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 Contents

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.

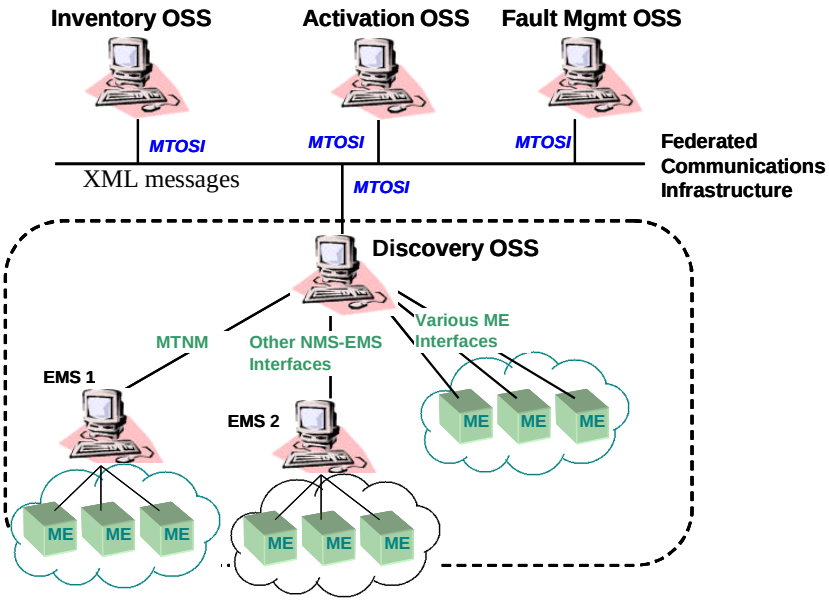

Figure 1: High level architecture

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.

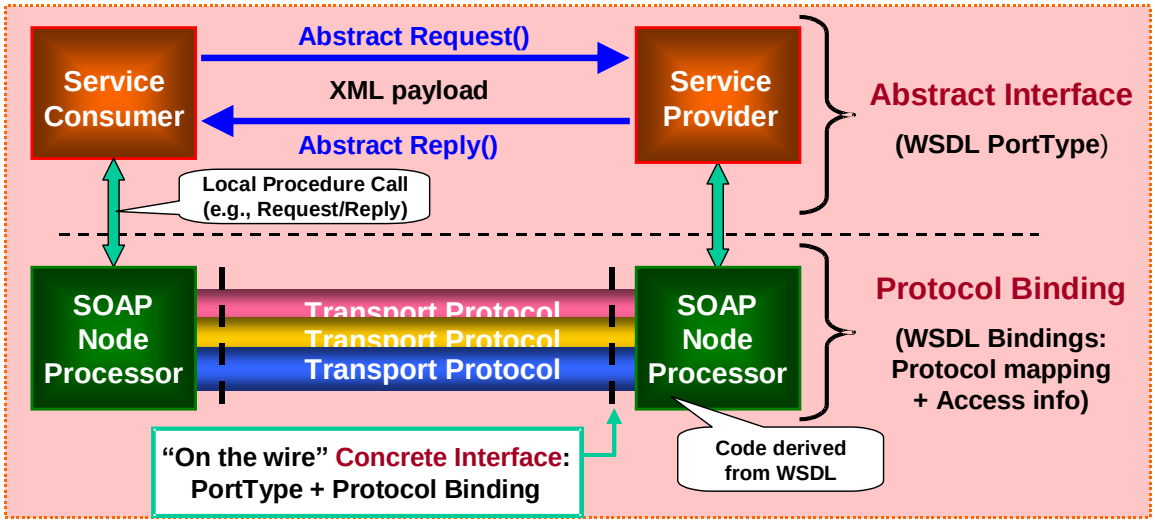

Figure 2 - MTOSI communication architecture

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 to 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 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.

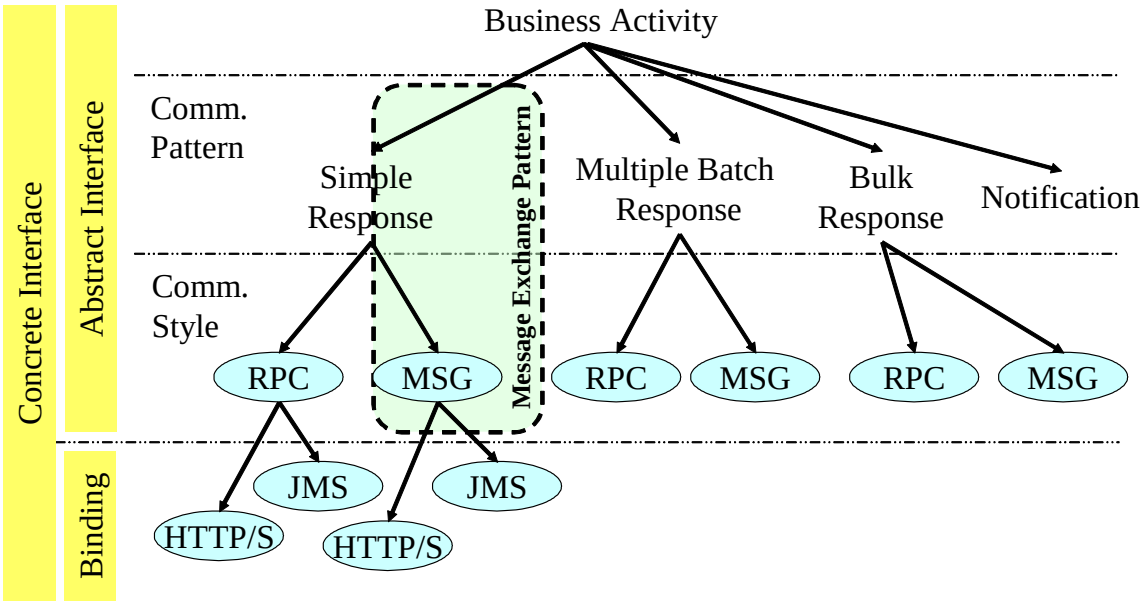

Figure 3 - Mapping an operation to communication styles and transport protocols

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:

- 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 one or more recipients (pub/sub).

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 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.

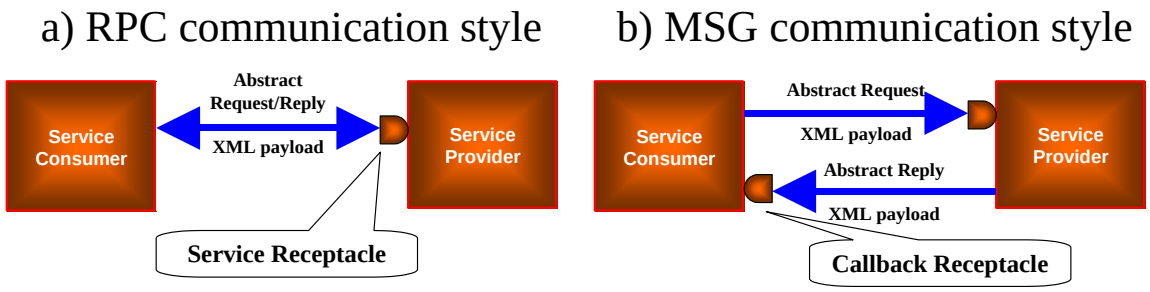

Figure 4 - Communication styles

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 mechanisms, 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 reverse 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

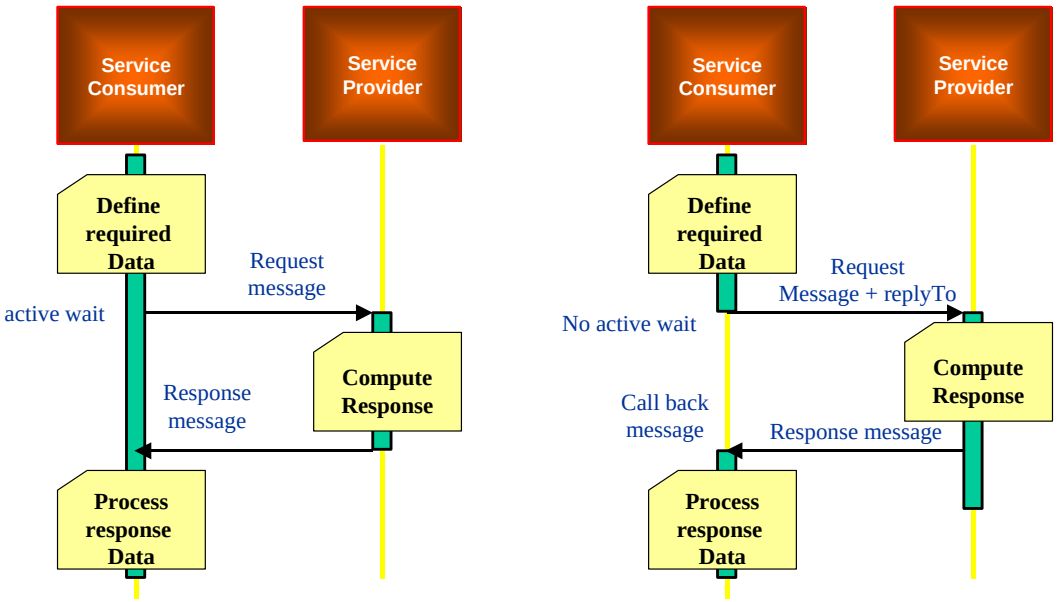

Figure 5 - RPC/MSG Style sequence diagram

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 MTOSI operations are by their nature a 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 asynchronous 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 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 a asynchronous transport

Mapping the RPC style to a 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.

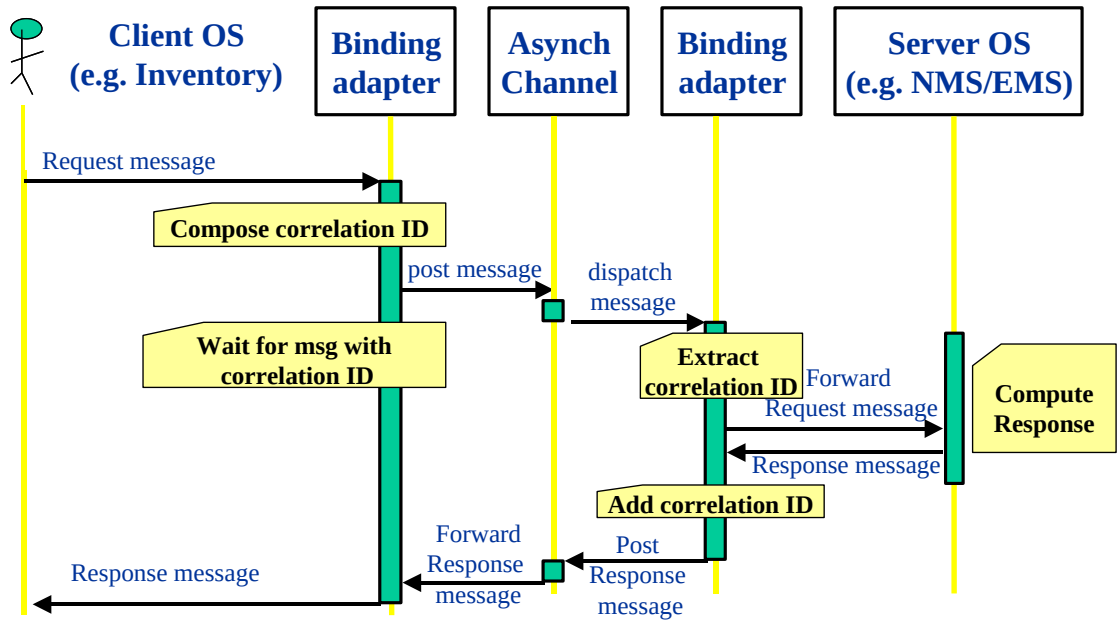

Figure 6 – RPC Request/Reply over Asynchronous transport

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”.

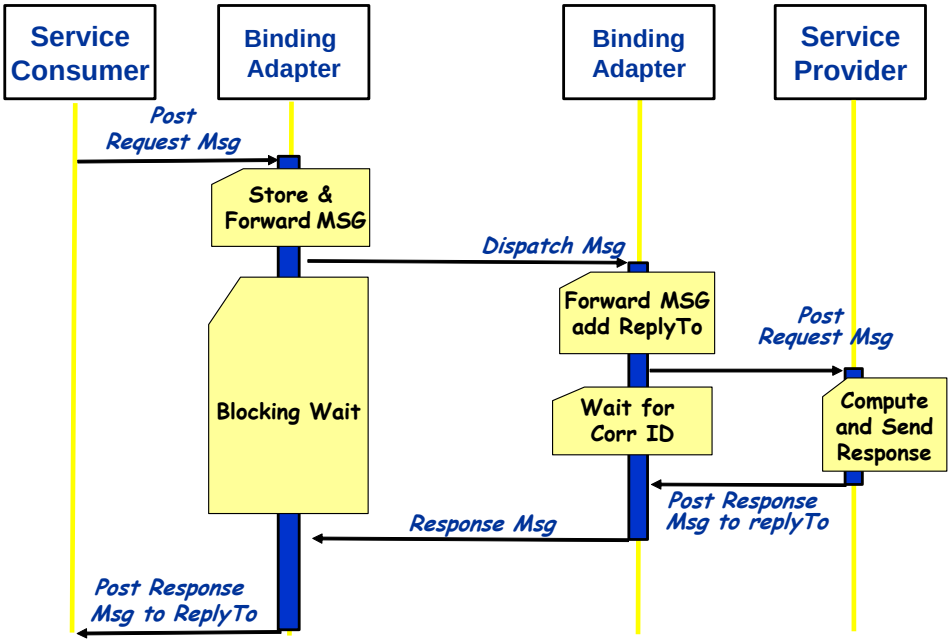

Figure 7 - MSG Request/Reply over Synchronous Transport

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.

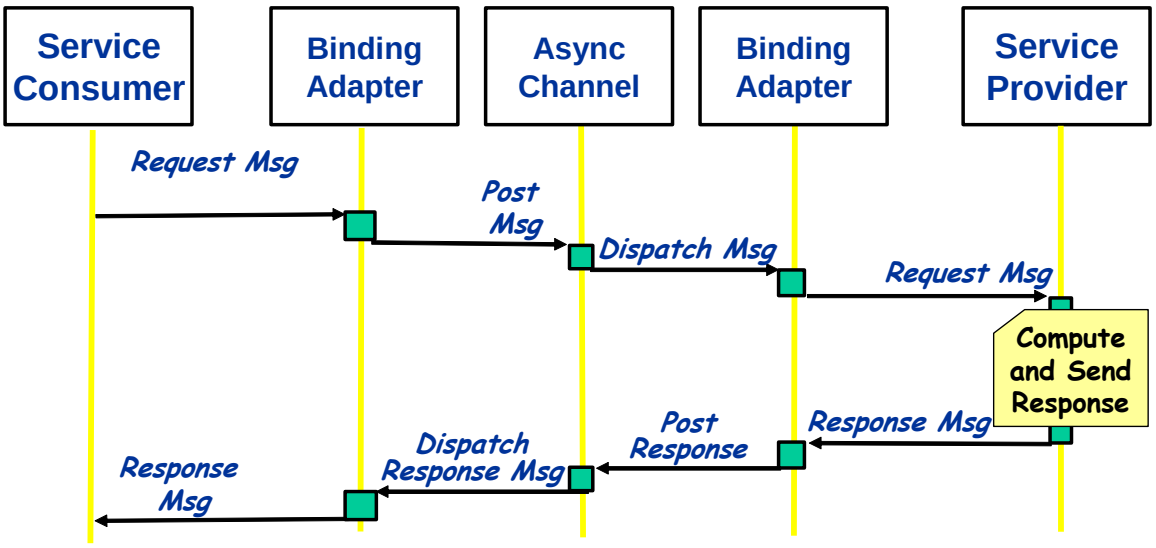

Figure 8 - MSG Request/Reply over Asynchronous Transport

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) summarizes the possible combinations mapping the communication style to a transport

Table 1 Communication Styles and Transport Type

	Transport type	
Communication Style	Synchronous (e.g. HTTP/S, IIOP)	Asynchronous (e.g. JMS, MQ,SMTP)
RPC	Maps natively	Binding adapter needs to handle messages correlation (correlation ID)
MSG	Binding adapter needs to handle ***store and forward*** 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 8 possible combinations of communication styles and communication patterns into individual MEPs.

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

Table 2 - MEPs used in MTOSI

Message Exchange Pattern (MEP)	Communication Pattern			
Communication Style	**Simple Response**	**Multiple Response**	**(File) Bulk Response**	**Notification**
RPC (Synch)	**SRR** Synchronous Request/Reply	**SIT** Synchronous Iterator	**SFB** Synchronous (File) Bulk	**SN** Notification with sync subscription
MSG (Asynch)	**ARR** Asynchronous Request/Reply	**ABR** Asynchronous Batch Response	**AFB** Asynchronous (File) Bulk	**AN** Notification with async subscription

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).
- Synchronous Notification (SN) and Asynchronous Notification (AN) – these MEPs facilitate the dissemination of notifications (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 = getTerminationPoint(in)*** operation defined in the ManagedResourceInventory DDP TerminationPointRetrieval service interface will have two variants:

- ***SRR*** ***out = getTerminationPoint(in)***

- ***ARR*** ***out getTerminationPoint(in + replyTo+ CorrelationID)***

3.2 Multiple response communication pattern (SIT, ABR)

Handling a large result data set or dealing with intermediate results 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. In MTOSI V2 the "Multiple Batch response communication pattern" has been renamed and extended to go beyond the batching of result sets and address multiple replies due to intermediate results. In order to minimize backward compatibility issues the approach taken keeps the MEP names and WSDL/XSD unchanged and repurposes the patterns to accommodate the new requirements.

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.

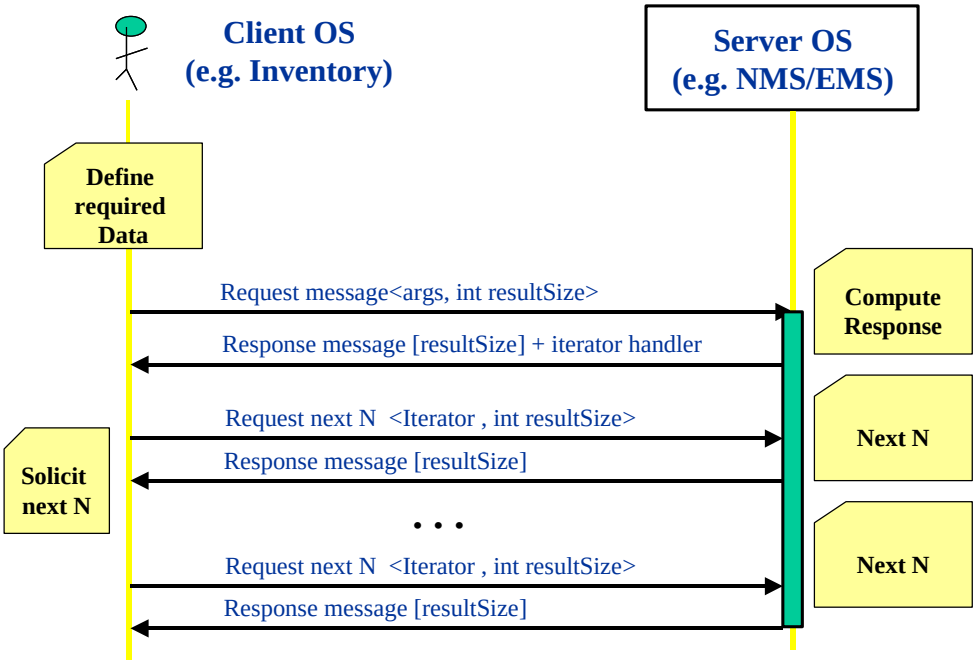

Figure 9 - Synchronous Iterator design pattern

The Iterator state on the server side can be controlled using a timeouts-based garbage collection mechanism.

This is functionally equivalent to the mechanism used in the MTNM CORBA interface (TMF814) to retrieve multiple result sets, with the difference that the getNext() is mapped to the getXXXIterator() and there are no explicit methods to control the Iterator lifecycle.

The iterator handler returned by the first invocation is modelled as a URI and has to allow a service consumer to invoke the subsequent getXXXIterator(). Note the actual WS implementing the Iterator can be static (shared across Iterators) or dynamic (dedicated to the particular Iterator).

- In case of a static Iterator implementation, the URI will contain two parts: an endpoint address and an iterator reference argument, e.g. HTTP://www.foo.com/myIteratorEndpoint?IteratorID=1233445676

  The Iterator endpoint will be used by the service consumer to invoke the Iterator WS and the argument (available in the MTOSI header) will be used by the consumer to identify the proper iterator status on the back-end.

- In case of a dynamic iterator implementation, the URI will simply contain an endpoint uniquely dedicated to the Iterator WS.

Note: The URI is opaque to the service consumer and this mechanism allows flexibility on the implementation side.

It is expected that a future release of MTOSI (TIP) will fully adopt the WS-Addressing, and WS-enumeration mechanism to replace the current MTOSI proprietary iteration solution.

The Iterator design pattern mechanism can also be used to carry out a request with multiple replies where the multiple replies are due to the nature of having intermediate results. In this context this pattern is also called “pull mode multiple responses”.

The initial request will have as response typically an ACK (or NACK) with a call-back handler to retrieve the subsequent responses (same as iterator handler).

The client OS can then periodically poll the Server OS in order to retrieve the intermediate results (e.g. Request next N).

The last result polled from the Server OS will have an indicator of “Last” (batchSequenceEndOfReply) set to true.

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 usually has 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].

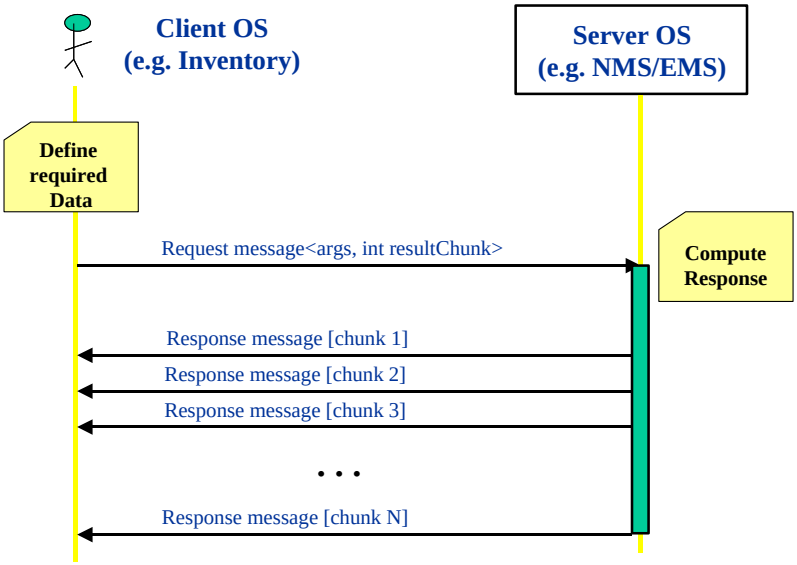

Figure 10 - Asynchronous batch response design pattern

The Asynchronous batch response can also be used to carry out a request with multiple replies where the multiple replies are due to the nature of having intermediate results, some of which are partial successes and some of which are partial failures (not to be confused with an exception). Note that an exception is only used when processing is stopped before the request can be completed. In this context this pattern is also called “push mode multiple responses”.

The initial request will have a “replyToURI” that will be used from the Server OS to post all the responses related to the initial request.

The only difference among the batch response and multiple responses resides in the payload. In the push mode concerning multiple responses, the payload of the messages will carry out the additional intermediate responses.

The last result sent from the Server OS will have an indicator of “Last” (batchSequenceEndOfReply) set to true.

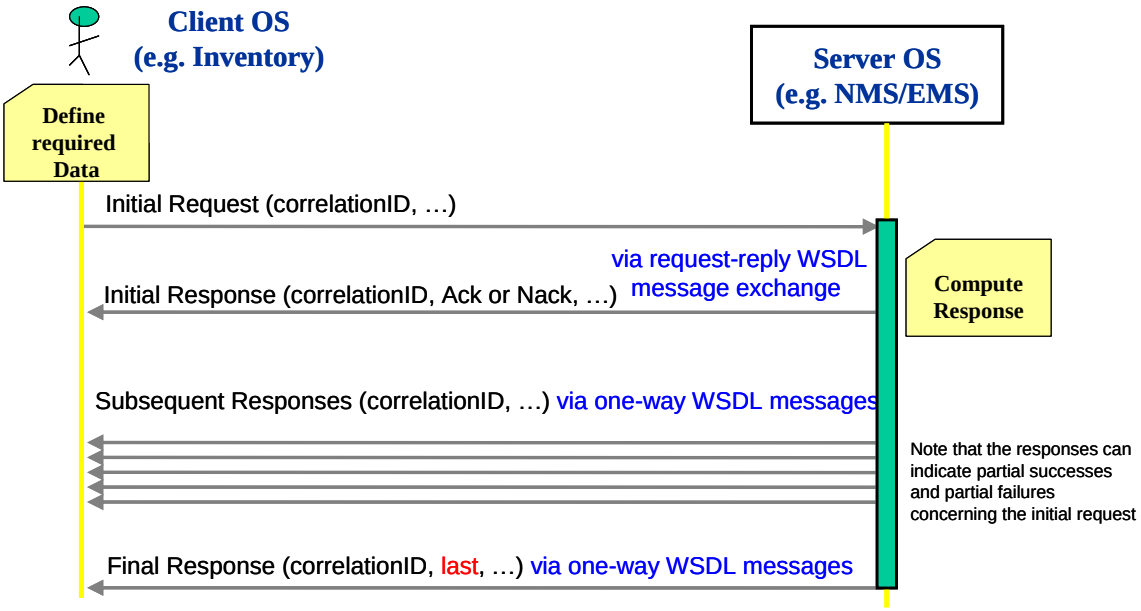

Figure 11 – Realizing asynchronous request multiple replies.

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

For example, the ***out[] = getAllPhysicalTerminationPoints(in)*** operation defined in the ManagedResourceInventory DDP TerminationPointRetrieval service interface will have two variants:

- ***SIT*** ***out[i..j] + iterator = getAllPhysicalTerminationPoints (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 than 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)

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

The Asynchronous Response does not need the additional Iterator interface since the service producer will directly fragment the result and send it to the service consumer, however the client OS will have to implement the oneway operation interface (call-back) to receive the responses back from the Server OS.

Although with slightly different semantics, the formal argument "**requestedBatchSize**" is common to both and can be carried in the message header of the request to convey how many elements to include in the batch. A value of 0 (zero) will imply the entire data result set in a single response.

In the pull mode multiple responses mode the "requestedBatchSize" will not be used.

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 the transfer of 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 out-of-band in both the communication style variants.

3.3.1 Synchronous (File) Bulk Response (SFB) MEP

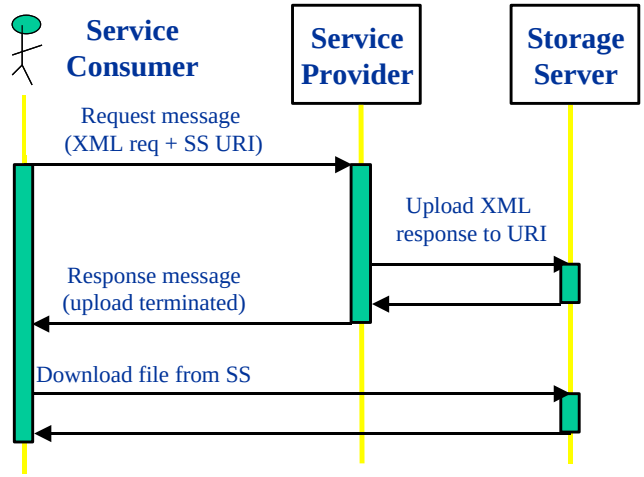

Figure 12 - Bulk transfer RPC style

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

3.3.2 Asynchronous (File) Bulk Response (AFB) MEP

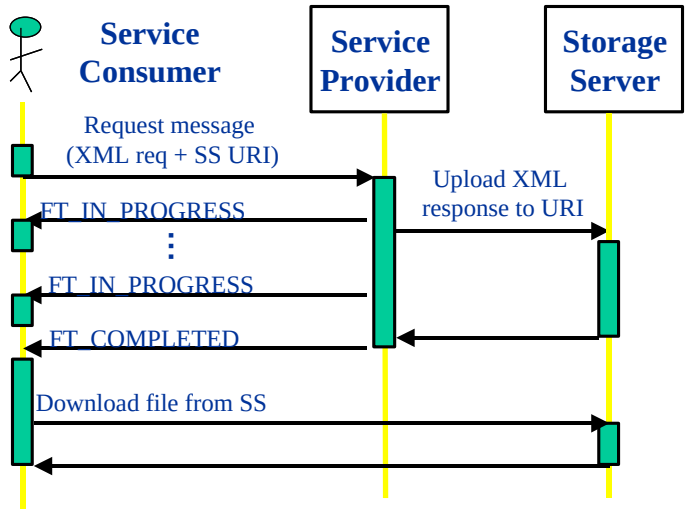

Figure 13 - Bulk transfer MSG style

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. 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 recipients (pub/sub), possibly greater than one. MTOSI leverages a subset of factures defined in the Web Service Notification specification [WSN]. The Web Service Notification specification [WSN] V1.3 as of October 11th 2006 has been officially adopted as an OASIS specification. 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 services, this specification is a good candidate for the MTOSI notification mechanism.. 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 MTOSI notifications 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 Notification Communication Patterns

The model describing the notification of events from Producers to Consumers shall account for the following two interaction patterns:

- Direct or Point-to-Point (P2P) notification pattern, where a Notification Consumer is subscribed directly to the Notification Producer.
- Brokered Notification or Publish/Subscribe (Pub/Sub) notification, where the interaction between a Consumer and a Producer is mediated by an intermediary such as a Message Oriented Middleware (MOM).

Note: both patterns allow disseminating information in a multicast fashion or can be restricted to have a single consumer.

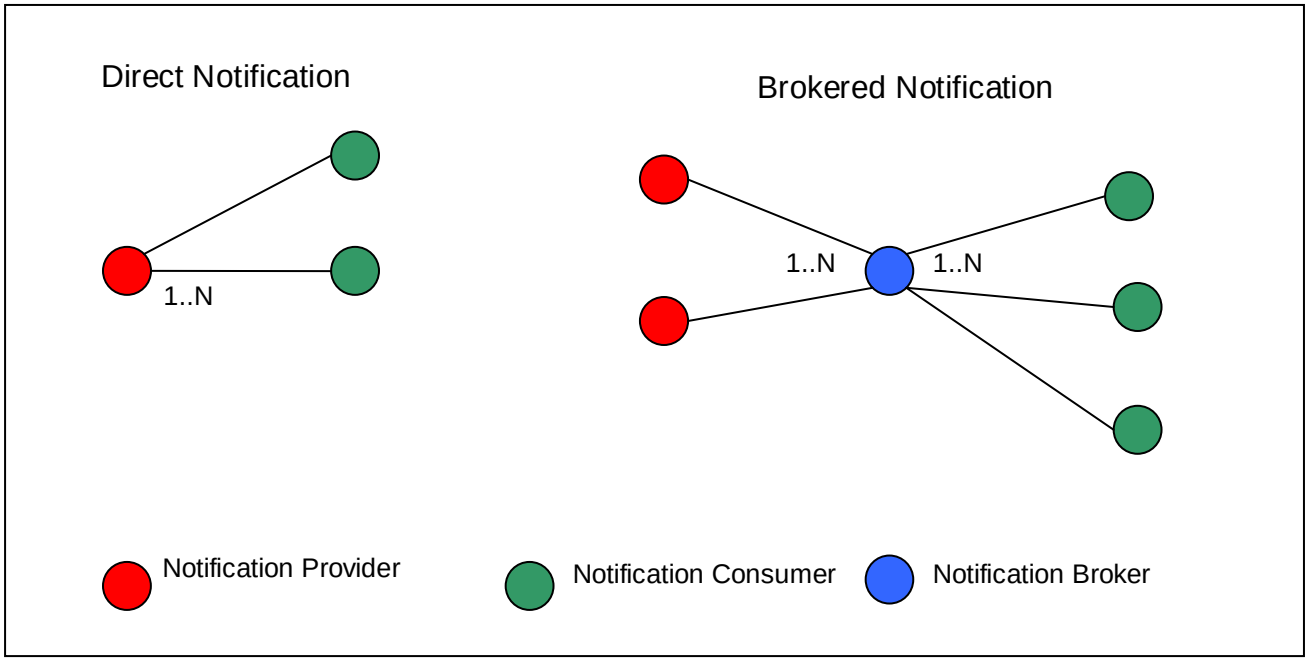

Figure 14 Notification Communication Patterns

3.4.2 MTOSI Notification Interfaces

3.4.2.1 Reference Model and Simplifications

In MTOSI, the definition of the notification service is driven by:

- Focusing on the brokered Pub-Sub notification pattern as it relates well with the JMS transport capabilities. This model was supported as of MTOSI R1.0
- Focusing on the direct notification pattern as it naturally relates with the HTTP transport. This model was supported as of MTOSI R1.1.
- Using a simplification of the WS-Notification standard, in particular of its WS-BrokeredNotification specification with the selection of the Simple Publishing pattern as the reference model
- In MTOSI we can 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.

3.4.2.2 Direct Notification Logical View

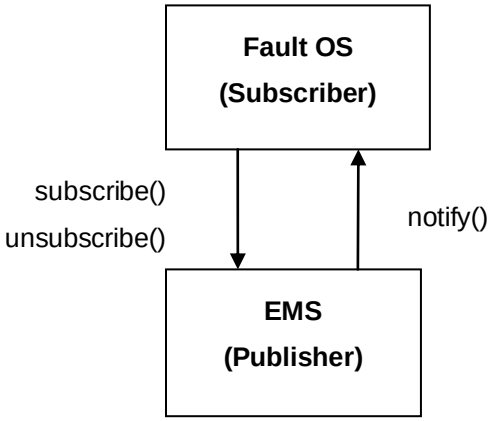

Figure 15 Direct Notification between Fault OS and an EMS

The above figure illustrates a simplified representation of the Direct Notification (P2P) involving one Fault OS and an EMS. In this notification pattern, the communication channels are direct between a Subscriber and a Publisher.

3.4.2.3 Brokered Notification Logical View

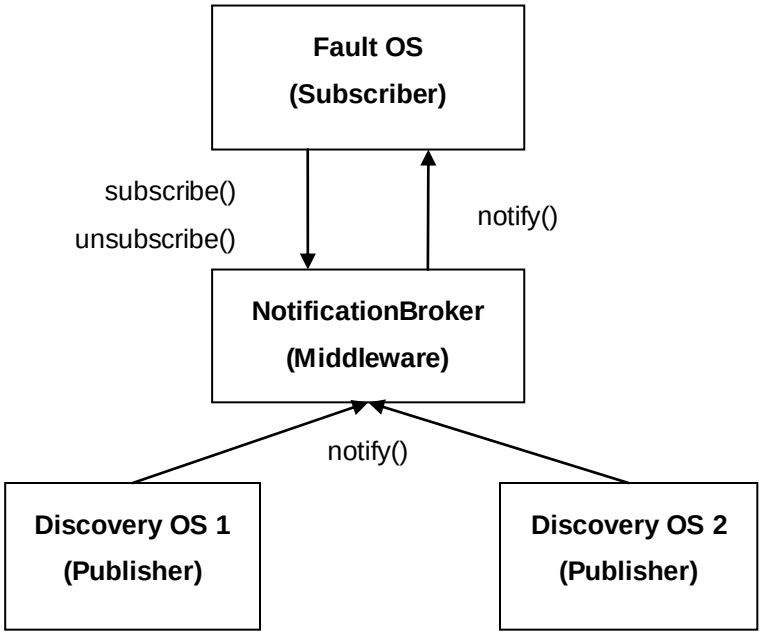

Figure 16 Brokered Alarm Notification between Discovery OS and Fault OS

The above figure illustrates a simplified representation of the Brokered Notification (Pub-Sub) involving one Fault OS, two Discovery OSs, and one NotificationBroker. In this example, the role of the

NotificationBroker is to filter and disseminate to the Fault OS all the alarms produced by the two Discovery OSs. The Fault OS would have initially subscribed to a Fault topic with the NotificationBroker.

Note that this configuration is also applicable to notification interfaces that exist between a Discovery OS and all the EMSs it manages.

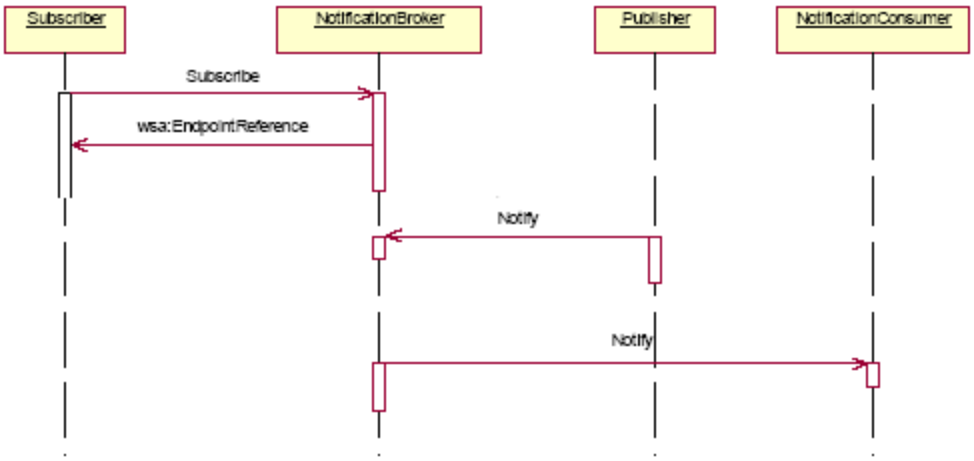

Figure 17 - WS-Notification, brokered publisher

The above figure further simplifies the sequence diagram proposed by WS-notification by removing the subscriber topic lookup in the broker and the Publisher registration with the broker.

3.4.2.4 Notification Topics

3.4.2.4.1 Description

MTOSI topics originate from the WS-Topics specification of the WS-Notification. It is important to understand the following two forms based on the context:

- Topic Types: All the MTOSI notification events are partitioned in three logical categories called MTOSI topics. MTOSI topic types can be defined as an enumeration of the four topic type names. Also, MTOSI topic types can be used in the definition of the topic instances.
- Topic Instances: The deployment of an MTOSI Notification Service (NS) solution relies on the existence of topic instances. The naming definition and implementation of these topic instances are specific to the transport used as the MTOSI CCV. Topic instances are based on the TopicExpression definition, which comes from the WS-Notification. But, it is a simplified to a string in MTOSI . A reference of one of these topic instances is provided by a Subscriber OS on invocation of the NotificationProducer or NotificationBroker subscribe operation.
- Refer to SD2-9 for special binding details of the topic instances with JMS as the MTOSI transport.
- Refer to SD2-16 for special binding details of the topic instances with HTTP as the MTOSI transport.

3.4.2.4.2 MTOSI Topic Types

There are no mandatory topic types in the MTOSI Notification Interface specifications. However, it is recommended that the following topic type structure being used at minimum by any implementation of the MTOSI Notification Interface specifications.

Recommended MTOSI topic types classification of the events :

Inventory Topic

Grouping of the following event notification types;

ObjectCreation, ObjectDeletion, AttributeValueChange, ObjectDiscovery and StateChange

Fault Topic

Grouping of the following event notification type;

Alarm, TCA

Protection Topic

Grouping of the following event notification types;

EquipmentProtectionSwitch, and ProtectionSwitch

The following events are generic and they apply to all MTOSI topics:

- Heartbeat,
- FileTransferStatus
- EventLossOccurred, and
- EventLossCleared

3.4.2.5 MTOSI Notification Filters

3.4.2.5.1 Query Expression

Filtering of the notifications is provided in the *selector* input parameter of a subscription request for a given topic. The *selector* is an instance of a QueryExpression structure, which is comprised of the following two attributes:

- QueryDialectType dialect: This attribute contains the name of the language/type of expression contained in the *query* attribute. The QueryDialectType is an enumerated list of all supported query expression dialects. In MTOSI R1.1, the list includes:
  - ANSI-92 which is the expression used by JMS in message selectors.
  - XPATH1 and XPATH2 which are applicable to WS implementation style.
- String: query: This attribute contains the query expression to be interpreted in the language as indicated in the *dialect* attribute.

3.4.2.5.2 Event Filterable Attributes

The following list contains all the recommended event filterable attributes:

- Event type
- objectName
- objectType
- osTime
- sourceTime
- isEdgePointRelated
- layerRate

- aliasNameList
- probableCause
- probableCauseQualifier
- nativeProbableCause
- perceiveSeverity
- rootCauseAlarmIndication
- acknowledgeIndication

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 Style, 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 flavours. It will be the providers responsibility to state which operations are implemented and offered to the service consumer.

5 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 1.3 October 11th 2006 – 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

6 Administrative Appendix

6.1 Document 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.
1.2	October 2006	Minor editorial changes
1.3	February 2008	Merged SD2-8_MTOSI_Notification_Service.doc. Updating to reflect MTOSI R2.0

6.2 Acknowledgments

First Name	Last Name	Company
Michel	Besson	Cramer > Amdocs OSS Division
Francesco	Caruso	Telcordia Technologies Inc.
Shlomo	Cwang	Cramer > Amdocs OSS Division
Felix	Flemisch	Siemens
Steve	Fratini	Telcordia Technologies Inc.
Elisabetta	Gardelli	Siemens
Jérôme	Magnet	Nortel Networks

6.3 How to comment on this 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.