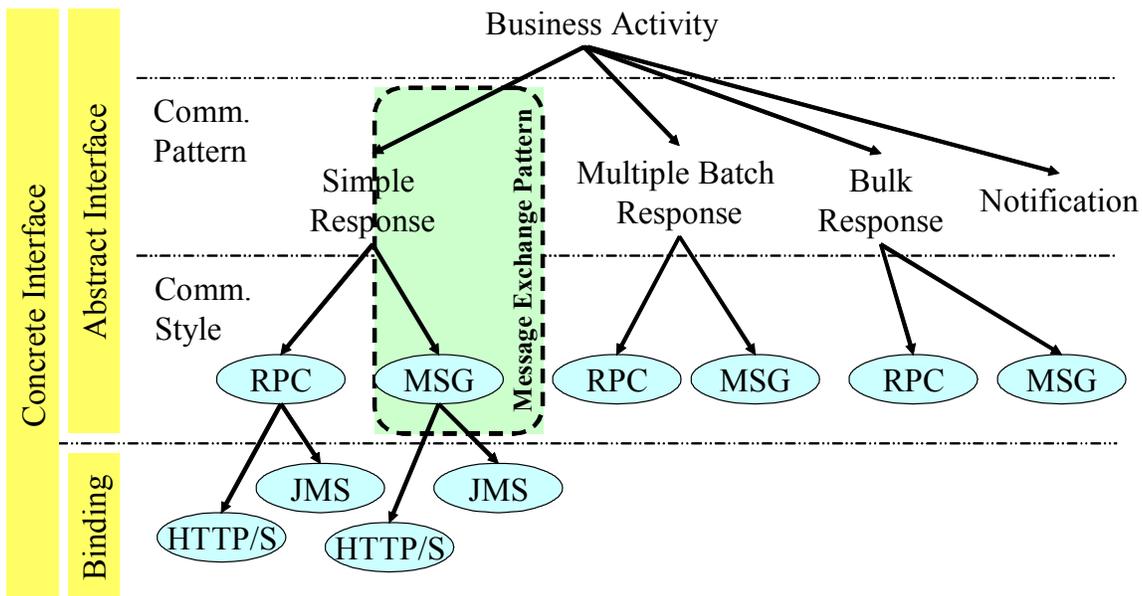


Figure 3 - Mapping an operation to communication styles and transport protocols

The picture also illustrates that an operation with a communication pattern and a style defines an abstract interface and with the addition of a protocol binding it defines a concrete interface. These concepts are explained further in the following subsections.

2.1 Communication patterns

A Communication Pattern identifies the actors, their role in the communication, and the abstract type of messages sent and/or received (e.g., request, response, notification, error).

We identified four distinct Communication patterns in MTOSI V1.0:

- Simple Response
- Multiple Batch Response
- Bulk Response (e.g. file transfer)
- Notification

These Communication patterns address different communication needs: while the first three are oriented towards an exchange of information between two parties in a business activity (P2P), the notification communication is designed to disseminate information to a set of recipient (pub/sub), possible greater than one.

In the MTOSI methodology, the design of a service realizing a business activity includes selecting one or more of these communication patterns.

For example, the `getInventory` business activity is likely to require result sets to be partitioned into several chunks and sent to the service consumer according to the multiple-batch-response business communication pattern. A communication pattern defines the collaboration as a high-level choreography without specifying how it is actually carried out. A communication pattern is an abstract concept and is analogous to the concept of WSDL Transmission Primitive in WSDL 1.1 [WSD] within the `portType` or the WSDL Message Exchange Pattern in WSDL 2.0 [WSD2] within the `Interface`.

2.2 Two different communication styles

A **communication style** identifies the interaction between a service implementation (or consumer) and its SOAP processor (often referred as communication middleware). Occasionally the SOAP processor may be part of the application where the specific transport binding is not available off the shelf or where full control of the bindings is required. Nevertheless, a logical boundary should be identifiable between the application implementing the business logic (or processing a service response) and the component responsible for the marshalling and un-marshalling the messages. Two communication styles are defined for MTOSI: RPC style and Message Style (defined below). This concept of style is common to the WSDL V2.0 specification and it is also called Style [WSD2].

It should be noted that a Service consumer at the abstract level should be able to bypass the soap processor and access a service provider by simply invoking the abstract interface according to the communication style. This abstraction simplifies the description of the styles, allowing us to conceptualize the messages exchanged in a provider/consumer configuration.

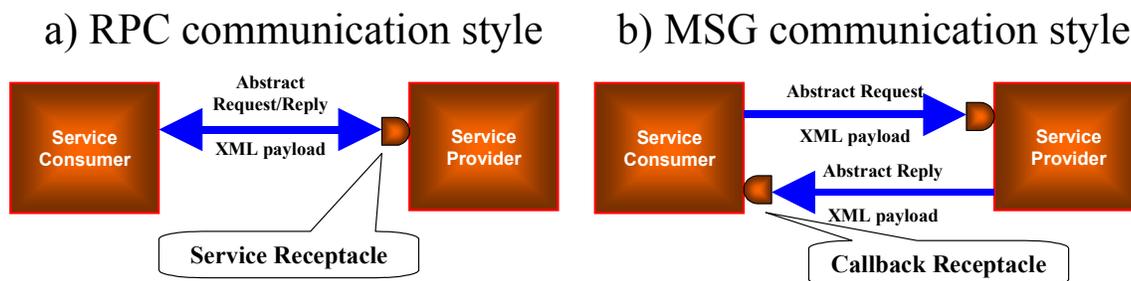


Figure 4 - Communication styles

In the **RPC communication** style (Figure 4a) the Service consumer invokes the service providers through a service receptacle and receives a response as return argument. The call to the middleware is a blocking synchronous call and implements the Remote Procure Call semantics. This interaction is blocking for the process or thread that invoked the operation.

In the **Messaging communication** Style (Figure 4b), the service consumer invokes the service by sending a request message through the Service receptacle but at the same time exposes a callback receptacle. The Service provider will then respond by sending the reply message to the callback receptacle. This interaction is non-blocking.

Beside the different coordination mechanics, the significant difference between these two styles lies in the exposure of the callback receptacle in the MSG style. These differences have implications in the operation signature as well as in the business patterns built on top of the communication styles.

Note that the callback receptacle is a logical entity and it may be implemented in different ways. It could be simply a service exposed by the consumer (e.g. HTTP URL) or a combination of a topic and correlation ID to filter the relevant messages.

Furthermore, the message originated in the MSG Style should not be confused with a Notification message. A notification is a higher level Communication Pattern used to disseminate information. It is possible to implement a Notification Pattern on top of a MSG Style, as suggested in Figure 4.

These styles are not interchangeable at the business level. The message style is somehow richer than the RPC. In the message style, the service producer has the ability to expose the state of its own private transaction flow related to an operation invocation. The RPC style does not offer the capability to produce more than one

response per invocation. For this reasons, while it is possible to reduce a MSG style exchange to an RPC style by ignoring the intermediate messages, the vice versa may not always be viable. To upgrade the RPC to a MSG style it is necessary to have access to the internal (private) process state of the service provider in order to produce the additional messages (e.g. intermediate state changes) .

2.3 Semantics and processing model of the two communication styles

The next figure illustrates how to use the RPC and MSG styles to carry out a simple business transaction between two parties. The business transaction will invoke a function on the service provider with formal arguments and return an output result.

a) RPC communication style b) MSG communication style

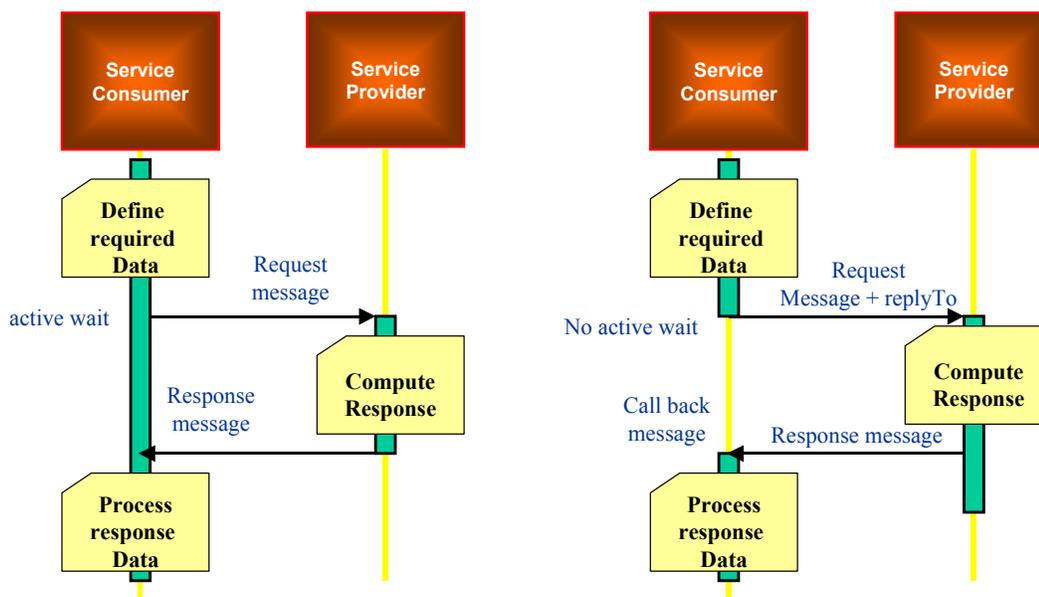


Figure 5 - RPC/MSG Style sequence diagram

2.3.1 RPC Style sequence description

Figure 5a describes the following sequence of synchronous events:

- A request message is generated and passed to the service Provider with a synchronous blocking call.
- The Service Consumer blocks on the call and waits for a response or failure notification.
- The Service Provider computes a response and replies with a Response message using the logical request communication channel (e.g. TCP/IP socket, HTTP session, IIOP session, etc)
- The Service Consumer receives the result
- A synchronous transport such as CORBA, or HTTP/S will natively support this interaction pattern.

A significant number of the MTO SI operations are by its nature request with a single reply. With some exceptions e.g. (bulk retrieval), a request is processed in a reasonable time and a single reply can carry the response back to the client.

2.3.2 MSG Style sequence description

Figure 5b describes the following sequence of synchronous events:

A request message is generated and passed to the Service Provider with an asynchronous call.

- The Service Consumer will not block on the call but rather be notified later when a response is available.
- The Service Provider computes a response and replies with a Response message using the ReplyTo logical channel exposed by the Service consumer.
- The Service Consumer receives and correlates the result, resumes its thread control, and then processes the response.

An asynchronous transport such as JMS, or MQ will natively support this interaction pattern.

2.4 Implications of the two communication styles in the abstract interface signature

The MSG communication style requires the Service Consumer to expose a callback receptacle, the identity of which is sent back to the Service Provider in the request message. This field is named “ReplyTo” and is specified in the header.

The MSG communication style also requires a CorrelationID to be used by the service consumer to correlate the acknowledgement and responses to the original service request. This field can also be specified in the header.

2.5 Mapping the communication styles to a transport

Both Communication styles can be mapped (with various degrees of difficulty) to different transport fabrics with different native characteristics. The transport capability to synchronously connect the parties or asynchronously store and forward the messages plays a major role in mapping the two communication styles.

2.5.1 RPC style with a synchronous transport

Mapping the RPC style to a synchronous transport is straightforward. A request can be carried out from the transport and the result will be received on the same logical communication channel established for the request. The call from the service consumer is blocking and both consumer and producer need to be active at the same time.

2.5.2 RPC style with an asynchronous transport

Mapping the RPC style to an asynchronous transport requires synchronization to be implemented in the layer between the application and the transport. A service consumer will invoke the middleware with a blocking call according to the RPC style semantic. The binding code (located in the application or in the middleware) in the binding adapter (a.k.a the SOAP node processor) will handle the dispatching and synchronization of the messages through an asynchronous transport. The following sequence diagram illustrates an example of a possible implementation.

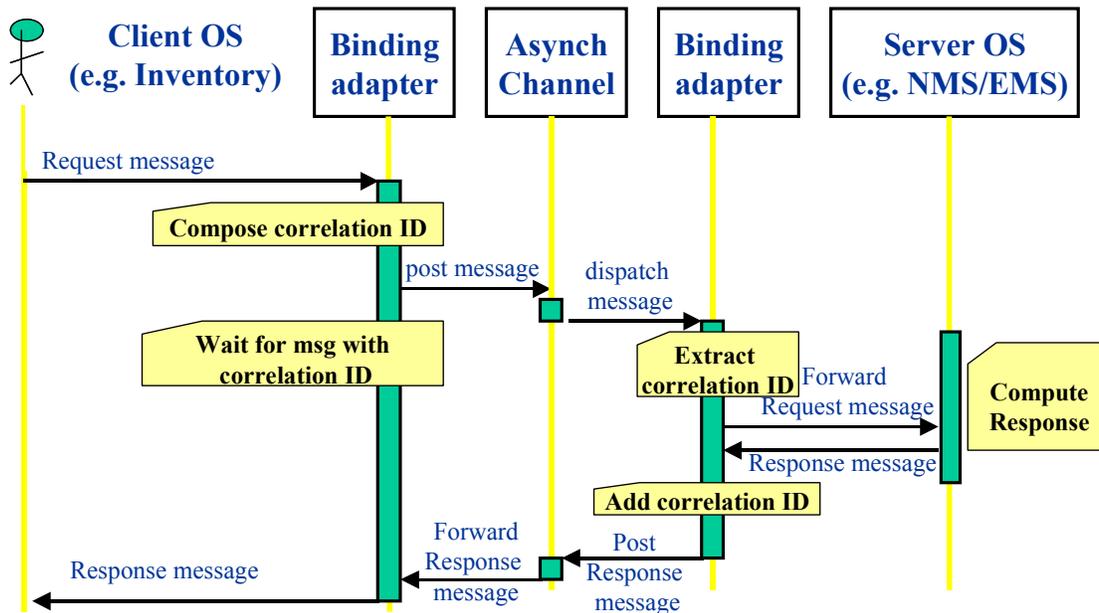


Figure 6 – RPC Request/Reply over Asynchronous transport

This previous diagram describes the following sequence of asynchronous events:

- A request message is generated and passed to the OS messaging adapter (this is the binding adapter on client-side of the diagram).
- The binding adapter generates a correlation ID (possibly suggested by the application).
- The binding adapter posts the request message in the logical channel related to the server OS.
- The binding adapter waits (passive wait) for a return message with the correlation ID.
- The server OS binding adapter (this is the binding adapter on the server-side of the diagram) receives the request message extracts and holds the correlation id.
- The OS binding adapter calls the relevant method on the underlying API.
- Based on the response from the Server OS, the OS binding adapter builds a Response Message which includes the correlation ID, which is passed back to the client OS via the Asynch Channel and Binding Adapter on the client-side.
- The client OS message adapter detects a message with the proper correlation ID and forwards it to the client OS.

2.5.3 MSG style in synchronous transport

In order to map a MSG style in synchronous transport the binding adapter needs to implement a “store and forward” policy to decouple the service consumer from the service provider. This is similar to adding “reliability QOS to the transport”.

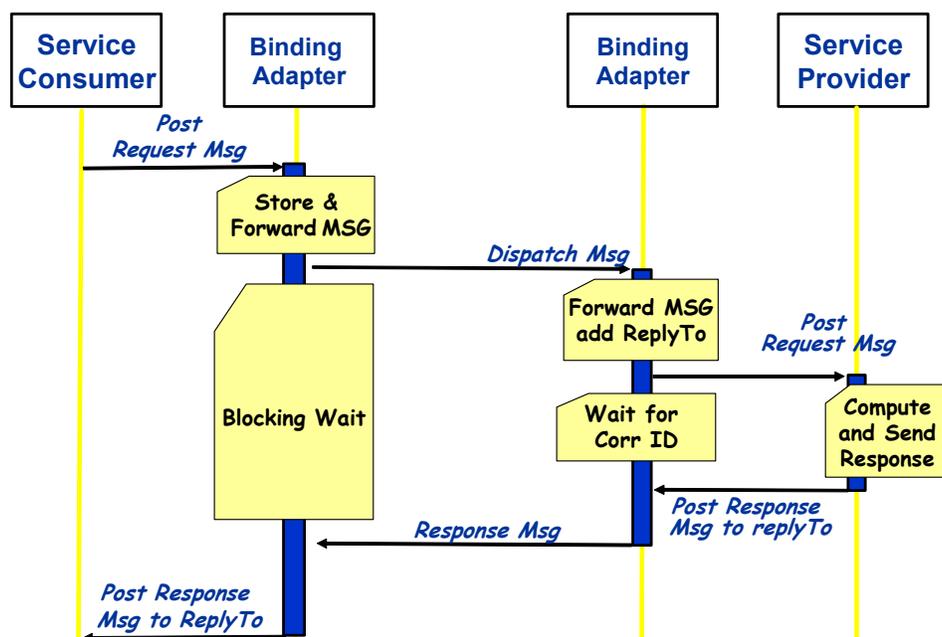


Figure 7 - MSG Request/Reply over Synchronous Transport

This previous diagram (Figure 7) describes the following sequence of synchronous events:

- A request message is generated and passed to the OS messaging adapter along with a call back handler. (this is the binding adapter on client-side of the diagram). This is a NON blocking call.
- The binding adapter stores the request message
- The binding adapter posts the request message to the target destination with a synchronous call.
- The above step may be repeated in case of communication failure according to a reliability protocol (see [WSR]).
- The binding adapter waits (active wait) for a return message.
- The server OS binding adapter (this is the binding adapter on the server-side of the diagram) receives the request message.
- The OS binding adapter calls the relevant method on the underlying API and waits for a response.
- Based on the response from the Server OS, the OS binding adapter builds a Response Message, which is passed back to the client OS resuming the pending synchronous call initiated by the Service consumer Binding Adapter..
- The client OS binding adapter receives a message forwards it to the client OS invoking the specified call back handler.

2.5.4 MSG style in asynchronous transport

The MSG style maps natively to an asynchronous transport.

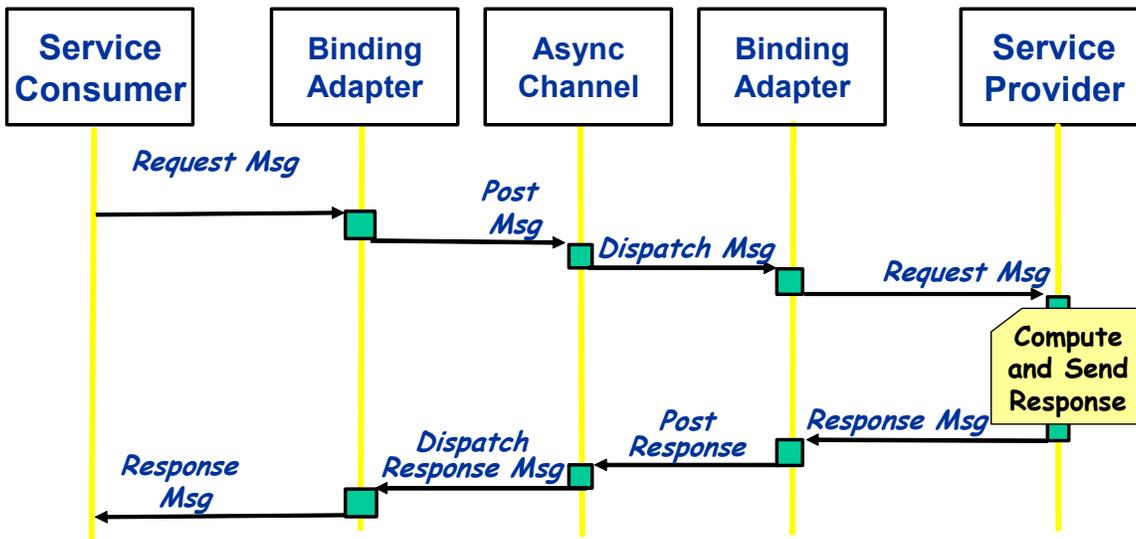


Figure 8 - MSG Request/Reply over Asynchronous Transport

The above diagram (Figure 8) highlights that the binding adapter is simply passing the message to the asynchronous channel and vice versa. No specific logic is required since the asynchronous transport natively implements the MSG style.

2.5.5 Style transport mapping summary

The following table (Table 1) summarize the possible combinations mapping the communication style to a transport

Table 1 Communication Styles and Transport Type

	Transport type	
Communication Style	Synchronous (HTTP/S, IIOP)	Asynchronous (JMS, MQ,SMTP)
RPC	Maps natively	Binding adapter needs to handle messages correlation (correlation ID)
MSG	Binding adapter needs to handle <i>store and forward</i> semantic	Maps natively

3 Message Exchange Patterns

A Message Exchange Pattern (MEP) is the combination of a business communication pattern and a communication style and fully identifies the messages and the choreography (sequencing and cardinality) of messages independently from a business activity. A MEP can be equated to a SOAP MEP [SOA].

Table 2 summarizes the possible 8 combinations of communication styles and communication patterns into individual MEPs.

This table represents the MTOSI MEP portfolio: Each business activity can reference to one or more MEP to fully identify the mechanism to achieve the business goal in the MTOSI specification.

Table 2 - MEPs used in TMF 854

Message Exchange Pattern (MEP)	Communication Pattern			
Communication Style	Simple Response	Multiple Batch Response	(File) Bulk Response	Notification
RPC (Synch) Note – None of the RPC styles is supported by MTOSI v1.0.	SRR Synchronous Request/Reply	SIT Synchronous Iterator	SFB Synchronous (File) Bulk	WSN Web Services Notification
MSG (Asynch)	ARR Asynchronous Request/Reply	ABR Asynchronous Batch Response	AFB Asynchronous (File) Bulk	WSN Web Services Notification

The following MEPs are used for the operations in MTOSI:

- Synchronous Request/Reply (SRR) and Asynchronous Request/Reply (ARR) – Message (SRM) are used for requests that have a single response.
- Synchronous Iterator (SIT) – this MEP allows for a synchronous (i.e., RPC style) request for an iterator.
- Asynchronous Batch Response (ABR) – this MEP allows for an asynchronous (i.e., message style) request for a multiple batch response.
- Synchronous (File) Bulk (SFB) – this MEP allows for a synchronous (i.e., RPC style) request for inventory to be returned in a file. The file is delivered via an out-of-band method (i.e., not using the CCV).
- Asynchronous (File) Bulk (AFB) - this MEP allows for an asynchronous (i.e., message style) request for inventory to be returned in a file. The file is delivered via an out-of-band method (i.e., not using the CCV).
- Web service Notification – to disseminate information (see Section 3.4 for a detailed description)

3.1 Simple response pattern: (SRR, ARR)

The *simple response pattern* involves a request/reply with a single result message. This pattern maps directly into the two native communication styles. Using RPC style the sequence diagram of Figure 5 applies at the business level. An acknowledge message may also be sent from the Service provider to the service consumer upon receiving a service request.

For example, the *out = getTP(in)* operation defined in the managedElementManager will have two variants:

- **SRR** *out = getTP(in)*
- **ARR** *out getTP(in + replyTo+ CorrelationID)*

3.2 Multiple batch response communication pattern

Handling a large result data set requires some additional coordination between the service consumer and producer. The Iterator design pattern is usually deployed in this situation to provide such coordination. Nevertheless the two communication styles natively lead to an Iterator pattern with significant differences in terms of flow control.

3.2.1 Synchronous iterator (SIT) MEP

This is the classical Iterator design pattern [GAM]. The response of the first invocation returns a partial data set as well as a pointer to an Iterator interface. The service consumer will then invoke the Iterator to receive the subsequent result data set partitions. The consumer has control of the flow, the service provider needs to maintain the state related to the pending Iterator. The following example illustrates a typical interaction.

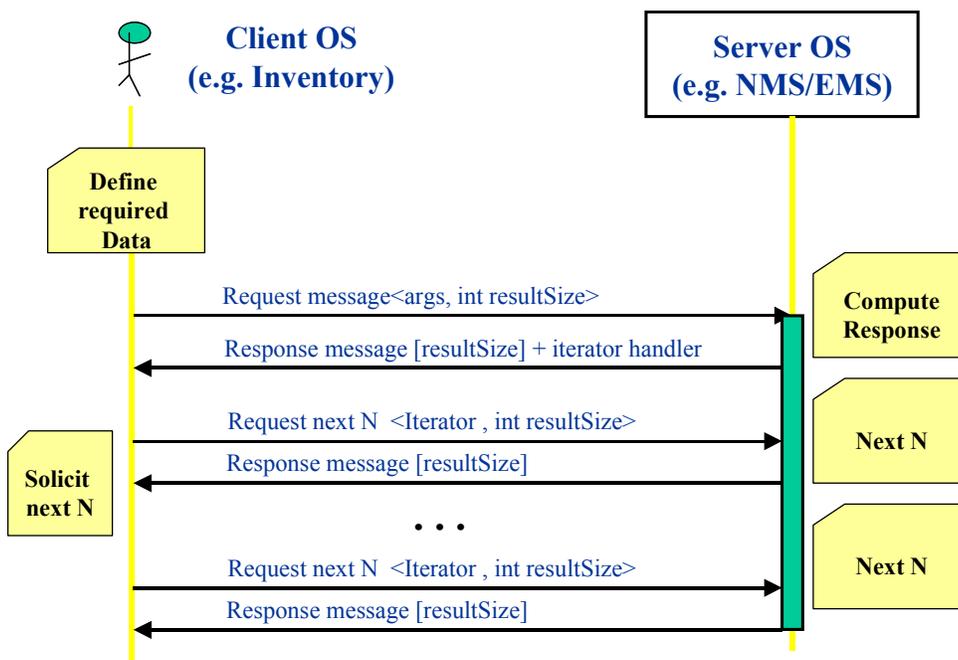


Figure 9 - Synchronous Iterator design pattern

The Iterator state on the server side can be controlled using a timeouts-based garbage collection mechanism. This is the common mechanism used in the MTNM CORBA interface to retrieve multiple result sets.

3.2.2 Asynchronous batch response (ABR) MEP

Using the MSG communication style in combination with the Multiple Batch Response Communication Pattern leads to a variation of the pattern with flow control in the service provider. The response of the first invocation returns an acknowledgement. The result set will then be sent in chunks to the service consumer (via the call back receptacle) as the data becomes available in the service producer. The consumer has usually control over the size of the chunks specified in the initial call. The following example (Figure 10) illustrates a typical interaction. This is the mechanism implemented in the OSS/J design guidelines [OSSJ].

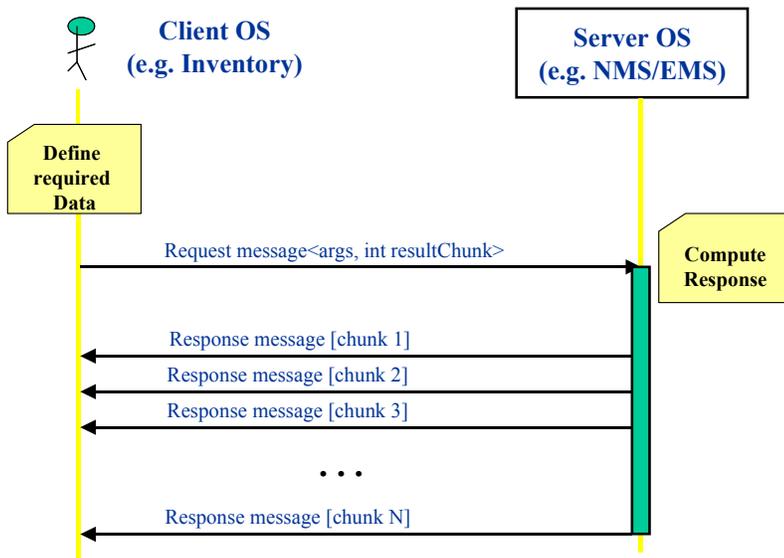


Figure 10 - Asynchronous batch response design pattern

The following example summarizes the mapping of the Multiple Batch Response Communication Pattern to the different styles RPC and MSG.

For example, the $out[] = getAllPTPs(in)$ operation defined in the managedElementManager will have two variants:

- **SBR** $out[i..j] + iterator = getAllPTPs(in + requestedBatchSize)$

The request will need to specify the size of the result set returned as response from the initial invocation. The service provider will also return a reference to an Iterator interface. The service consumer will then invoke the Iterator to get the subsequent result data set.

- $out[i..j]=next_n(n)->Iterator$

We can use the same signature adopted in the CORBA MTNM implementation TMF814. (See supporting document: overview of Iterator usage Overview of Iterator Usage)

- **ABR** $out[i..j]= getAllPTPs (in + requestedBatchSize + replyTo+ CorrelationID)$

The Asynchronous batch Response does not need the additional Iterator interface since the service producer it will directly fragment the result and send it to the service consumer.

Although with slight different semantic, the formal argument “**requestedBatchSize**” is common to both Iterator and can be carried in the message header of the request. How may is the number of elements in the batch. A value of 0 (zero) will imply the entire data result set in a single response.

The “replyTo” callback identifier needs to be provided in the MSG Asynchronous Iterator request. This field can be provided in the header section of the request message.

Note that although the Iterator responses are similar in nature to Notifications they are not Notifications. Notifications should be used to disseminate information (such as alarms and/or state changes) and not to convey a result data set.

3.3 Bulk Response Pattern

This pattern enables to transfer the result XML payload using an additional specialized protocol different from the one used to carry the messages conversation. For instance, with this pattern it is possible for a service consumer to request a service provider to upload the bulk result set to an ftp server according to the FTP protocol.

This pattern can be further mapped in the two styles: RPC and MSG. It should be noted that the actual payload will be transferred off-band in both the communication style variants.

3.3.1 Synchronous (File) Bulk Response (SFB) MEP

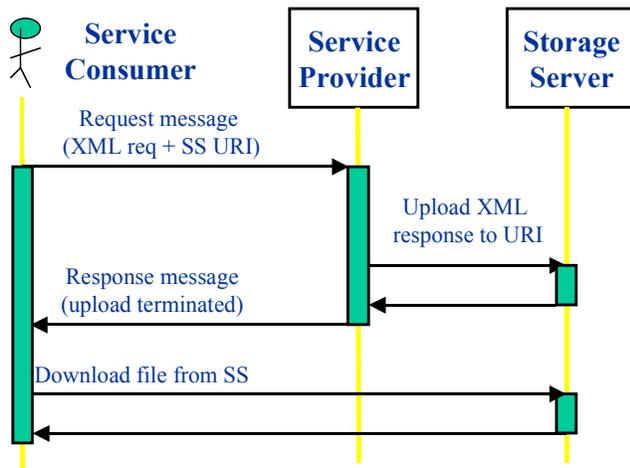


Figure 11 - Bulk transfer RPC style

In this MEP the service consumer request a response set to be uploaded in a storage server and the blocking call returns when the transfer is complete.

3.3.2 Asynchronous (File) Bulk Response (AFB) MEP

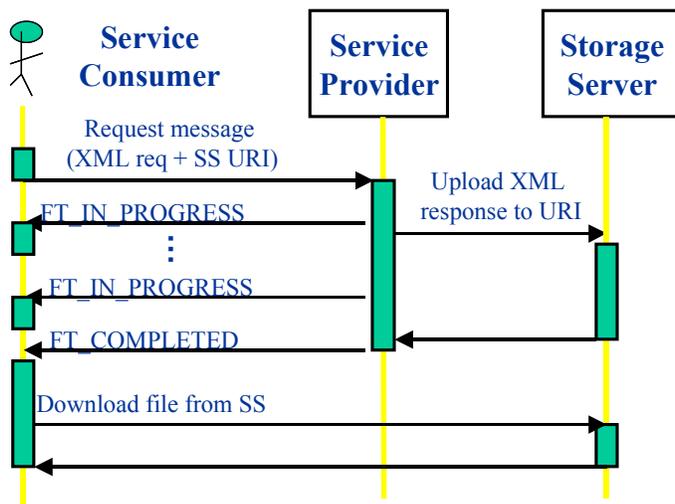


Figure 12 - Bulk transfer MSG style

In this pattern, the initial request is non-blocking and the service consumer gets notified when the transfer is completed. The `NT_FILE_TRANSFER_STATUS` notification (defined in the MTNM specifications) is used to indicate when the file transfer is complete or when a failure has occurred. The number of events indicating `FT_IN_PROGRESS` that will be transferred is an implementation decision for the designer of the *target OS*. However, at least one event indicating `FT_COMPLETED` with `percentComplete=100`, or `FT_FAILED` with a supplied `failureReason` is mandatory.

3.4 Notifications

The notification communication is designed to disseminate information to a set of recipient (pub/sub), possible greater than one. MTOSI leverages on a subset of features defined in the Web Service Notification specification [WSN]. The Web Service Notification specification [WSN] has been proposed by IBM et. al. and it is currently V1.0 as of 1/20/2004. The purpose of this spec is to “specify a standard Web services approach to notification using a topic-based pub/sub pattern”. Given the transport independent nature of web service, this specification is a good candidate for the MTOSI notification mechanism. Nevertheless the WS-notification is more sophisticated than what we may find useful in the first release of MTOSI. The rest of this document will present a minimal subset of concepts and mechanisms to be proposed as MTOSI notifications. To be fair, WS-Eventing [WSE] [WSE2] is another proposal submitted by IBM, BEA Systems, Microsoft, Computer Associates, Sun Microsystems, TIBCO Software, overlapping in some parts with the WS-Notification. From what we see on the web, it seems that WS-Notification and WS-Eventing are slowly being aligned. The following paragraphs highlight a brief view of the MTOSI notification from the user perspective. Refer to SD2-8 MTOSI Notification Service for the details and in depth specification of the notification mechanism in MTOSI.

3.4.1 The MTOSI topics

A topic is a logical entity identifying a related stream of notifications. In MTOSI, we decided to have the following Topics.

- Inventory Topic
- Fault Topic
- Protection Topic

- File Transfer Topic

3.4.2 Publishing in MTOSI

This proposal advocates the “simple publishing” mechanism described in the WS-notification spec.

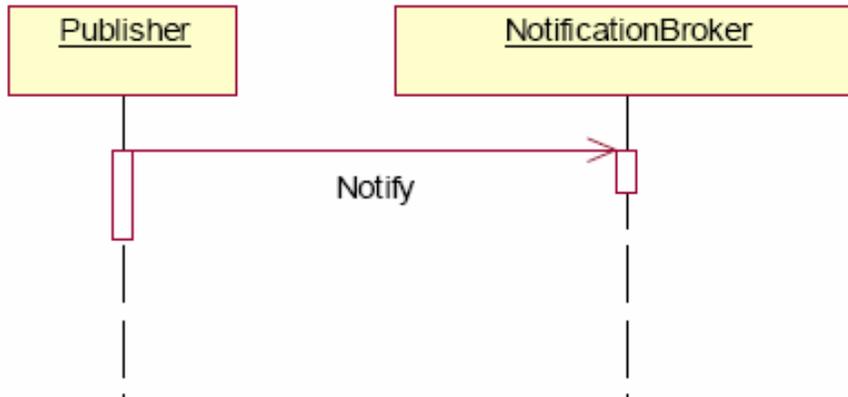


Figure 13 - WS-Notification simple publishing

A Publisher sends a message to a NotificationBroker and the Notification broker (typically middleware) will take care of disseminating the information.

The notification operation exposed by the NotificationBroker should have the following signature:

Notify(TopicPathExpression, Message)

Where:

TopicPathExpression - identifies the unique name of the MTOSI topic the publisher intent to publish to.

Message - is the payload message fully described by the MTOSI XSDs.

Note WS-notification has an additional optional Publisher registration to the broker to allow security.

We can/may introduce this step in phase II.

3.4.3 Receiving notifications in MTOSI

In WS-notification there is a distinction between the actor requesting a subscription (service requestor) and the actual actor receiving the stream of event notifications (Notification consumer).

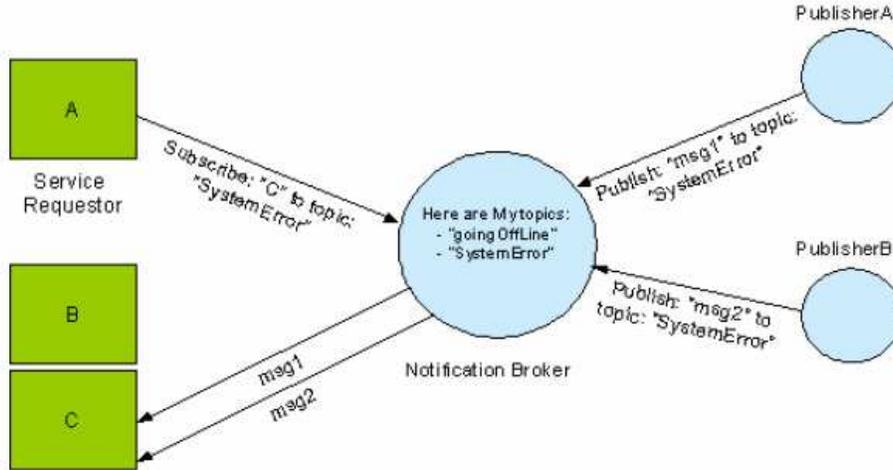


Figure 14 - WS-Notification, Brokered subscription

In MTOSI we probably safely assume that a Notification consumer is also the entity initiating the subscription request. This assumption will not change the notification API but it will simplify the mechanism.

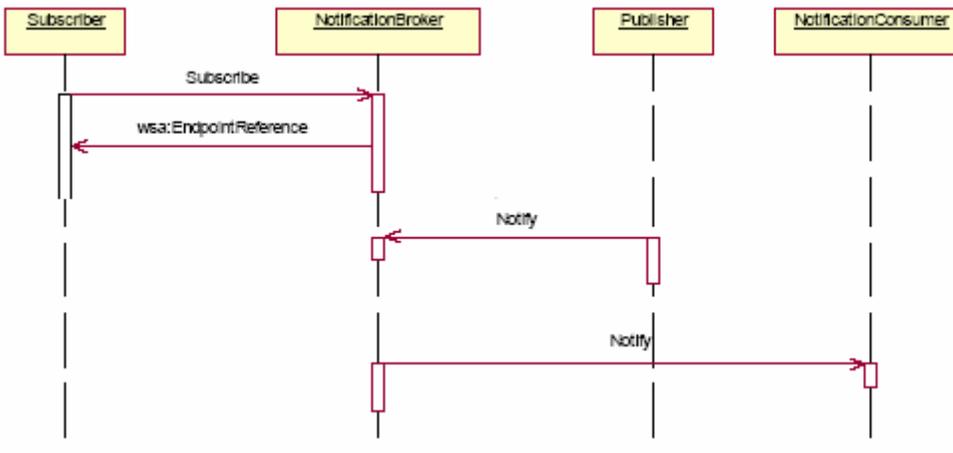


Figure 15 - WS-Notification, brokered publisher

Figure 15 further simplified the sequence diagram propose by WS-notification by removing the subscriber topic lookup in the broker and Publisher registration with the broker.

The notification consumer has to have the capability to receive notifications sent from the broker. This translates into exposing an operation with the following signature:

Notify(TopicPathExpression, Message)

Where:

TopicPathExpression identifies the unique name of the topic on which the message was published.

Message is the payload message fully described by the MTOSI XSDs.

Note this is the same operation the broker exposes to the publisher.

The subscriber (possibly the same entity as the notification consumer) has to notify the broker of its intention to receive events from a topic. The subscriber also has an option to further constraint the messages received by specifying a filter at the subscription time.

The broker has to expose the following operation:

Subscribe(ConsumerEndpointReference, TopicPathExpression, [Selector])

returns: WS-Resource qualified EPR to a Subscription

where:

ConsumerEndpointReference - is the endpoint (callback handler) that the broker will call to send the notification.

TopicPathExpression - Is the unique name of the topic object of the subscription

Selector Is the optional “filter” expression further restricting the flow of messages dispatched to the notification consumer.

3.4.4 The Selector syntax

WS-Notification provides the XPATH syntax for expressing the filter. While the XPATH notation is a rich and effective language for constraining XML, it is important to keep an eye on the possible implementation. When the notification broker is implemented as a JMS broker, the JMS subscriber has a capability to set a JMS selector to constraint the JMS messages. This JMS selector syntax is less XML oriented and the scope of the predicates are bound to the header and additional application specific properties [SD2-9]. In MTOSI phase I we decided to structure the selector in two parts: an identifier specifying the selector format, and the selector itself. In this way we can easily support the JMS selector syntax in the subscription implemented in JMS.

4 Summary

We identified two Communication Styles in the context of the MTOSI OS to OS interactions: Remote Procedure Call (RPC) and Message (MSG). Each Communication Styles, RPC and MSG can be supported (with different efforts) by both Synchronous and Asynchronous Transport fabrics with the same signature and without changing the applications. Nevertheless, these different styles, while pursuing the same business transaction objective (accessing a service), have a distinct signature and behaviour in terms of coordination. As a consequence, the style conditions the business communication Patterns (Req/Reply, Iterator, Notification) into two classes: Synchronous and Asynchronous. While a synchronous behaviour is more effective in a tight integration such as the NMS/EMS relationship, the Asynchronous behaviour is more suitable to a loosely integrated application such as the OS to OS ecosystem. Since MTOSI is defining the services that will be used in these two different contexts we are proposing the adoption of both Communication Styles RPC and MSG. MTOSI will define operations with these two different styles and related Business Patterns (e.g Iterator) with different flavour. It will be the provider responsibility to state what operation are implemented and offered to the service consumer. Although in MTOSI phase I we focus only on the Message Exchange Patterns related to the MSG communication style the subsequent phases can easily extend the scope to RPC.

References

[SD2-8] MTOSI Notification Service

[SD2-9] Using JMS as an MTOSI Transport

[TMF854] MTOSI R1.0 XML Solution Set

[GAM] Gamma, Helm, Johnson, and Vlissides. Design Patterns. Addison-Wesley, 1995

[WSD] Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001, <http://www.w3.org/TR/wsdl>

[WSD2] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Working Draft 3 August 2004, <http://www.w3.org/TR/wsdl20/>

[SOA] SOAP Version 1.2, W3C Recommendation 24 June 2003, <http://www.w3.org/2000/xp/Group/>

[WSR] WS-Reliability 1.1, K. Iwasa, ed., OASIS Web Services Reliable Messaging TC, Committee Draft 1.086, 24 August 2004

[WSN] WS-Notification version 1.0 – 1/20/2004 - http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn

[WSE] WS-Eventing– August 2004 - <http://www-106.ibm.com/developerworks/webservices/library/specification/ws-eventing/>

[WSE2] WS-Events version 2.0 - 21/07/2003 - <http://devresource.hp.com/drc/specifications/wsmf/WS-Events.jsp>

[OSSJ] OSS through Java Initiative - <http://java.sun.com/products/oss>

5 Revision History

Version	Date	Description of Change
1.0	May 2005	This is the first version and as such, there are no changes to report.
1.1	Dec 2005	Applied member evaluation feedback.

6 Acknowledgements

<FirstName>	<LastName>	<Company>
Michel	Besson	Cramer
Francesco	Caruso	Telcordia Technologies Inc.
Shlomo	Cwang	TTI Telecom
Felix	Flemisch	Siemens
Steve	Fratini	Telcordia Technologies Inc.
Elisabetta	Gardelli	Siemens
Jérôme	Magnet	Nortel Networks

7 How to comment on the document

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

Francesco	Caruso	Telcordia Technologies Inc.
Phone:	+1 732 699 3072	
Fax:	+1 732 699 7015	
e-mail:	caruso@research.telcordia.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.