CONTENTS

FIGURES

INTRODUCTION

The purpose of this document is to define a set of implementation profiles for IEC 61968 using technologies commonly used for enterprise integration. More specifically, this document describes how message payloads defined by parts 3-9 of IEC 61968 are conveyed using web services and the Java Messaging System. Guidance is also provided with respect to the use of Enterprise service Bus (ESB) technologies. The goal is to provide details that would be sufficient to enable implementations of IEC 61968 to be interoperable. In addition, this document is intended to describe integration patterns and methodologies that can be leveraged using current and future integration technologies.

The IEC 61968 series of standards is intended to facilitate *inter-application integration* as opposed to *intra-application integration*. Intra-application integration is aimed at programs in the same application system, usually communicating with each other using middleware that is embedded in their underlying runtime environment, and tends to be optimised for close, real-time, synchronous connections and interactive request/reply or conversation communication models. IEC 61968, by contrast, is intended to support the inter-application integration of a utility enterprise that needs to connect disparate applications that are already built or new (legacy or purchased applications), each supported by dissimilar runtime environments. Therefore, these interface standards are relevant to loosely coupled applications with more heterogeneity in languages, operating systems, protocols and management tools. This series of standards, which are intended to be implemented with middleware services that exchange messages among applications, will complement, not replace utility data warehouses, database gateways, and operational stores.

This standard is based upon the EPRI Technical Report 1018795 and other contributed works.

Implementation Profiles for IEC 61968

1 Scope

1.1 Scope of IEC 61968

The IEC 61968 standard, taken as a whole, defines interfaces for the major elements of an interface architecture for distribution systems within a utility enterprise. Part 1: Interface Architecture and General Recommendations, identifies and establishes requirements for standard interfaces based on an Interface Reference Model (IRM). Parts 3 through 9 of IEC 61968 define interfaces relevant to each of the major business functions described by the Interface Reference Model.

As described in IEC 61968, there are a variety of distributed application components used by the utility to manage electrical distribution networks. These capabilities include monitoring and control of equipment for power delivery, management processes to ensure system reliability, voltage management, demand-side management, outage management, work management, automated mapping, meter reading, meter control and facilities management. This set of standards is limited to the definition of interfaces and is implementation independent. It provides for interoperability among different computer systems, platforms, and programming languages. Methods and technologies used to implement functionality conforming to these interfaces are considered outside of the scope of these standards; only the interface itself is specified in these standards.

1.2 Scope of this Document

This document is Part 100 of the IEC 61968 standard and specifies an implementation profile for the application of the other parts of 61968 using common integration technologies, including JMS and web services. Guidance is also provided with respect to the use of Enterprise Service Bus (ESB) technologies. This provides a means to derive interoperable implementations of IEC 61968 parts 3 through 9. At the same time, this standard can be leveraged beyond information exchanges defined by IEC 61968, such as for the integration of market systems or general enterprise integration.

The following diagram attempts to provide an overview of scope, where IEC 61968 compliant messages are conveyed using web services or JMS. Through the use of an ESB integration layer, the initiator of an information exchange could use web services, where the receiver could use JMS, and vice versa. The integration layer also provides support for one to many information exchanges using publish/subscribe integration patterns and key functionality such as delivery guarantees.

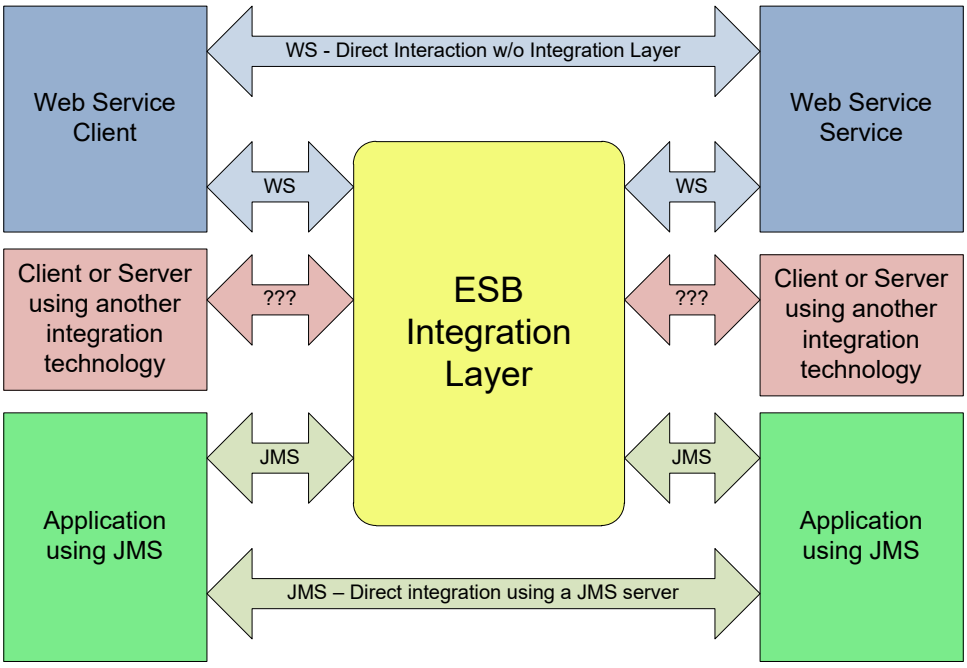


Figure 1 - Overview of Scope

The scope of this document specifically includes the following:

- Integration patterns that support IEC 61968 information exchanges
- Design of interfaces for use of strongly typed web services
- Design of interfaces for use of generically typed web services
- Design of interfaces using JMS
- Definition of standard design artefacts and related templates
- Recognition that technologies other than JMS and web services may be used for integration leveraging this standard (with some specific examples and associated recommendations described in appendices)

This profile can also be applied to integration problems outside the scope of IEC 61968.

It is important to note that other implementation profiles can potentially be defined for IEC 61968, and that this is not intended to be the only possible implementation profile. In addition, this profile can be adapted to meet specific needs of specific integration projects.

It is also not within the scope of this document to prescribe those implementation details as required for security.

2 Normative References

The following documents contain provisions, which through reference in this text, constitute provisions of this International Standard. For dated references, only the editions cited apply. For undated references, the latest edition of the referenced document (including any amendments) applies. Members of IEC and ISO maintain registers of currently valid International Standards.

- IEC 60050-300, *International Electrotechnical Vocabulary (IEV) - Electrical and electronic measurements and measuring instruments - Part 311: General terms relating to measurements - Part 312: General terms relating to electrical measurements - Part 313: Types of electrical measuring instruments - Part 314: Specific terms according to the type of instrument*
- IEC 61970-301 *Energy Management System Application Program Interfaces – Part 301 Common Information Model Base.*
- IEC 61968-11 Common Information Model (CIM) Extensions for Distribution
- IEC 61968-1 *System Interfaces for Distribution Management – Interface Architecture and General Recommendations*
- IEC 61968-2 *Glossary*
- IETF RFC 1738: *Univeral Resource Locators*
- IETF RFC 2616: *Hyper-Text Transport Protocol 1.1*
- IETF RFC 4122: *A Universally Unique Identifier (UUID) URN Namespace*
- ISO-8601: *Representation of dates and times*
- *Extensible Markup Language (XML): http://www.w3.org/TR/REC-xml*
- *XML Schema: http://www.w3.org/XML/Schema*
- *Simple Object Access Protocol (SOAP) 1.2: http://www.w3.org/TR/soap12-part1/*
- WS-I Basic Profile Version 1.0: *http://www.oasis.org*
- JMS API*: JSR-914 Java Message Service (JMS) API*
- IEC 61970-552 *CIM XML Model Exchange Format*
- IEC 62361-100: *Naming and Design Rules for CIM Profiles to XML Schema Mapping*[1]

[1] under consideration

3 Terms, definitions and abbreviations

3.1 Terms and definitions

3.1.1 General

For the purposes of this specification, the terms and definitions given in IEC 60050-300, IEC 61968-2, IEC 62051, IEC 62055-31 and the following terms apply.

Where there is a difference between the definitions in this standard and those contained in other referenced IEC standards, then those defined in 61968-2 shall take precedence over the others listed, and those defined in this document shall take precedence over those defined in 61968-2. This clause includes some definitions taken from Wikipedia.

3.1.2 Enterprise Service Bus

An Enterprise Service Bus (ESB) refers to a software architecture construct that is used as an integration layer. This construct is typically implemented by technologies found in a category of middleware infrastructure products, usually based on recognized standards, which provide foundational services for more complex architectures via an event-driven and standards-based messaging engine (the bus).

An ESB generally provides an abstraction layer on top of an implementation of an enterprise messaging system, which allows integration architects to exploit the value of messaging without writing code. Contrary to the more classical enterprise application integration (EAI) approach of a monolithic stack in a hub and spoke architecture, the foundation of an enterprise service bus is built of base functions broken up into their constituent parts, with distributed deployment where needed, working in harmony as necessary.

An ESB does not implement a service-oriented architecture (SOA) but provides the features with which one may be implemented.

3.1.3 Java Messaging Service (JMS)

The **Java Message Service** (**JMS**) API is a Java Message Oriented Middleware API for sending messages between two or more clients. JMS supports request/reply, publish/subscribe and point to point messaging patterns. JMS is a part of the Java Platform, Enterprise Edition, and is defined by a specification developed under the Java Community Process as JSR 914.It is important to note that some ESB product vendors provide language bindings for JMS using C, C++ and/or C#, making the term JMS a misnomer. Where the wire protocol is different between different JMS implementations it is often trivial to bridge between different JMS implementations.

3.1.4 Service-Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is a computer systems architectural style for creating and using business processes, packaged as *services*, throughout their lifecycle. SOA also defines and provisions the IT infrastructure to allow different applications to exchange data and participate in business processes. These functions are loosely coupled with the operating systems and programming languages underlying the applications. SOA separates functions into distinct units (services), which can be distributed over a network and can be combined and reused to create business applications. These services communicate with each other by passing data from one service to another, or by coordinating an activity between two or more services. SOA

concepts are often seen as built upon and evolving from older concepts of distributed computing and modular programming.

3.1.5 Event-Driven Architecture (EDA)

Event-Driven Architecture (EDA) is a software architecture pattern that promotes the production, detection and consumption of events. An event is any change of state of potential interest. Within an EDA, events are transmitted between loosely coupled software components and services, typically using publish/subscribe messaging patterns. EDA is complementary to SOA, to the extent that SOA 2.0 is also known as ‘event-driven’ SOA. EDA is fundamental to a variety of business intelligence patterns, including complex event processing patterns.

3.1.6 Simple Object Access Protocol (SOAP)

The Simple Object Access Protocol (SOAP) is a standard that defines the formatting of XML messages. SOAP serves as a foundation layer of the web services protocol stack. SOAP is also commonly used within JMS. Common transports for SOAP include HTTP, HTTPS and proprietary JMS transports. SOAP is also now sometimes referred to as ‘Service-Oriented Architecture Protocol’. SOAP is a W3C Recommendation.

Two versions that are in common use include SOAP 1.1 and SOAP 1.2. New integrations or interfaces should use SOAP 1.2 when applicable.

3.1.7 Web Services (WS)

A Web Service is defined by the W3C as ‘a software system designed to support interoperable Machine to Machine interaction over a network.’ Web services are frequently just Web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services.

The W3C Web service definition encompasses many different systems, but in common usage the term refers to clients and servers that communicate using XML messages that follow the SOAP standard. Common in both the field and the terminology is the assumption that there is also a machine readable description of the operations supported by the server written in the Web Services Description Language (WSDL). The latter is not a requirement of a SOAP *endpoint*, but it is a prerequisite for automated client-side code generation in many Java and .Net development tools.

Where web services have two primary styles, document-centric and RPC-centric, the focus of this specification is document-centric. This is supportive of SOA and the transport of IEC 61968 payloads. For increased interoperability, the document wrapped form is required.

It is important to note that web services do not readily support publish/subscribe messaging unless mechanisms such as WS-Eventing are used. However, this specification also describes an approach for the ESB to route messages asynchronously to configured subscribers.

This standard will discuss two approaches for use of web services, where WSDL operations are either generic or strongly typed. Generic web services have WSDLs and related operations that are defined in a payload type independent manner. Strongly typed web services are defined where WSDLs and operations are defined individually for specific payload types.

This also provides for automated frameworks for server-side validation of message content based on message schemas contained in the WSDL document.

3.1.8 Web Services Definition Language (WSDL)

Web Services Definition Language (WSDL) is an XML-based language that is used to describe web services. WSDL is often used in combination with SOAP and XML schema to provide web services over the internet (or an intranet). WSDL is a W3C Recommendation.

Two versions that are in common use include WSDL 1.1 and WSDL 2.0.WSDL 2.0 better supports interoperability between Java and .Net implementations.

The WSDL templates provided by this document are based upon WSDL 1.1, and consequentially WSDL 1.1 is a requirement. In order to better support interoperability between dverse implementations, all WSDL documents are WS-I Basic Profile 1.1 compliant.

3.1.9 XML Schema (XSD)

XML Schema, published as a W3C recommendation in May 2001, is one of several XML schema languages. It was the first separate schema language for XML to achieve Recommendation status by the W3C.

Like all XML schema languages, XML Schema can be used to express a schema: a set of rules to which an XML document must conform in order to be considered 'valid' according to that schema. However, unlike most other schema languages, XML Schema was also designed with the intent that determination of a document's validity would produce a collection of information adhering to specific data types. An XML Schema instance is an XML Schema Definition (XSD) and typically has the filename extension ".xsd". The language itself is sometimes informally referenced as XSD.

It is important to note that 61968 payload are defined using XML schemas. Those XML schemas provide normative specifications for application payloads that are conveyed using this standard.

3.1.10 Representational State Transfer (REST)

Representational State Transfer (REST) is an alternative to the use of SOAP-based web services. REST itself is not currently a standard, but instead an architectural style.

REST leverages HTTP, where each URL is a representation of some object. Interfaces can be defined in terms of XML payloads for requests and responses. A WSDL is not required for the use of REST. Additionally, the XML documents used for requests and responses do not need to be defined using an XSD, although it is common for the XML to be compliant to an XSD.

Within REST, an object can be retrieved (read) using an HTTP GET. Similarly, an HTTP POST is used to create an object, an HTTP PUT is used to modify an object and an HTTP DELETE is used to delete an object.

REST is discussed here for completeness purposes only as it may be encountered by an integration project. However, there are currently no specific recommendations for mappings by this standard. REST may be supported in the future.

3.1.11 Queue

A queue is a construct supported by many messaging products to provide reliable messaging with delivery guarantees. A standard API that includes queue-based messaging models is provided by JMS. AMQP also defines an open protocol for queue based messaging.

3.1.12 Topic

A topic is a construct supported by many messaging products to enable publish/subscribe messaging patterns where there may be potentially many consumers of a message that has been sent to a named topic by a publisher. Topics are commonly used as a destination for event messages. Topics are directly supported by JMS.

3.1.13 Message Destination

A message destination is the target address for a message, whether it be a request, response or an event message. When using JMS, the destination may be a topic or queue. When using HTTP mechanisms such as web services, the destination is specified as a URL.

3.1.14 Request

A request is a message sent from a client (or source) to a server (or target) where a response is expected. The request may be either a query (where data is returned from the target) or a transaction (where data is modified in the target). A request will use verbs such as ‘get’, ‘create’, ‘change’, ‘delete’, ‘cancel’, ‘close’ or ‘execute’.

3.1.15 Response

A response is a message sent as a consequence of a request, typically from the target of the request to the source of the request. Response messages are synonomous with ‘reply’ messages. A response message will use a ‘reply’ verb.

3.1.16 Query

A query is a type of request where the target is expected to return information to the source of the request. The request message for a query will use the ‘get’ verb. This will typically be implemented using a request/reply pattern.

3.1.17 Transaction

A transaction is a type of request where the target typically will modify information that it manages. A transaction request will use verbs such as ‘create’, ‘change’, ‘delete’, ‘cancel’, ‘close’ or ‘execute’. This may be implemented using a variety of patterns. If a pattern other than request/reply is used, there should be a delivery guarantee provided by the transport.

3.1.18 Event

The term ‘Event’ is significantly overloaded, and can have different meanings in different contexts. The most general definition of an event is ‘something that happens at a given place and time’, or ‘a change of state of potential interest’. As a consequence of an event, ‘something’ may generate an ‘event message’ to report the fact that a certain type of event occurred at a given time. Event messages are therefore asynchronous in nature and are typically published to potentially interested subscribers using topic-based messaging.

Event messages are one type of asynchronous message. It may be common for an application to generate an event when a condition of interest is detected, or a transaction has been processed. There are several integration patterns that are related to the support of events.

Events are typically published asynchronously as event messages by a system or application in order to report conditions of interest. In some cases a system or application may internally identify a condition of interest, but in other cases the condition may be detected by an external source such as a device. Event messages will use past tense verb such as ‘created’, ‘changed’, ‘deleted’, ‘closed’, ‘canceled’ or ‘executed’.

3.2 **Abbreviations**

The following terms and abbreviations are used within this document:

API	Application Programming Interface
AMQP	Advanced Message Queue Protocol
CIM	Common Information Model
CME	Common Message Envelope
CRUD	Create, Read, Update, Delete
EDA	Event Driven Architecture
ESB	Enterprise Service Bus
IEC	International Electrotechnical Commission
IETF RFC	Internet Engineering Task Force Request For Comments
ISO	International Standards organization
JEE	Java Enterprise Edition
JMS	Java Message Service
JSR	Java Specification Request
mRID	CIM master resource identifier
OASIS	Organization for the Advancement of Structured Information Standards
RDF	Resource Description Framework
REST	REpresentational State Transfer
RFC	Request for Comments
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSL	Secured Socket Layer
TLS	Transport Layer Security
UML	Unified Modelling Language
URL	Uniform Resource Locators
UUID	Universal Unique Identifier
W3C	World-Wide Web Consortium
WS	Web Services
WS-*	Web Services standards
WS-I	Web Services Interoperability
WSDL	Web Services Definition Language
XML	eXtensible Markup Language
XSD	XML Schema
XSL	XML Stylesheet Language

4 Use Cases

4.1 General

The purpose of this clause is to describe several use cases related to the interactions between components within a set of systems cooperating to support a set of business processes. It is important to note that the use cases presented are from the perspective of the integration of systems, as opposed to end use application-level use cases. The actors for the use cases described in this clause include the following:

- Client
- Server
- ESB
- Adapter
- Subscriber (an Event Listener)

Three key terms related to messaging are request, reply and event. Within the terminology of IEC 61968, these are reflected in terms of the verbs used to define specific information flows.

Central to the use cases is the assumption that a variety of integration technologies may be used, where the focus of this standard is JMS and web services.

4.2 Simple Request/Reply

The first use case is a simple request/reply between a client and server. This is synonymous with request/response. The initiator of the request is the client, where the requested is processed by the server. The first view of this is the simple view without the use of an ESB. This use case involves one of two cases:

1. A client making a query request to a server, where the server will return a set of objects to the client based upon some filter criteria
2. A client making a transaction request to a server, where a set of objects will be created or modified in some way

Both cases are illustrated by the following diagram.

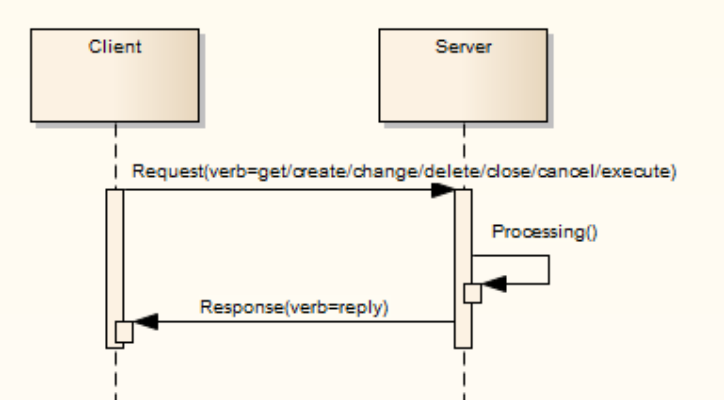


Figure 2 - Simple Request/Reply

In terms of IEC 61968, requests use the verbs such as *get, create, change, delete, close, cancel* or *execute*.

4.3 Request/Reply Using an ESB

The simple request/reply use case can also be extended to leverage an ESB. Within the ESB many actions can be taken by intermediaries as needed to facilitate integration and the decoupling of components, such as transformations and routing.

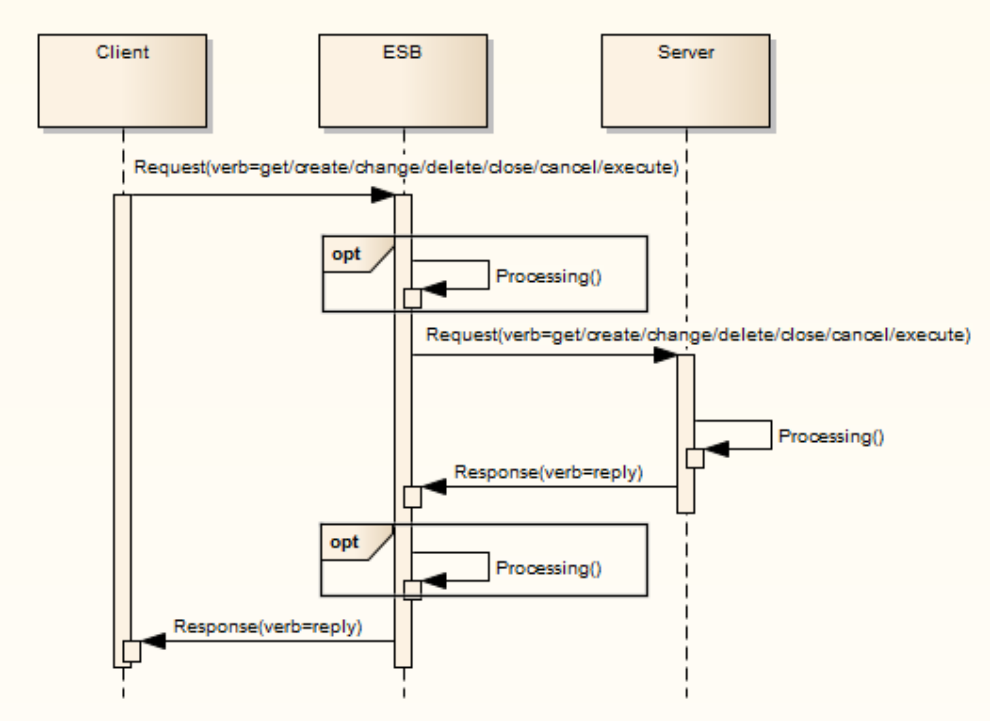


Figure 3 - Request/Reply Using Intermediaries

A key aspect of this use case is the decoupling of the client from the server, so that the client does not have to know the exact location of the server or conform to the exact interface used by the server. The routing and mapping can take place in the integration layer.

It is also recommended that ESB intermediaries be stateless. The use of stateless intermediaries simplifies the implementation of load balancing and high availability.

4.4 Events

There is often the need for client processes to be informed of an event of potential interest. Many client processes can listen (subscribe) for events. One example of this is EndDeviceEvents messages that may be published by a metering system. Other examples include events that report the execution of a control or transaction, such as the creation or update of a work order.

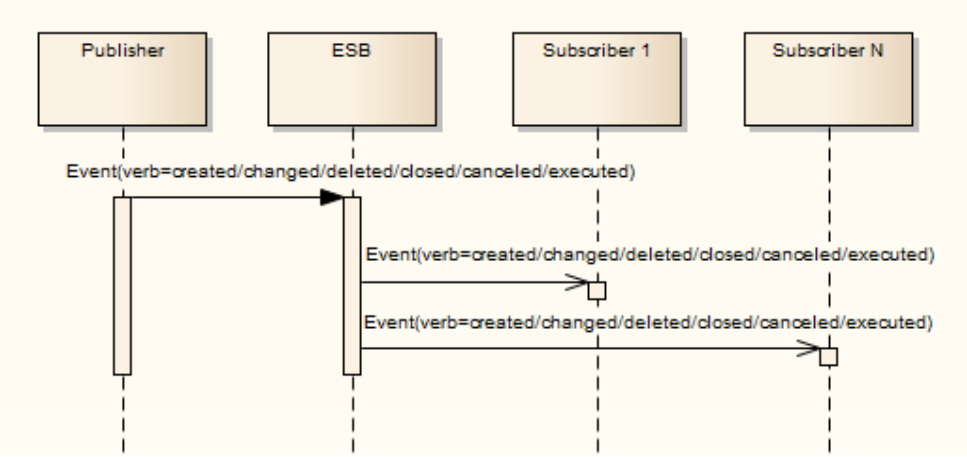


Figure 4 – Events

In terms of IEC 61968, events use the 'past tense' *verbs created, changed, deleted, canceled, closed* or *executed*.

This is an example of a one way pattern, where the sender does not expect a response at the application level, although the transport used may allow for lower level acknowledgements in order to facilitate delivery guarantees when needed.

Listeners that use web services will need to have an exposed interface at a URL that is known by an intermediary so that the events can be appropriately distributed. Rules must also be defined for retry processing where guaranteed delivery is required. The implications of message ordering also need to be considered. Message ordering is a significant topic in itself that may require attention in a future edition.

4.5 Transactions

The use case for a transaction is typically a combination of a request/reply exchange between a client and a server, with a consequential publication of events. An important aspect is that the clients and services may use different transport mechanisms, where a client may use web services but a server may use JMS. An intermediary within the ESB can provide the necessary routing functions.

Transactions are commonly implemented using a request/reply pattern, but they can also be implemented using a point-to-point or one way pattern. The one way pattern can use either present tense verbs for transactions or past-tense verbs for events and the needs of a specific implementation may require.

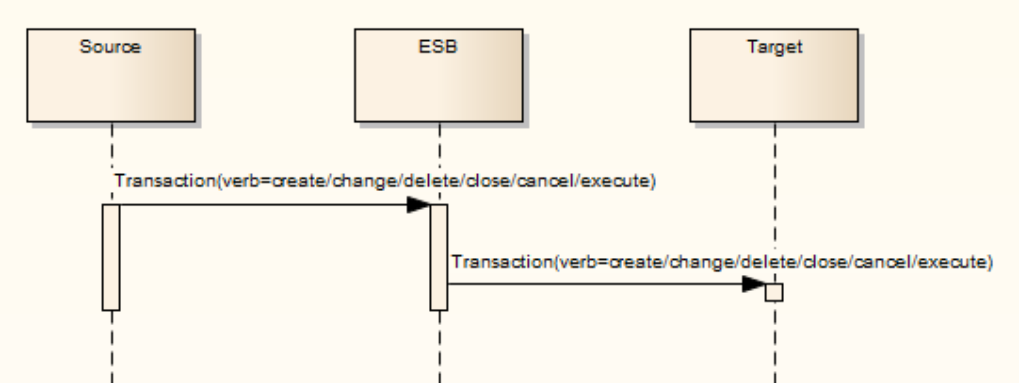


Figure 5 - Point-to-Point (One Way) Pattern

The one way pattern is used in cases where a system of record wants to forward a transaction to a peer system that also must apply the transaction. If errors are detected by the target, they must be logged for resolution within the target system. The following figure provides an example where a transaction will result in events that get propagated to interest subscribers.

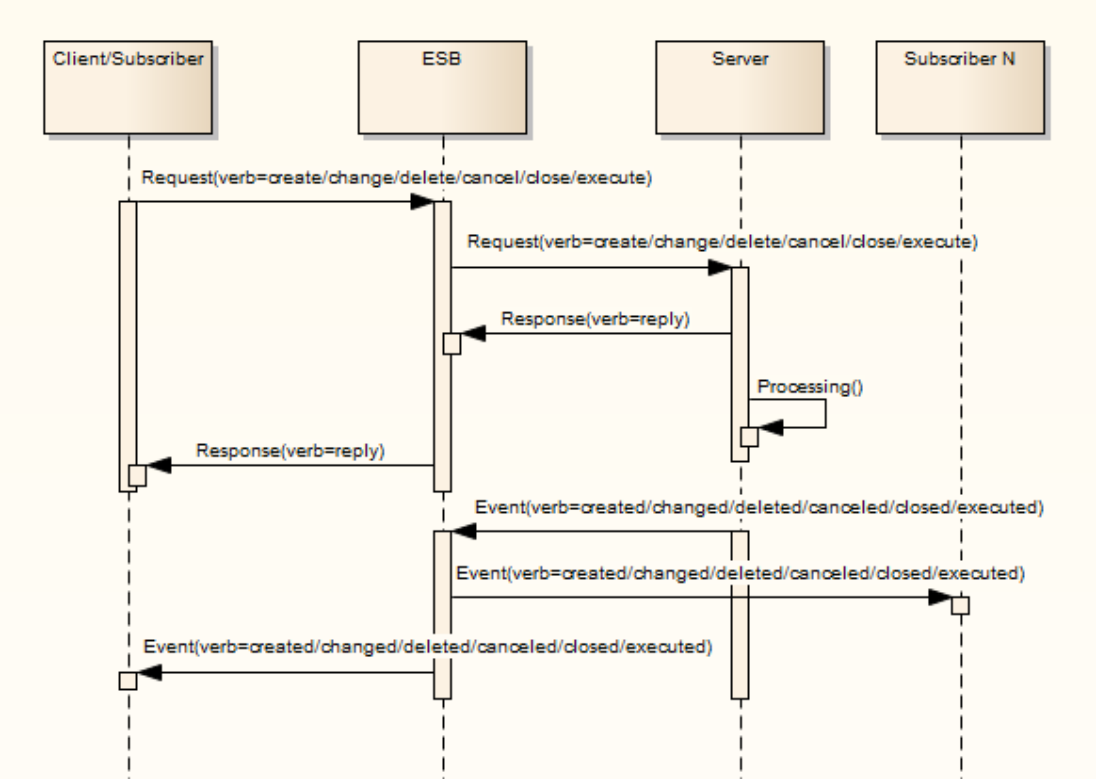


Figure 6 – Transaction Example

In terms of IEC 61968, requests related to transactions use the verbs create, change, delete, cancel, close, and execute. Verbs related to events use past tense. The use of the execute verb implies a complex transaction and the use of the Payload.OperationSet element, as described in section 6.9.

4.6 Callback

A callback is an asynchronous process for message exchange. It is made of two request/response (initial and final) synchronous calls. The two are correlated in a way that each party can unambiguously identify which callback goes with which initial

request. In this case, the client sends an initial request to server. Once the server receives the message, it returns a response message back. At this point the initial message transaction is completed and client application is freed to perform other processing. Once the server has completed processing, it then invokes the final request/response sequence with a request message. The whole call-back process is completed after the client (of the initial request) replies to the final request. This is illustrated in the following sequence diagram.

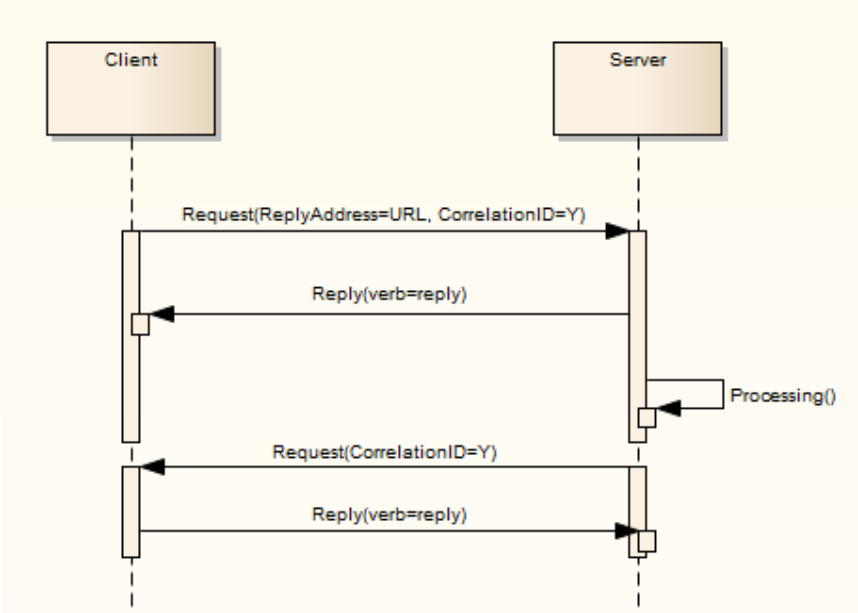


Figure 7 – Callbacks

In the call back process, the client has to inform the server of the location for the final response, often a URL called a callback address. This piece of information is included in the initial request message. Section 6.3.2 discusses specific message elements that are used to control this type of dialog.

Callbacks are typically implemented through asynchronous replies when using JMS.

4.7 Adapters

There are cases where an application (client or server) cannot directly connect to an ESB or another application. In these cases, an adapter can be used to handle the 'impedance mismatch' between the application and the ESB. In some cases the application may simply be a database or file directory. In the following diagram an adapter is used to connect a server to the ESB.

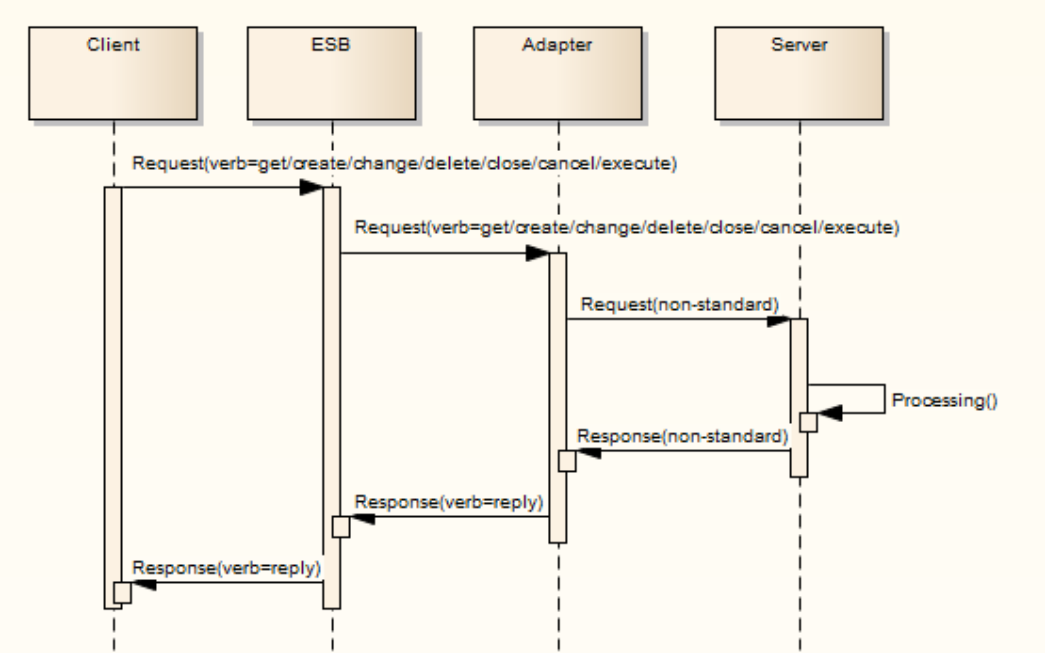


Figure 8 - Use of Adapters

The adapter may perform a variety of functions, and may take actions on behalf of the application such as generation of events that are consequential to the success of a transaction. The most common activity is to convert data between an application model and an enterprise canonical model.

4.8 Complex Messaging

There are some use cases that may require more complex messaging patterns. For example, there may be cases where a transaction may result in potentially many consequential events that can be sent to the client in the form of asynchronous events.

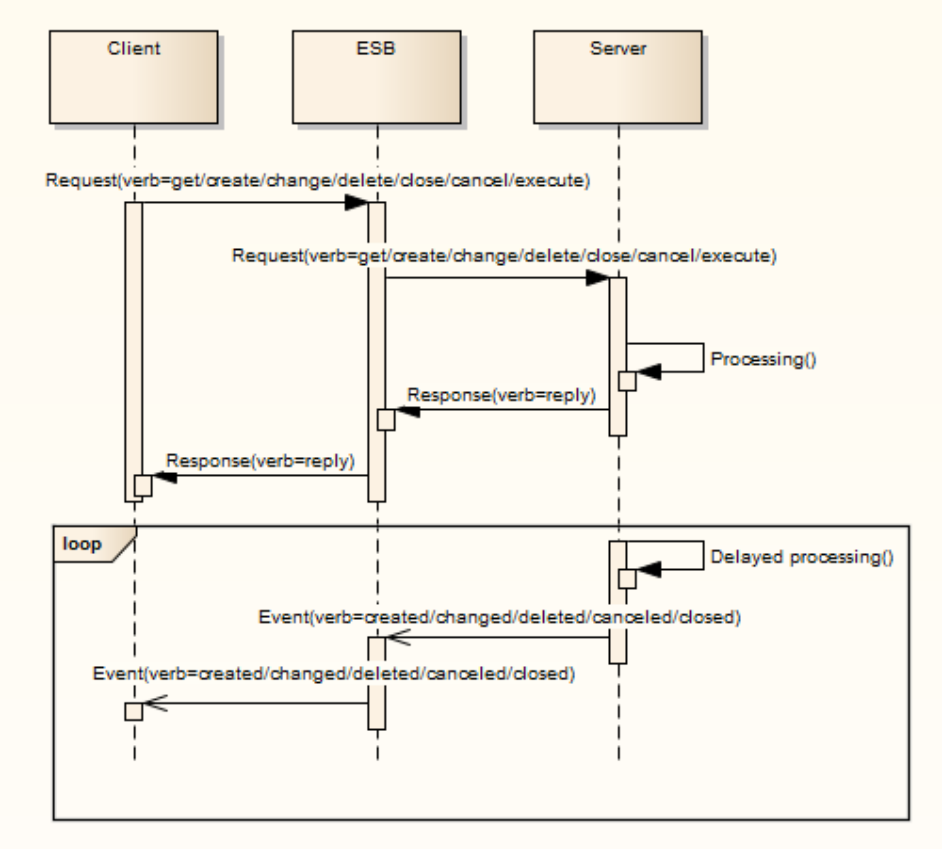


Figure 9 - Complex Messaging

The above use case is seen in applications such as metering, where it may take significant time to obtain results as in the case of a disconnect or load control, where an EndDeviceControls message might be issued, but the results may be reported after some delay using an EndDeviceEvents message.

It is important to note that this is a variation of the callback pattern. The initial request could either be a query, where results are returned asynchronously, or a transaction where one or more events could be generated as a consequence.

4.9 Orchestration

Use cases related to the orchestration or choreography of a business process as well as short or long running distributed transactions are outside the scope of this specification. The integration approaches described by this document can be leveraged by more complex integrations that address those needs.

4.10 Application-Level Use Cases

The use cases described in the previous sub-clauses can be applied to end use application-level use cases. In the following example application-level use case, messages are defined using IEC 61968 verbs and nouns, in the form '<verb>(<Noun>)'. Specific (or representative) types of systems are also identified as opposed to more generic types of actors.

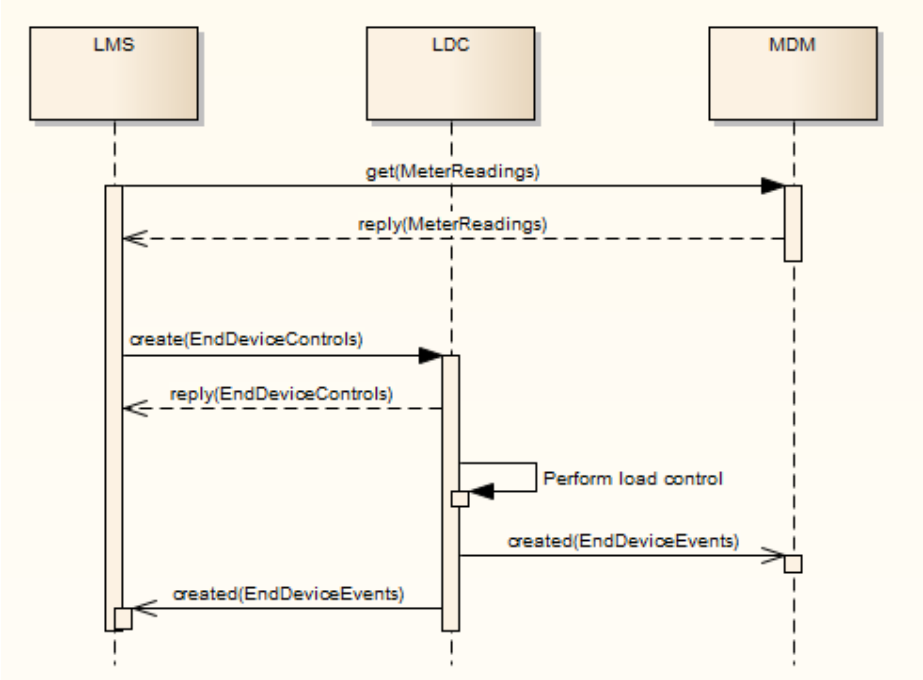


Figure 10 - Application-Level Use Case Example

5 Integration Patterns

5.1 General

This document recognizes a set of basic service integration patterns that support the previously described use cases. It also allows for more complex integration patterns through the use of intermediaries within an enterprise server bus (ESB), or where an application serves an an intermediary.

5.2 Client and Server Perspectives

5.2.1 General

From the perspective clients and servers there are several basic integration patterns described within this document. These patterns include, but are certainly not limited to:

- Synchronous request/reply
  - Web Service implementations use an operation with input and output messages
  - JMS implementations exchange messages on queues
- Asynchronous request/reply
  - Web Service implementations use separate operations for request and callback reply messages
  - JMS implementations exchange messages on queues
- Publish/subscribe, where potentially many targets are listening for messages
  - Web Service implementations will involve a client listening on specified URLs for events that are sent to multiple targets by an ESB intermediary
  - JMS implementations will involve targets listening for messages on a topic

From the perspective of either a client alone or a server alone, whether or not the client is communicating with the server directly, or through intermediaries is irrelevant so long as they have chosen to use the same transport mechanism. The value of an ESB is that an integration architecture can be provided where a client has a choice of using web services, topics or queues, and each target service can also choose to use web services, topics or queues independently from the choices of any client. This can be used to isolate the technology choices of applications in the enterprise such that any bridging technology used has no direct impact on individual applications which would limit application migration options.

5.2.2 Basic Web Service Pattern

The following diagram illustrates a basic request/reply pattern using web services.

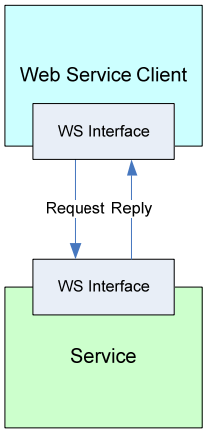


Figure 11 - Basic Request/Reply using Web Services

In this pattern the client issues a request to a web service interface exposed by some service. This interface is defined using a WSDL. However, this interaction pattern may also be realized using REST. The client should expect one of several outcomes:

- The request is successfully processed, where a reply message is returned in a timely manner (Result=OK)
- The request is accepted, but results in a reply message that returns an application level error code (Result=FAILED)
- The request is accepted, but results in a reply message with a partial set of results (Result=PARTIAL)
- The request results in a fault being returned to the client
- After sending the request, no reply or fault is returned in a timely manner

It is also to note that there may be varying levels of security that may be required for the implementation. The extreme case is where the client is using a public network to communicate with the service, where authentication, authorization, encryption and signing may be important.

In the case where a reply is not required, this is called a ‘one way’ pattern. However care should be taken with respect to any required delivery guarantees.

5.2.3 Basic JMS Request/Reply Pattern

The following diagram describes a basic request/reply pattern where the client sends a JMS message to a topic (or queue). In this pattern the reply message is optional, where there may be some cases where a reply message is never sent.

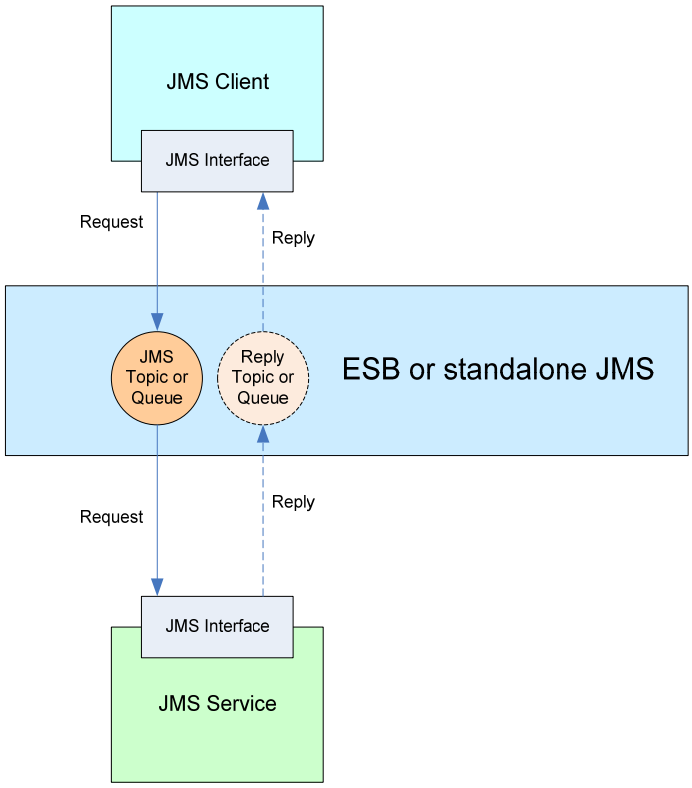


Figure 12 - Basic Request/Reply using JMS

A service listening on a message destination (topic or queue) consumes the request message and issues a reply to the client. The message destination is managed by the JMS implementation (typically part of an ESB). The client may consume the reply message synchronously or asynchronously, with the decision being left solely to the discretion of the client implementation.

Where this document refers to a JMS ‘message destination’, implementations may use either topics or queues for request/reply messaging. When using topics, a service would typically use a durable subscription in cases where request messages must not be lost.

The client initiating the request can expect one of the following results:

- The request message is successfully sent to a topic (or queue), where a reply message (correlated to the request using an ID) is returned in a timely manner (Result=OK)
- The request message is sent, but results in a reply message that returns an application level error code (Result=FAILED)
- The request is accepted, but results in a reply message with a partial set of results (Result=PARTIAL)
- The attempt to send a request message to the topic (or queue) fails
- After sending the request message, no reply message is ever received (which may or may not be normal behaviour depending upon the service)

Reply messages are sent using topics or queues as appropriate, where the specific topics may be statically or dynamically defined. For simplicity subsequent diagrams will not explicitly identify reply topics or queues. The delivery guarantees offered by

JMS implementations provide the option for clients to making transaction requests to not require replies, where this is sometimes referred to as a one-way pattern.

JMS is typically used within a secure, private enterprise network. However there may be isolated cases where this is not the case and security is a significant concern. JMS can also be readily configured to use SSL/TLS and/or use client authentication.

This pattern actually does not require a full ESB implementation, where only a JMS implementation is actually needed. It is also important to note that JMS implementations from different vendors are typically not interoperable, and a 'bridge' may be required in cases where clients cannot use a common JMS implementation.

5.2.4 Event Listeners

Another integration pattern is that of a process that listens for events that may be published. There are many cases where a service may publish event messages that are of potential interest to many other processes.

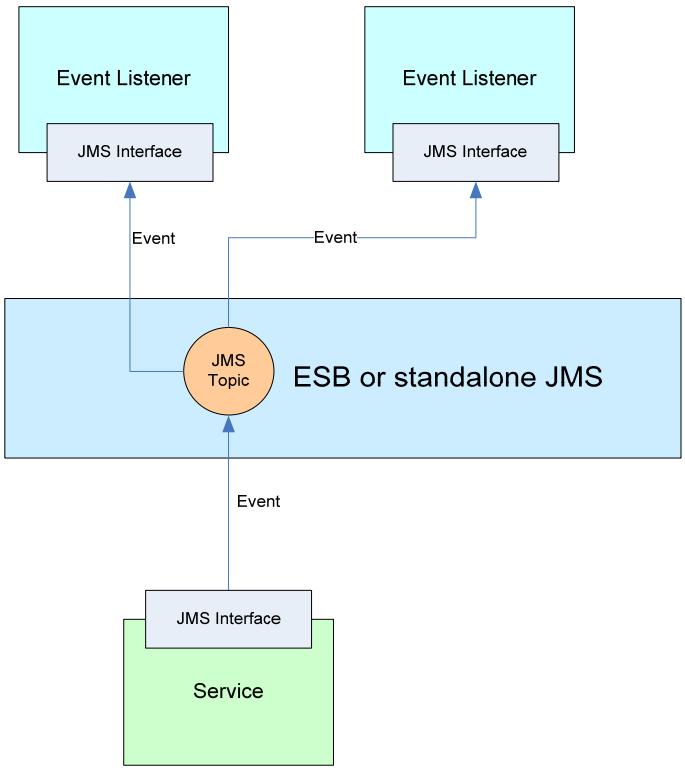


Figure 13- Event Listeners using JMS

Events should typically be published on topics (as opposed to queues), as there will typically be many consumers of events. The listener is responsible to subscribe to one or more JMS topics of potential interest. Some listeners may choose to use a durable subscription in cases where events must not be lost. Event messages are sent and consumer asynchronously. There is no acknowledgement message of any kind returned to the service.

There may be issues related to the number of topics (or queues) and the fan out (i.e. number of event listeners) that can be supported using a given JMS implementation.

It is also possible to extend the architecture described by this document to leverage the use of WS-Eventing for the publication and subscription of events by web service clients. There is currently an open source of WS-Eventing through the Apache project. However, this document will not focus on specific aspects of the implementation and use of WS-Eventing. This document also provides another example of routing events using web services.

5.2.5 Asynchronous Request/Reply Pattern

The asynchronous request/reply allows one or more replies to be returned to a requesting client asynchronously. This is a complex integration pattern that is slightly more complex to implement using web services than for JMS.

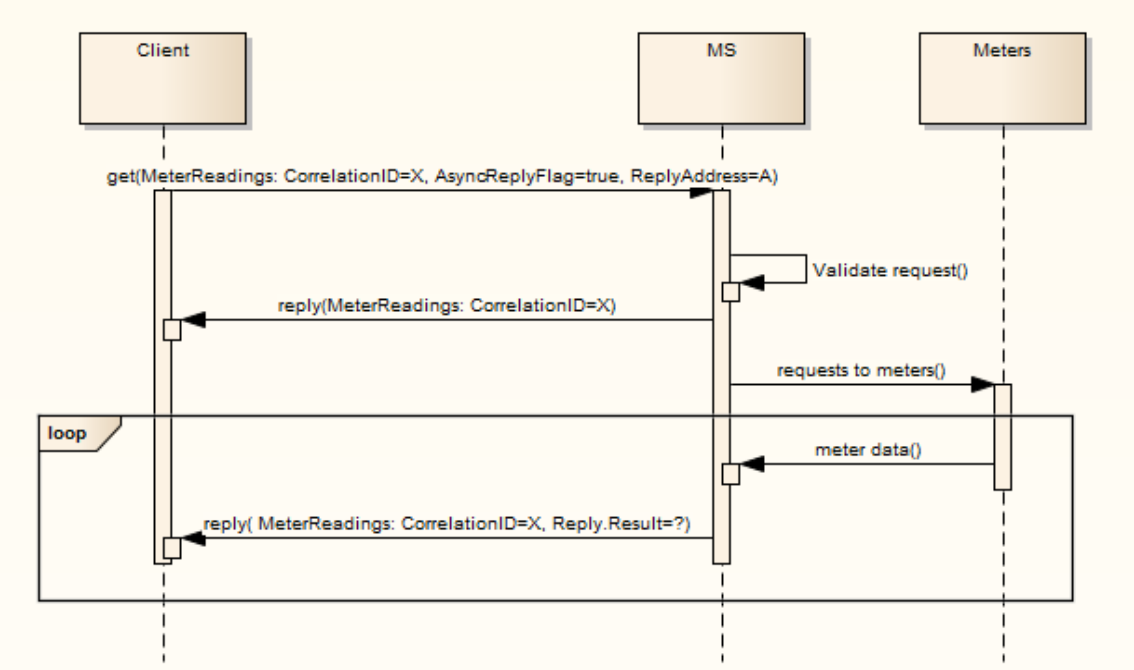


Figure 14 - Asynchronous Request/Reply Pattern

In this pattern, a client makes a request to a service. Where the service may not be able to process and/or respond with the desired information immediately, the service responds immediately with a trivial acknowledgement of the request. After processing is able to occur and/or the desired information is obtained, the service can provide one or more asynchronous replies to the client using a specified URL, topic or queue. The key elements (as described in section 6) used to control this exchange include:

- Header.CorrelationID is used to logically link all messages together. Where a CorrelationID is provided by the client on the initial request, the server must include it on all related response and event messages.
- Header.AsyncReplyFlag is set to ‘true’ on the initial request.
- Header.ReplyAddress is set on the initial request to identify the destination where replies should be set.
- Reply.Result is used on responses, where ‘PARTIAL’ indicates that more responses may be expected, or ‘OK’ indicates that processing is complete and

no more responses should be expected. A value of ‘FAILED’ indicates an error condition.

One use of this pattern is to obtain meter readings, where a metering system head end must request the desired information from one or more meters.

5.3 Bus Perspective

5.3.1 General

Clients and servers can either communicate directly, or through intermediaries. The enterprise service bus (ESB) is used for the management of intermediaries. The use of an ESB provides for many variations in communication patterns. However, the client still sees the ESB as being no different than a server. Similarly, the server sees the ESB as being no different than any other client.

This is important in that:

- No knowledge of the ESB is imposed upon any client or server implementation of an IEC 61968 interface
- There is no requirement for an ESB product placed upon any IEC 61968 interface, except that a JMS server is needed where JMS messages are to be used
- A project implementation using IEC 61968 can use an ESB and freely implement integration patterns that are appropriate for the project and the associated integrations

Given that an ESB is not required for the implementation of an IEC 61968 interface, the remainder of this sub-clause describes recommendations only, and is provided as informative material.

5.3.2 ESB Messaging Pattern Using JMS

The basic ESB messaging pattern using JMS introduces the option for routing of requests. This serves to further decouple the client and server, where the bus (through use of routing logic, often referred to as a ‘Content-Based Router’ integration pattern [EIP]) can make decisions related to the handling of the request. This is described in the following diagram.

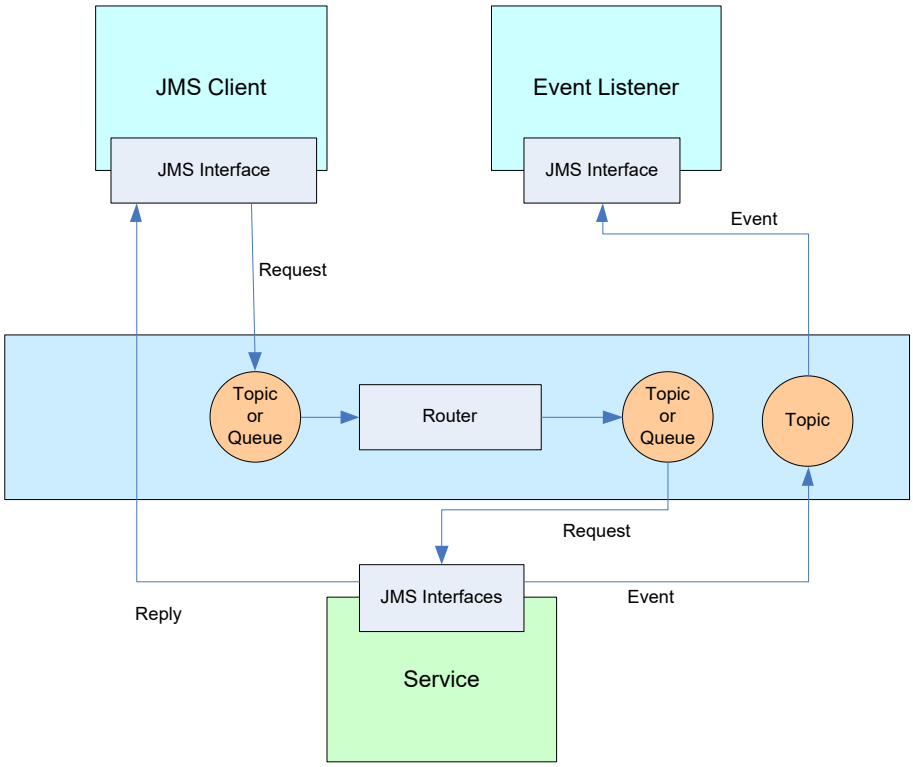


Figure 15 - ESB Content-Based Routing

When a client issues a request to a topic (or queue), a router on the bus can decide to forward the message to another topic (or queue). The decisions by the router may take into account any of the following:

- Contents of a message header
- Contents of a message payload (although this should be avoided)
- The status of a destination service instance
- The need to balance load

It is important to note that one advantage of the loosely coupled approach described by this document is that routing components are not tied to messages of specific payload types. The router can be configured using XPath expressions to identify message content to determine actual routing.

5.3.3 ESB Messaging Patterns Using Web Service Request

The following diagram extends the previously described pattern to permit a request to be initiated by a web service client as well as a JMS client.

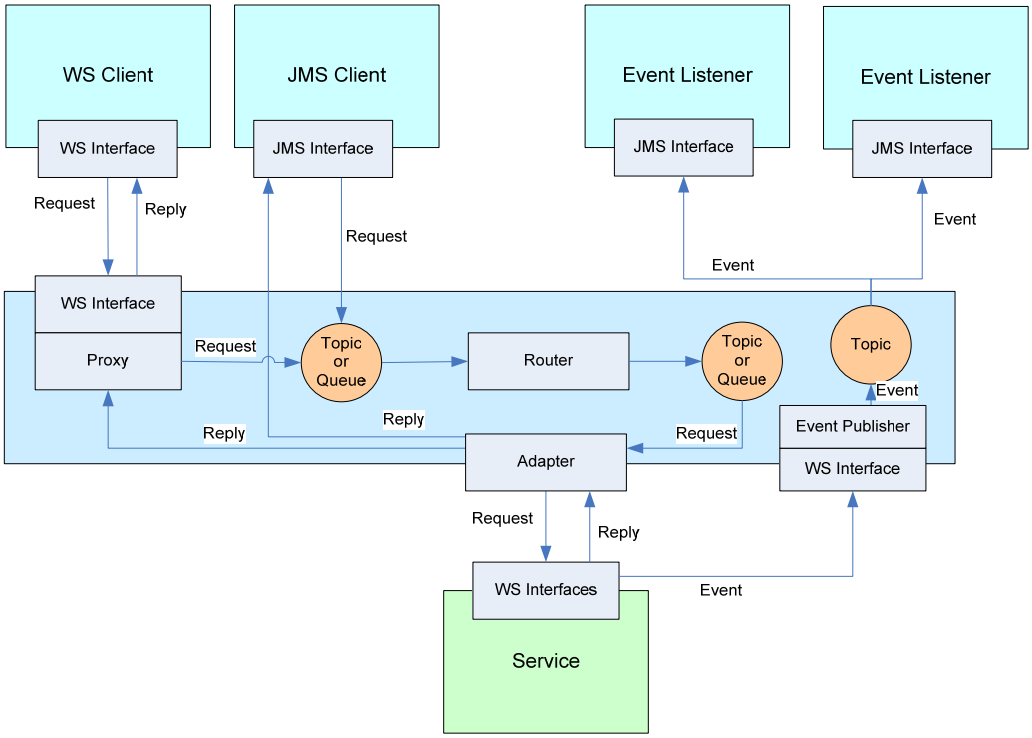


Figure 16 - ESB with Smart Proxy and Content-Based Routing

A proxy (sometimes referred to as a 'Smart Proxy' integration pattern [EIP]) component is implemented on the ESB to expose a web service (as defined by a WSDL). The Smart Proxy can make decisions with respect to dispatching of requests and correlation of responses. The message conveyed through the WSDL is simply converted to a JMS message and is then routed as appropriate.

5.3.4 ESB Request Handling to Web Service

The following diagram extends the previous pattern to allow for a service to expose its interface as a web service with an appropriately defined WSDL.

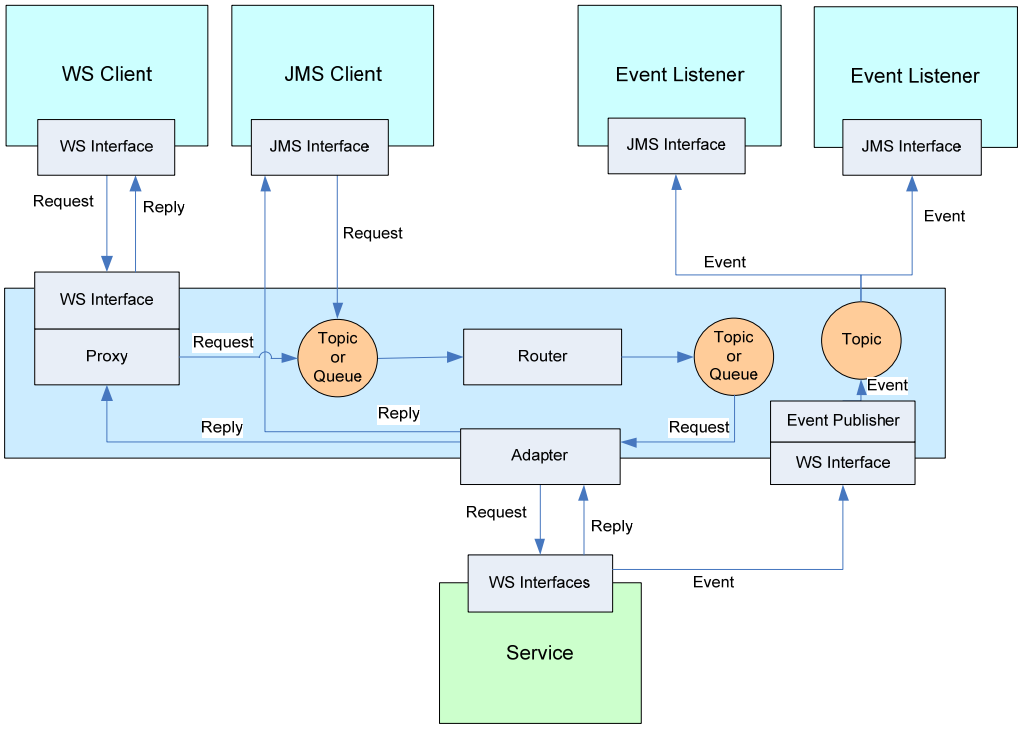


Figure 17 - ESB with Proxies, Routers and Adapters

Within this pattern, an adapter is implemented within the ESB to convert the internal JMS message to an appropriate web service request.

5.3.5 ESB Request Handling via Adapter

The following diagram is a variation on the previous integration pattern, where the server uses an interface that would otherwise not be compliant with the interface profile described by this document. This shows that an IEC 61968 compliant interface can be used to integrate with a server, database, file system or other data source or sink that is otherwise not compliant with IEC 61968 through the use of an adapter within the ESB. Adapters may also be independent of an ESB.

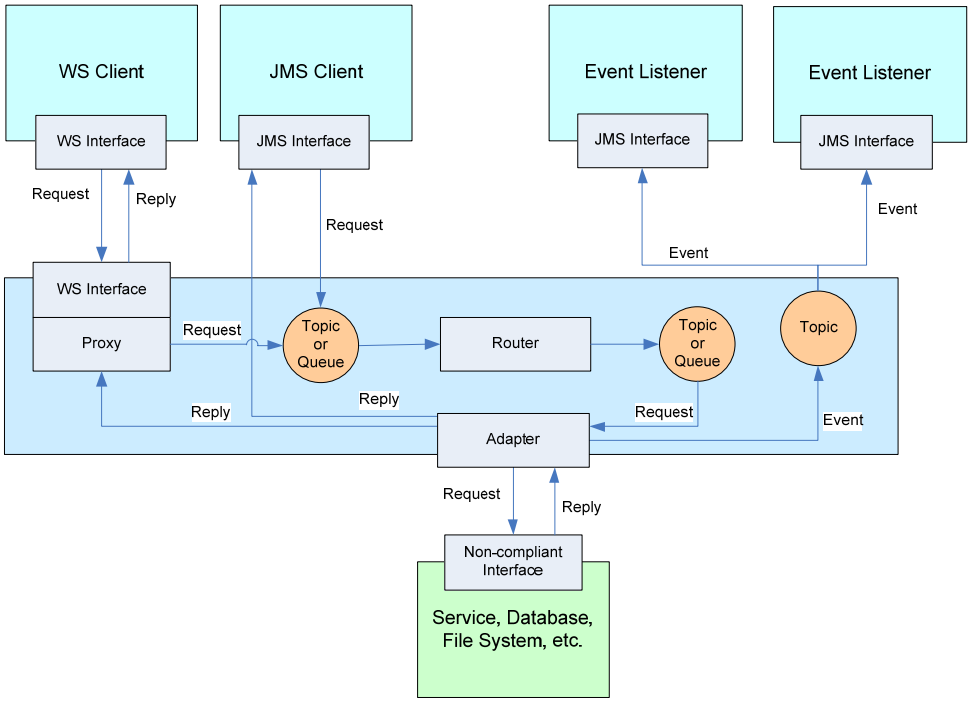


Figure 18 - ESB Integration to Non-Compliant Resources

The integration between the adapter and non-compliant interface can use a variety of integration mechanisms depending upon the capabilities of the specific ESB product. These mechanisms can include, but are not limited to:

- JMS
- Web services
- HTTP
- Java Database Connectivity (JDBC)
- File Transfer Protocol (FTP)
- File read and/or writes
- Proprietary database access

Where the specification of JMS and JDBC imply the use of a Java Enterprise Edition (JEE) framework, it does not impose an actual requirement. Most databases that support JBDC can also be accessed using Open Database Connectivity (ODBC), either directly or through bridge products. Many ESB products that support JMS also have APIs that can be used to send and receive JMS messages using languages other than Java (e.g. C, C++). However, it is important to recognize that the use of the JEE framework provides for a high degree of platform independence.

5.3.6 Custom Integration Patterns

Typically an integration project will involve the implementation of a variety of custom integration patterns. The previous sub-clauses alluded to the potential existence and use of some of these patterns implemented as intermediary processes within the ESB. These would potentially include, but not be limited to patterns such as [EIP]:

- ***Content-Based Router***, where messages are routed based upon message content typically referenced using XPath expressions

- ***Smart Proxy***, where messages may be re-dispatched to a specific destination service, where replies are accepted from the service and passed back to the client

- ***Claim Check***, where a copy of an often very large file is maintained as a document for use by other processes, where the current status of the document is tracked, but the document is typically transported by means other than messaging

- ***Transformation***, where transformations usually defined by XSL are used to reformat message contents

- ***Bridge***, where a message published on a topic or queue may be forwarded to or received from another messaging infrastructure (this pattern can sometimes be implemented using third party products or simply through configuration)

However, it is important to note that this standard does not mandate the implementation or use of any specific custom integration pattern. It is also important to note two primary philosophies for the implementation of integration patterns:

1. Patterns are implemented as a single process definition that may be instantiated one or more times to support potentially many information flows, where there are no type constraints

2. Patterns are templates, where the template is used to implement a process to support a specific information flow, resulting in a ‘type-specific’ implementation

There are significant trade-offs with the above two philosophies. The focus of this specification is to recommend the first option through the use of a common message envelope that readily supports leveraging common implementations of specific integration patterns as opposed to type-specific instances of a given integration pattern.

6 Message Organization

6.1 General

Each service interface is constructed to accept a message that has a verb and a noun. The noun identifies the type of the payload that may be provided on the request, response or event message. This allows the interfaces to be loosely coupled.

The service interfaces are defined using one or both of the following:

- Web Services Definition Language (WSDL), where request, response and fault messages are defined for one or more operations
- JMS message definition

In all cases, XML Schemas (XSDs) are used to define the structure of message envelopes. In most cases, XSDs are used to define the structure of message payloads. The content of message payloads is described in section 7.

6.2 IEC 61968 Messages

6.2.1 General

IEC 61968-1 prescribes information exchanges in terms of a verb, noun and payload. The following diagram shows the directional flow of messages between clients, servers and the ESB based upon the verb.

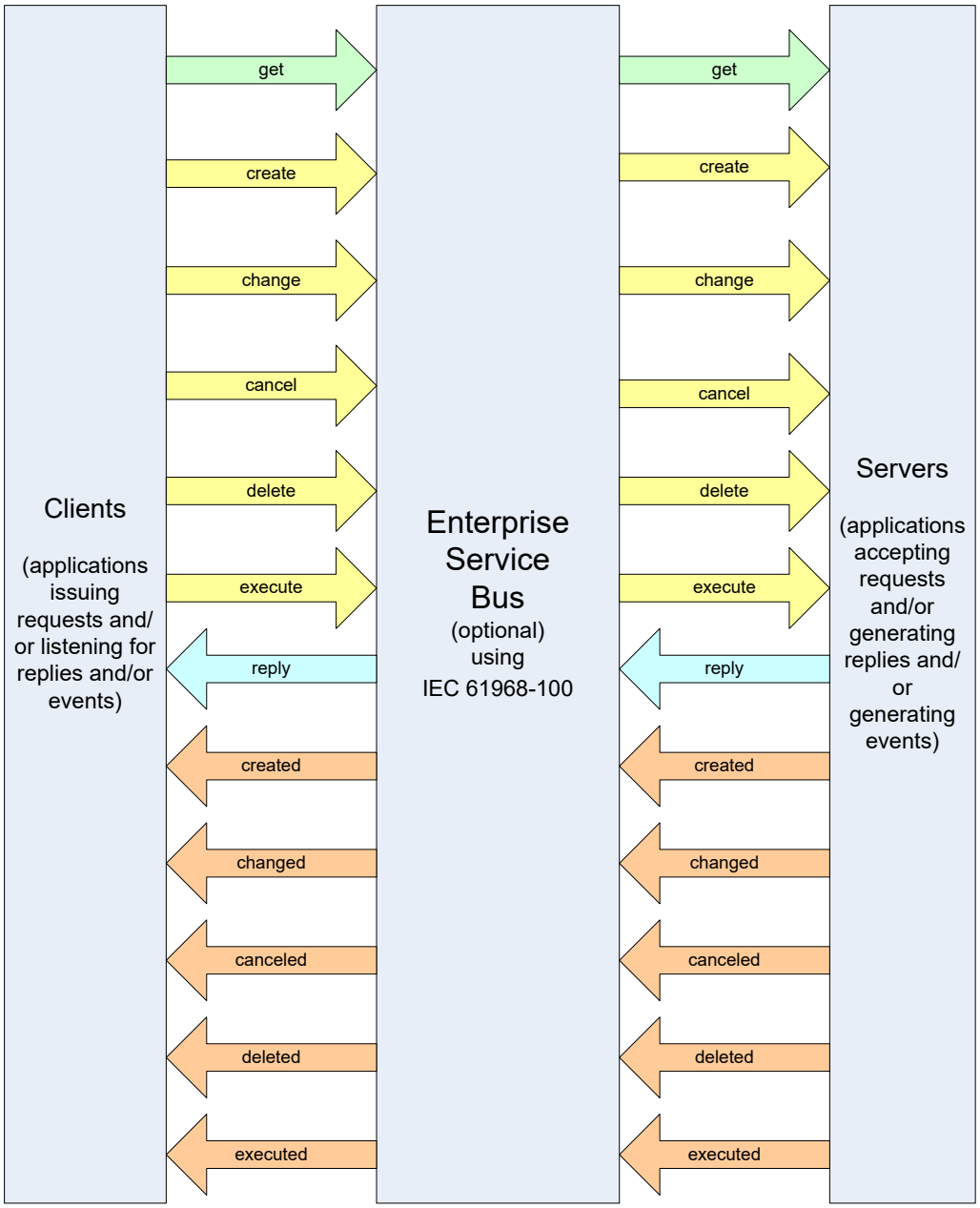


Figure 19 - Messaging Between Clients, Servers and an ESB

6.2.2 Verbs

IEC 61968-1 identifies a set of verbs, where annex B of this standard defines a normative list. This sub-clause is to provide more specificity on the usage of each verb and identify the deprecation some verbs as well as synonyms. In the following table verbs used for requests are associated with the verb that should be used on a response message and as would be used for publication of an event, where often events are a consequence of the successful completion of a transaction initiated by a request.

Request Verb	Reply Verb	Event Verb	Usage
get	reply	(none)	query

create	reply	created	transaction
change	reply	changed	transaction
cancel	reply	canceled	transaction
close	reply	closed	transaction
delete	reply	deleted	transaction
execute	reply	executed	transaction

Figure 20 – Verbs and their Usage

The usage of request verbs are as follows.

- ‘get’ is used to query for objects of the type specified by the message noun
- ‘create’ is used to create objects of the type specified by the noun
- ‘delete’ is used to delete objects, although sometimes an object is never truly deleted in the target system in order to maintain a history
- ‘close’ and ‘cancel’ imply actions related to business processes, such as the closure of a work order or the cancellation of a control request
- ‘change’ is used to modify objects, but it is important to note that there can be ambiguities that need to be addressed through business rules, especially in the case of complex data sets (e.g. complex data sets typically have N:1 relationships and it is important to be clear when relationships are additive or are to be replaced by an update).
- ‘execute’ is used when a complex transaction is being conveyed using an OperationSet, which potentially contains more than one verb.

The response to each of the above requests uses the ‘reply’ verb. Event verbs are often the consequence of a request, where a ‘create’ may result in the generation of a ‘created’ event. The verbs used for events use the ‘past tense’ form of the associated request verb. There is no requirement that event be initiated through a request, as it may be appropriate for events to be generated independently of any specific request.

Validation and business rules may need to be defined for application of verbs in specific cases. This is in part true in that many rules are beyond the descriptive capabilities of UML and XML Schema.

It is also important to note that the enumerations for verbs in the standard Message XML Schema use the **lower case** form. The uppercase form is otherwise convenient for documentation purposes.

IEC 61968-1 previously identified verbs ‘update’, ‘updated’, ‘show’, ‘subscribe’, ‘unsubscribe’ and ‘publish’, all of which have been deprecated. The reason is that ‘show’ is a synonym for ‘reply’, and the verbs ‘subscribe’, ‘unsubscribe’ and ‘publish’ are functions that are performed within the transport layer (e.g. using JMS).

6.2.3 Nouns

Nouns are used to identify the type of the information being exchanged. These are also commonly called profiles. Each noun typically has a corresponding XML Schema definition defined using a namespace unique to each noun. Nouns are typically identified by use cases. Within a message, the noun is used to identify the type of the payload or the type of object to be acted or has been acted upon. Some common example nouns taken from IEC 61968-9 are:

- EndDeviceControls
- EndDeviceEvents
- MeterReadings

Nouns can be defined as needed to distinguish the contents of different information flows. They need not be defined as classes in a UML model, but instead the contents and structure of the noun are defined using classes, attributes and relationships from a UML model.

6.2.4 Payloads

Each noun identifies a payload structure that is typically conveyed using an XML document that conforms to an XML Schema. The structure of the payload is typically defined as a contextual profile from a UML model. This is the approach taken to define message structures by IEC 61968-9.

The following is an example payload structure that results from the contextual profile definition:

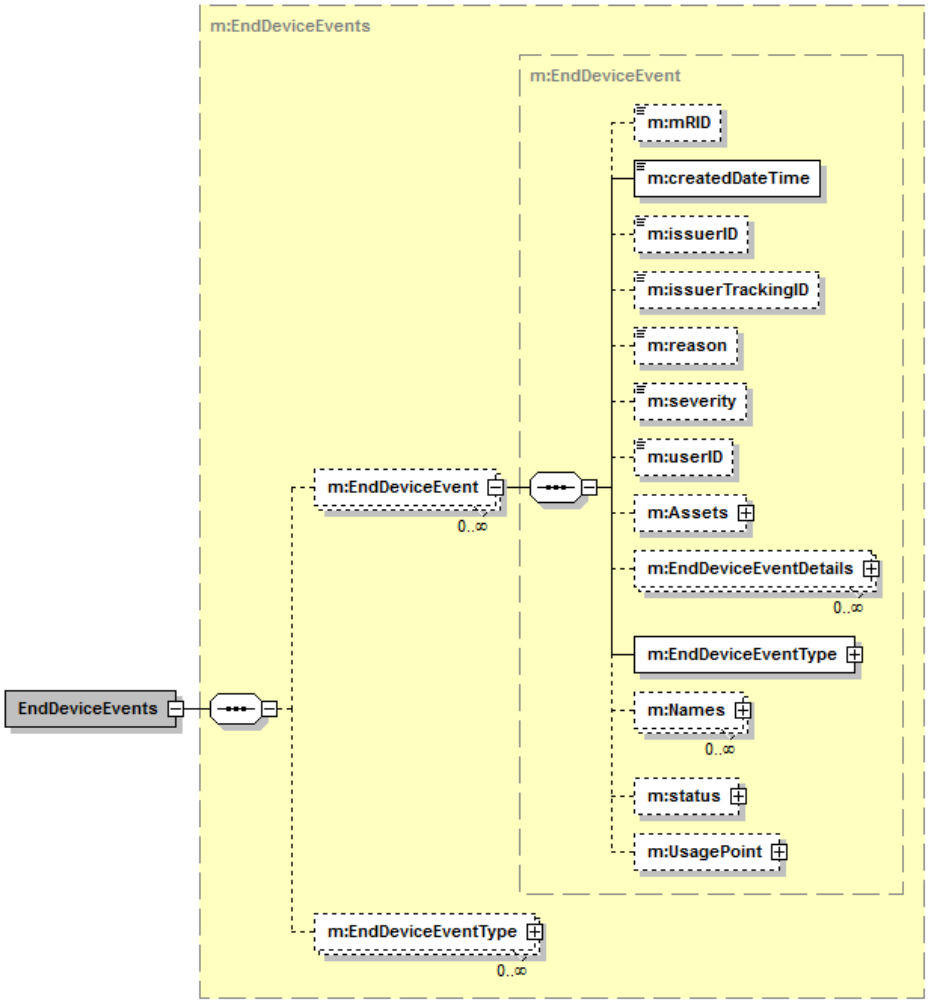


Figure 21 - Example Payload Schema

Depending upon the situation, a payload may or may not be required in a message. A message payload is required for the following cases:

- When issuing a ‘create’ request
- When issuing an ‘change’ request
- When issuing an ‘execute’ request
- In a reply message for a successful ‘get’ request
- For any event message (‘created’, ‘changed’, ‘deleted’, ‘closed’, ‘canceled’ or ‘executed’)

In cases where an event is a consequence of a transactional request and the noun of the request is the same as the noun of the event, the payload of the request is copied as the payload of the event.

A message payload should not be used in the following cases:

- In a reply message for an unsuccessful ‘get’ request in which no results are returned.
- In a reply to a ‘create’, ‘change’, ‘delete’, ‘close’, ‘cancel’ or ‘execute’ request
- In a ‘delete’, ‘close’ or ‘cancel’ request, since the ID of the object(s) is specified using the Request.ID elements
- In a ‘get’ request, as the parameters used to filter the request are supplied in the message Request element, optionally using a ‘Get’ profile in the Request.any element.

6.3 Common Message Envelope

6.3.1 General

Unless otherwise specified, all messages use a common message envelope (CME), where a predefined stereotype is used for requests and another stereotype is used for responses. There are also stereotypes for events and faults. This structure is based upon the IEC 61968-1 recommendations. Messages are constructed with several sections, including:

- ***Header:*** Required for all messages (except for fault response messages), using a common structure for all service interfaces.

- ***Request:*** optional, defining commonly used parameters needed to qualify ‘get’ query requests, or identify specific objects for ‘delete’, ‘cancel’ or ‘close’ requests. There is a provision to allow for inclusion of a complex structure using the Payload.any element. As an example, in the case of a request to get MeterReadings, a ‘GetMeterReadings’ profile can be defined to pass request qualifiers. In cases such as this, the profile should be named using the convention ‘Get<Noun>’. Not used for event or response messages.

- ***Reply:*** Required only for response messages to indicate success, failure and error details. Not used for request or event messages.

- ***Payload:*** Used to convey message information as a consequence of the ‘Verb’ and ‘Noun’ combination in the message Header. Required for ‘create’, ‘change’ and ‘execute’ requests. It is also required for event messages. Optional in other cases as described later in this document and specifically within annex B. The payload structure provides options for payload compression.

The following diagram provides a generalize view of the high-level message structure:

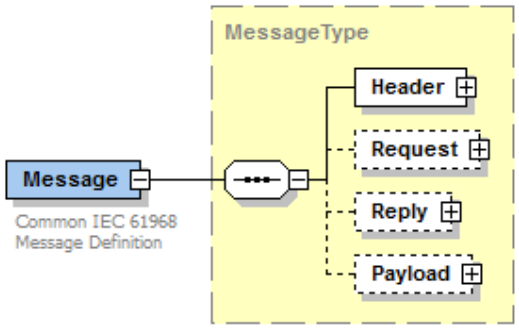


Figure 22 - Common Message Envelope

From this common message envelope there are four stereotypes which identify the specific subset of elements that are used for requests, responses, events and faults:

- RequestMessage
- ResponseMessage
- EventMessage
- FaultMessage

Where these stereotypes are useful when defining interfaces to clearly differentiate requests from responses from events from faults, it is a common practice to internally use the more generic Message structure within software such as common services and intermediaries.

6.3.2 Message Header Structure

Common to request, response and event messages is a header structure. The header currently has two required fields that must be populated, these include:

- Verb, to identify a specific action to be taken. There are an enumerated set of valid verbs, where commonly used values include ‘get’, ‘create’, ‘change’, ‘cancel’, ‘close’, ‘execute’ and ‘reply’. Within event notification messages ‘past tense’ verbs are used, which can include ‘created’, ‘changed’, ‘canceled’, ‘closed’ and ‘executed’. Implementations should treat deprecated verbs ‘update’ and ‘updated’ as synonyms to ‘change’ and ‘changed’.

- Noun: to identify the subject of the action and/or the type of the payload, such as MeterReadings, Notification, etc.

Field that can be optionally supplied include the following:

- Revision: To indicate the revision of the message definition. This should be ‘1’ by default.

- ReplayDetection: This is a complex element with a timestamp and a nonce used to guard against replay attacks. The timestamp is generated by the source system to indicate when the message was created. The nonce is a sequence number or randomly generated string (e.g. UUID) that would not be repeated by the source system for at least a day. This serves to improve encryption.

- Context: A string that can be used to identify the context of the message. This can help provide an application level guard against incorrect message consumption in configurations where there may be multiple system environments running over the same messaging infrastructure. Some example values are PRODUCTION, TESTING, STUDY and TRAINING.

- Timestamp: An ISO-8601 compliant string that identifies the time the message was sent. This is analogous to the JMSTimestamp provided by JMS. Either Zulu (‘Z’) time or time with a time zone offset may be used.

- Source: identifying the source of the message, which should be the name of the system or organization.

- AsyncReplyFlag: A Boolean (‘true’ or ‘false’) that indicates whether a reply message will be sent asynchronously. Replies are assumed to be sent synchronously by default.

- ReplyAddress: The address to which replies should be sent. This is typically used for asynchronous replies. This should take the form of a URL, topic name or queue name.This is analogous to the JMSReplyTo field provided by JMS. This is ignored when using one-way integration patterns (e.g. AckRequired=false). If the reply address is a topic, the topic name should be prefixed by ‘topic:’. If the reply address is a queue, the queue name should be prefixed by ‘queue:’. If the reply address is a web service, the reply address should be a URL beginning with ‘http://’ or ‘https://’.

- AckRequired: This is a Boolean (‘true’ or ‘false’) that indicates whether or not an acknowledgement is required. If false, this would indicate that a one-way integration pattern is being used for communicating transactional messages.

- User: A complex structure that identifies the user and associated organization. Should be supplied as it may be required for some interfaces, depending upon underlying implementations. This allows a UsersID string and optional Organization string as sub-elements.

- MessageID: A string that uniquely identifies a message. Use of a UUID or sequence number is recommended. This is analogous to the JMSMessageID provided by JMS. A process should not issue two messages using the same MessageID value.

- CorrelationID: This is used to ‘link’ messages together. This can be supplied on a request, so that the client can correlate a corresponding reply message. The server will place the incoming CorrelationID value as the CorrelationID on the outgoing reply. If not supplied on the request, the CorrelationID of the reply should be set to the value of the MessageID that was used on the request, if present. This is analogous to the used of the JMSCorrelationID provided by JMS. Given that the CorrelationID is used to ‘link’ messages together, it may be reused on more than one message. Use of a UUID or sequence number is recommended.

- Comment: Any descriptive text, but must never be used for any processing logic.

- Property: A complex type that allows custom name/value pairs to be conveyed. The source and targets would need to agree upon usage. These are analogous to a Property as defined by JMS.

- any: Can be used for custom extensions.

The following diagram describes the header structure used for request, response and event messages.

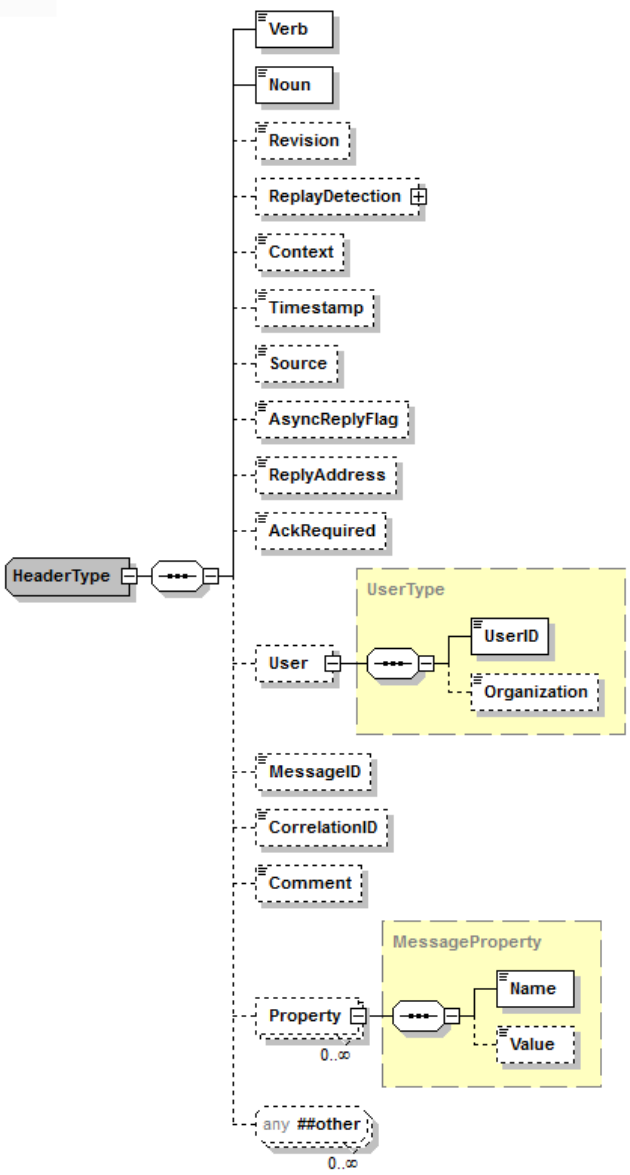


Figure 23 - Common Message Header Structure

Where there are only two required elements, verb and noun, there are many optional elements that may be populated. In the above diagram, the optional items are represented using dashed borders.

The following is an XML example for a message that populates all header fields.

```
<?xml version="1.0" encoding="UTF-8"?>
<RequestMessage xsi:schemaLocation="http://iec.ch/TC57/2011/schema/message
Message.xsd" xmlns="http://iec.ch/TC57/2011/schema/message"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
       <Header>
              <Verb>get</Verb>
              <Noun>LoadForecast</Noun>
              <Revision>1</Revision>
              <ReplayDetection>
                     <Nonce>dcd98b7102dd2f0e8b11d0f600bfb0c093</Nonce>
                     <Created>2012-12-16T09:30:47.0Z</Created>
              </ReplayDetection>
              <Context>PRODUCTION</Context>
              <Timestamp>2001-12-16T09:30:47.0Z</Timestamp>
              <Source>EMS</Source>
```

```
                <AsyncReplyFlag>false</AsyncReplyFlag>
                <ReplyAddress>queue:EMS.ReplyQueue</ReplyAddress>
                <AckRequired>true</AckRequired>
                <User>
                        <UserID>Bob</UserID>
                        <Organization>Scheduling</Organization>
                </User>
                <MessageID>3432626</MessageID>
                <CorrelationID>3432626</CorrelationID>
                <Comment>Example message</Comment>
                <Property>
                        <Name>timeout</Name>
                        <Value>10</Value>
                </Property>
        </Header>
        <Request>
                <StartTime>2012-12-17T00:00:00.0Z</StartTime>
                <EndTime>2012-12-17T24:00:00.0Z</EndTime>
        </Request>
</RequestMessage>
```

In cases where the message is conveyed using a transport such as SOAP or JMS, there is some redundancy between the optional fields in the message envelope and the transport-level header. In these cases, both fields can simply be set to the same value. In cases where they are different, they shall be used as appropriate for the transport-level and application-level message envelope.

6.3.3 Request Message Structures

The following diagram describes the structure of a request message that would be used in conjunction with a message or WSDL operation.

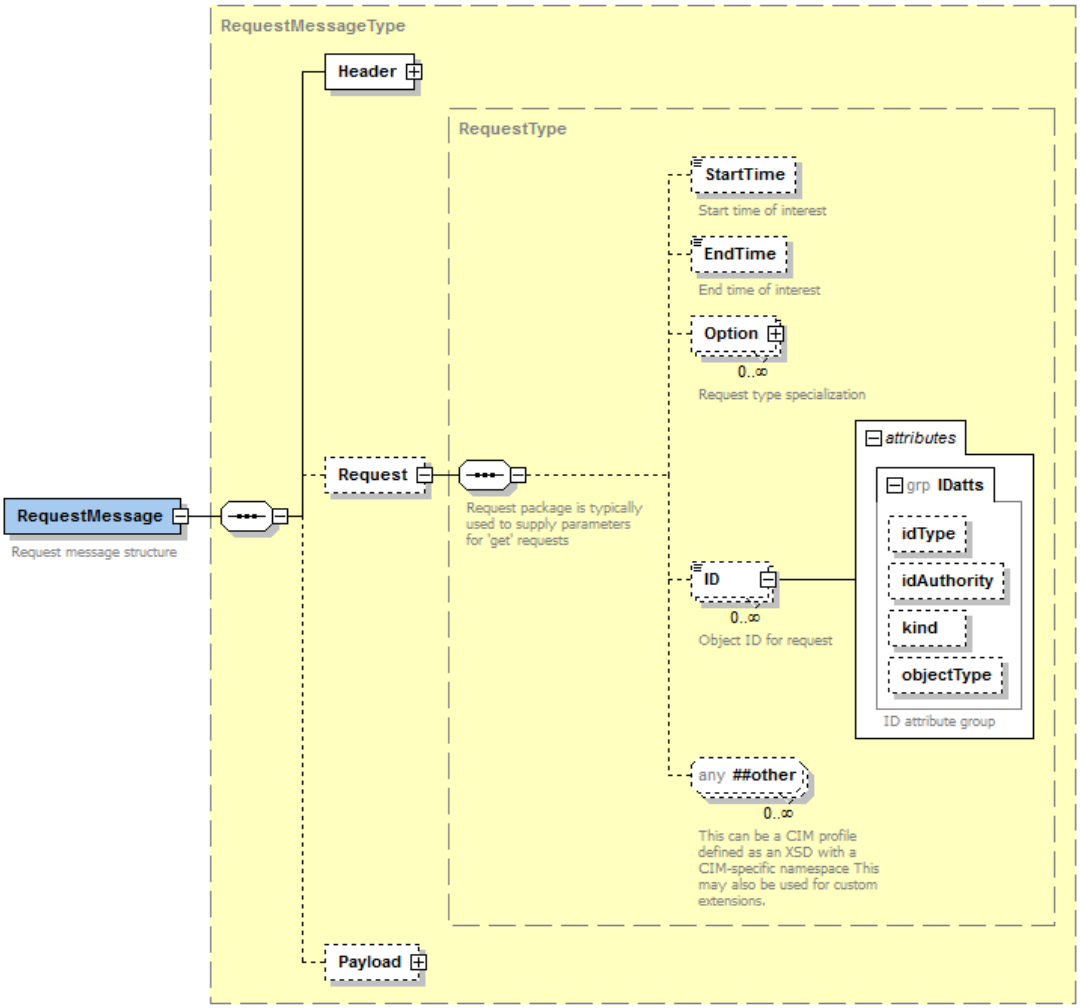


Figure 24 - Request Message Structure

The RequestMessage can also optionally contain an element with parameters relevant to the request, called Request. One key use of the RequestType is to avoid the placement of application specific request parameters in the header or within payload definitions.

There are no required elements in the Request element. The usage of elements within the Request element is described as follows:

- StartTime: Used when a query needs to specify a start time as a filter, but no such parameter is provided in a ‘Get’ profile. If both exist, this will be ignored.
- EndTime: Used when a query needs to specify an end time as a filter, but no such parameter is provided in a ‘Get’ profile. If both exist, this will be ignored.
- Option: Used when name/value pairs are useful in filtering a query or to convey general or custom request options. Examples of usage are the specification of a transaction timeout value or specifying a response mode such as ‘Aggregated’ or ‘Streaming’. At the current time there are no normative enumerations for these values.

- ID: Used when the ID of one or more objects are needed to filter a query request. Can also be used to identify specific objects in the case of ‘delete’, ‘cancel’ or ‘close’ transactions. Each ID can specify attributes, first to identify the kind of ID, which can be name, uuid, transaction or other. The default of uuid is used for mRID values. If a name, the idType and idAuthority can be specified.
- any: Used to supply a ‘Get<Noun>’ profile (e.g. GetMeterReadings) element to be conveyed with parameters that can qualify a request. Can also be used for other non-standard extensions. In cases where a ‘Get’ profile is used, the elements defined within the ‘Get’ profile take precedence over the StartTime, EndTime and ID elements. This recognizes the asymmetry between the information needed to qualify a request from the information that is returned on a reply.

Situations that may use the Option Name/Value pair can be described as a part of other standards. In some cases it may be decided that these should require changes to other standard request elements (i.e. the Get<Noun> elements described in the next sub-clause) in order to facilitate such a request.

The following is an example of a RequestMessage where the Request.ID elements are used to identify objects of interest:

```
<ns0:RequestMessage xmlns:ns0 = "http://www.iec.ch/TC57/2011/schema/message">
<ns0:Header>
   <ns0:Verb>get</ns0:Verb>
   <ns0:Noun>Switches</ns0:Noun>
   <ns0:Revision>1</ns0:Revision>
   <ns0:CorrelationID>1729363b5b7d9c6a0a88d02ae97c64b0</ns0:CorrelationID>
</ns0:Header>
<ns0:Request>
   <ns0:ID>b9cd8d2a-56a2-45e3-89d0-caaabb9e2985</ns0:ID>
   <ns0:ID>e6d957ba-792a-4fcf-9f33-fd176a66dee8</ns0:ID>
   <ns0:ID>567fdc86-0ccd-4a96-a318-bdc1a3015643</ns0:ID>
</ns0:Request>
</ns0:RequestMessage>
```

Figure 25 - XML for Example RequestMessage

The ‘any ##other’ element should be used when more complex request parameters are needed in order to qualify a request so that the resulting response message is appropriately filtered. The following is an example of a ‘GetMeterReadings’ element that is used to provide qualifiers for get MeterReadings requests.

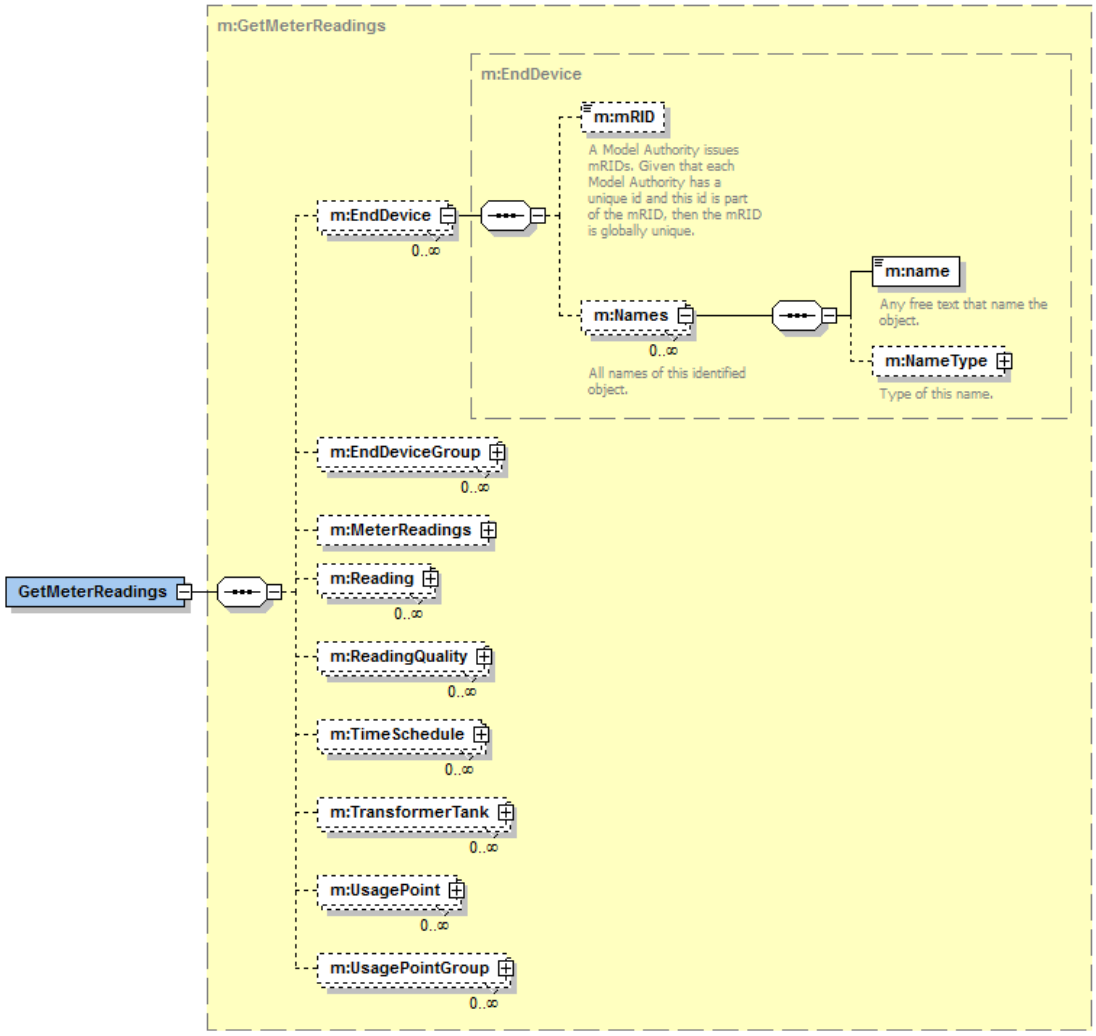


Figure 26 - Example 'Get<Noun>' Profile

6.3.4 Response Message Structures

The following diagram describes the structure of a response message that would be used in conjunction with a message or WSDL operation, as a response to the request message.

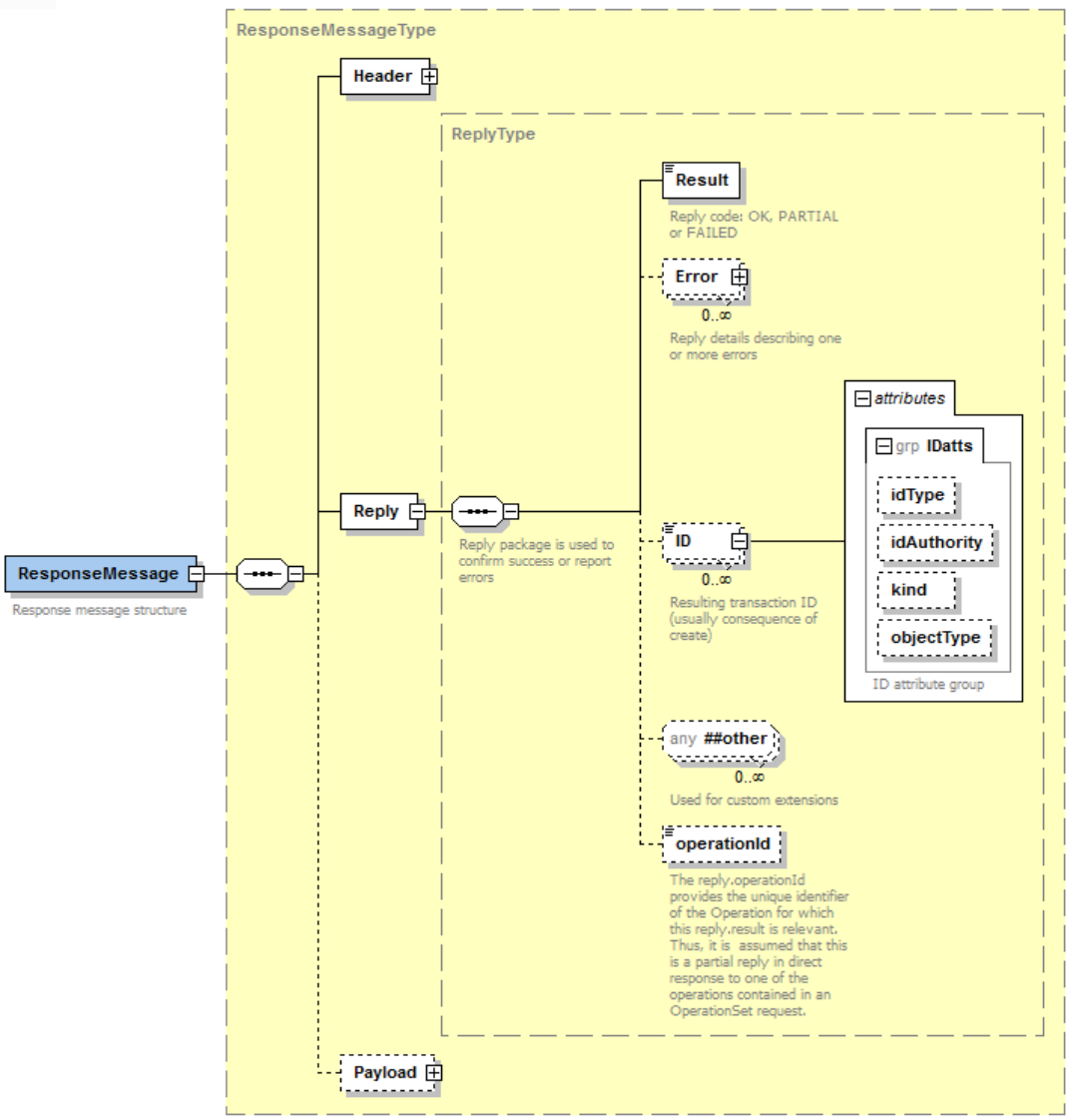


Figure 27 - Response Message Structure

The Reply.result value is enumerated in Message.xsd, and would be populated in the following manner:

- "OK" if there are no errors and all results have been returned. There is no requirement that a Reply.Error element be present.
- "PARTIAL" if only a partial set of results has been returned, with or without errors. Existence of errors is indicated with one or more Reply.Error.code elements.
- "FAILED" if no result can be returned due to one or more errors, indicated with one or more Reply.Error elements, each with a mandatory application level ‘code’ ‘.

This is represented by the following state transition diagram.

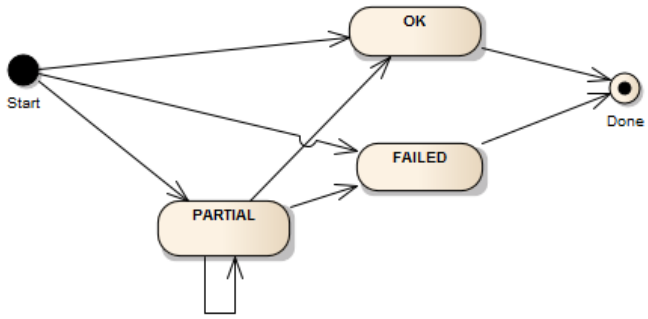


Figure 28 - Reply Message States

There may also be more specific error information provided within the payload itself. The following figure describes the details of the Error element, which allows for both human and machine readable error messages. The structure allows the use of XPath to allow specific references to elements within an XML document.

If the entire response to a request message is being provided in a single response message and the response message contains no fatal errors, then the Reply.Result is set to "OK" and the Reply.Error.code is set with a value of "0.0".

Otherwise, if the entire response to a request message is being returned in a single message and the response message contains at least one fatal error then the Reply.Result is set to "FATAL". Such a message may contain a mixture of data items and error notifications. The Reply.Error.code, Reply.Error.ID, and other associated Reply.Error structure attributes are then set appropriately for each fatal error or informational condition being reported.

If the responding system is sending multiple response messages to a request message the Reply.Result is set to "PARTIAL". Such messages may contain a mixture of data items and error notifications. There must be at least one Reply.Error.code of "0.2" or "0.1", depending upon whether a given response message is the last in a sequence or not (*). The Reply.Error.code, Reply.Error.ID, and other associated Reply.Error structure attributes are then set appropriately for each fatal error or informational condition being reported.

In the case where the responding system cannot determine last message in a set of response messages, then all messages in the set are to be sent with a Reply.Error.code = "0.1".

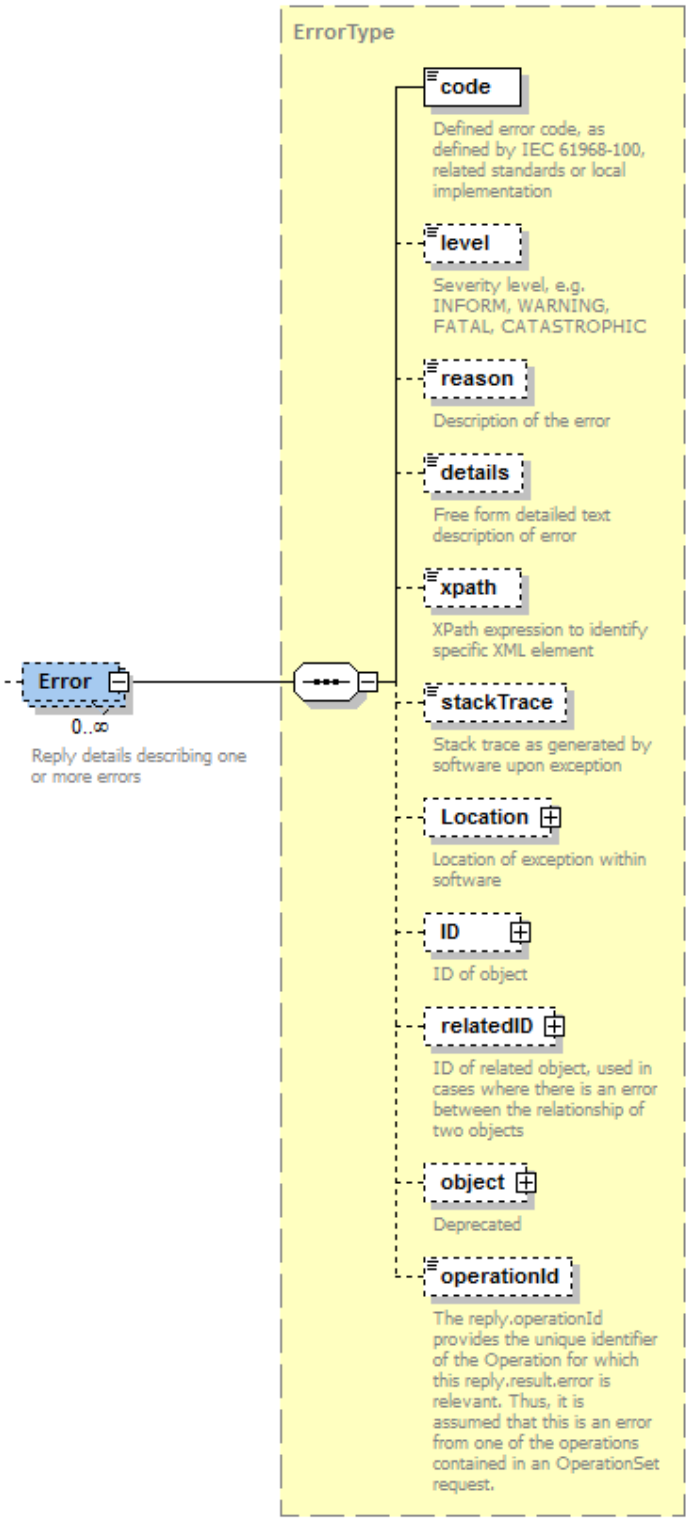


Figure 29 - Error Structure

The following is an example of a ResponseMessage:

```
<ns0:ResponseMessage xmlns:ns0 = "http://www.iec.ch/TC57/2011/schema/message">
<ns0:Header>
   <ns0:Verb>reply</ns0:Verb>
   <ns0:Noun>Switches</ns0:Noun>
```

```
   <ns0:CorrelationID>1729363b5b7d9c6a0a88d02ae97c64b0</ns0:CorrelationID>
</ns0:Header>
<ns0:Reply>
   <ns0:Result>OK</ns0:Result>
</ns0:Reply>
<ns0:Payload>
   <m:Switches xsi:schemaLocation="http://iec.ch/TC57/2012/Switches# Switches.xsd"
xmlns:m="http://iec.ch/TC57/2012/Switches#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <m:Switch>
         <m:mRID>b9cd8d2a-56a2-45e3-89d0-caaabb9e2985</m:mRID>
         <m:normalOpen>true</m:normalOpen>
      </m:Switch>
      <m:Switch>
         <m:mRID>e6d957ba-792a-4fcf-9f33-fd176a66dee8</m:mRID>
         <m:normalOpen>true</m:normalOpen>
      </m:Switch>
      <m:Switch>
         <m:mRID>567fdc86-0ccd-4a96-a318-bdc1a3015643</m:mRID>
         <m:normalOpen>false</m:normalOpen>
      </m:Switch>
   </m:Switches>
</ns0:Payload>
</ns0:ResponseMessage>
```

Figure 30 - XML for Example ResponseMessage

The following is an example of a ResponseMessage where the payload is compressed:

```
<ns0:ResponseMessage xmlns:ns0 = "http://www.iec.ch/TC57/2011/schema/message">
<ns0:Header>
   <ns0:Verb>reply</ns0:Verb>
   <ns0:Noun>Switches</ns0:Noun>
   <ns0:CorrelationID>1729363b5b7d9c6a0a88d02ae97c64b0</ns0:CorrelationID>
</ns0:Header>
<ns0:Reply>
   <ns0:Result>OK</ns0:Result>
</ns0:Reply>
<ns0:Payload>
   <ns0:Compressed>dghuywqeiwihn353218u23hb2b3b3bhu</ns0:Compressed>
   <ns0:format>XML</ns0:format>
</ns0:Payload>
</ns0:ResponseMessage>
```

Figure 31 - XML Example of Payload Compression

The following is an example ResponseMessage that returned an error:

```
<ns0:ResponseMessage xmlns:ns0 = "http://www.iec.ch/TC57/2011/schema/message">
<ns0:Header>
   <ns0:Verb>reply</ns0:Verb>
   <ns0:Noun>Switches</ns0:Noun>
   <ns0:Revision>1</ns0:Revision>
   <ns0:CorrelationID>1729363b5b7d9c6a0a88d02ae97c64b0</ns0:CorrelationID>
</ns0:Header>
<ns0:Reply>
   <ns0:Result>FAILED</ns0:Result>
   <ns0:Error>
      <ns0:code>2.15</ns0:code>
      <ns0:level>WARNING</ns0:level>
      <ns0:details>Unknown object: e6d957ba-792a-4fcf-9f33-fd176a66dee8</ns0:details>
   </ns0:Error>
</ns0:Reply>
<ns0:Payload>
   <m:Switches xsi:schemaLocation="http://iec.ch/TC57/2012/Switches# Switches.xsd"
xmlns:m="http://iec.ch/TC57/2012/Switches#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <m:Switch>
         <m:mRID>b9cd8d2a-56a2-45e3-89d0-caaabb9e2985</m:mRID>
         <m:normalOpen>true</m:normalOpen>
      </m:Switch>
      <m:Switch>
         <m:mRID>567fdc86-0ccd-4a96-a318-bdc1a3015643</m:mRID>
```

```
        <m:normalOpen>false</m:normalOpen>
      </m:Switch>
   </m:Switches>
</ns0:Payload>
</ns0:ResponseMessage>
```

Figure 32 - XML Example for Error ResponseMessage

An important advantage of payload compression over the use of SOAP attachments is for signing, as a SOAP signature does NOT sign the contents of the attachment, only the message body. Using payload compression the signature covers the payload, providing for non-repudiation.

6.3.5 Event Message Structures

An EventMessage is typically published to report a condition of potential interest. The verbs used in an event message are past tense, e.g. created, changed, cancelled, etc. An EventMessage will not include request or reply parameters, just a header and usually a payload.

The following diagram describes the structure of an EventMessage.

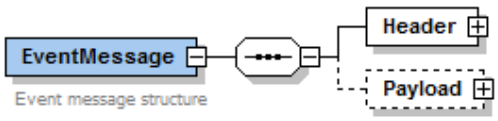


Figure 33 - EventMessage Structure

The following is an example of an EventMessage:

```
<ns0:EventMessage xmlns:ns0 = "http://www.iec.ch/TC57/2011/schema/message">
<ns0:Header>
   <ns0:Verb>changed</ns0:Verb>
   <ns0:Noun>Switches</ns0:Noun>
   <ns0:Revision>1</ns0:Revision>
</ns0:Header>
<ns0:Payload>
   <m:Switches xsi:schemaLocation="http://iec.ch/TC57/2012/Switches# Switches.xsd"
xmlns:m="http://iec.ch/TC57/2012/Switches#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <m:Switch>
         <m:mRID>b9cd8d2a-56a2-45e3-89d0-caaabb9e2985</m:mRID>
         <m:normalOpen>false</m:normalOpen>
      </m:Switch>
      <m:Switch>
         <m:mRID>567fdc86-0ccd-4a96-a318-bdc1a3015643</m:mRID>
         <m:normalOpen>true</m:normalOpen>
      </m:Switch>
   </m:Switches>
</ns0:Payload>
</ns0:EventMessage>
```

Figure 34 - XML Example for EventMessage

Note that in an EventMessage the ‘verb’ will be past tense, e.g. ‘created’, ‘changed’, ‘canceled’, etc.

6.3.6 Fault Message Structures

A FaultMessage is typically used within the definition of a WSDL and implemented by a web service to report a fault condition as a consequence of a failed attempt to process a RequestMessage (e.g. detection of a SOAP fault). It only uses a reply element (i.e.

no header), as it may not have been able to interpret even the header of the RequestMessage.

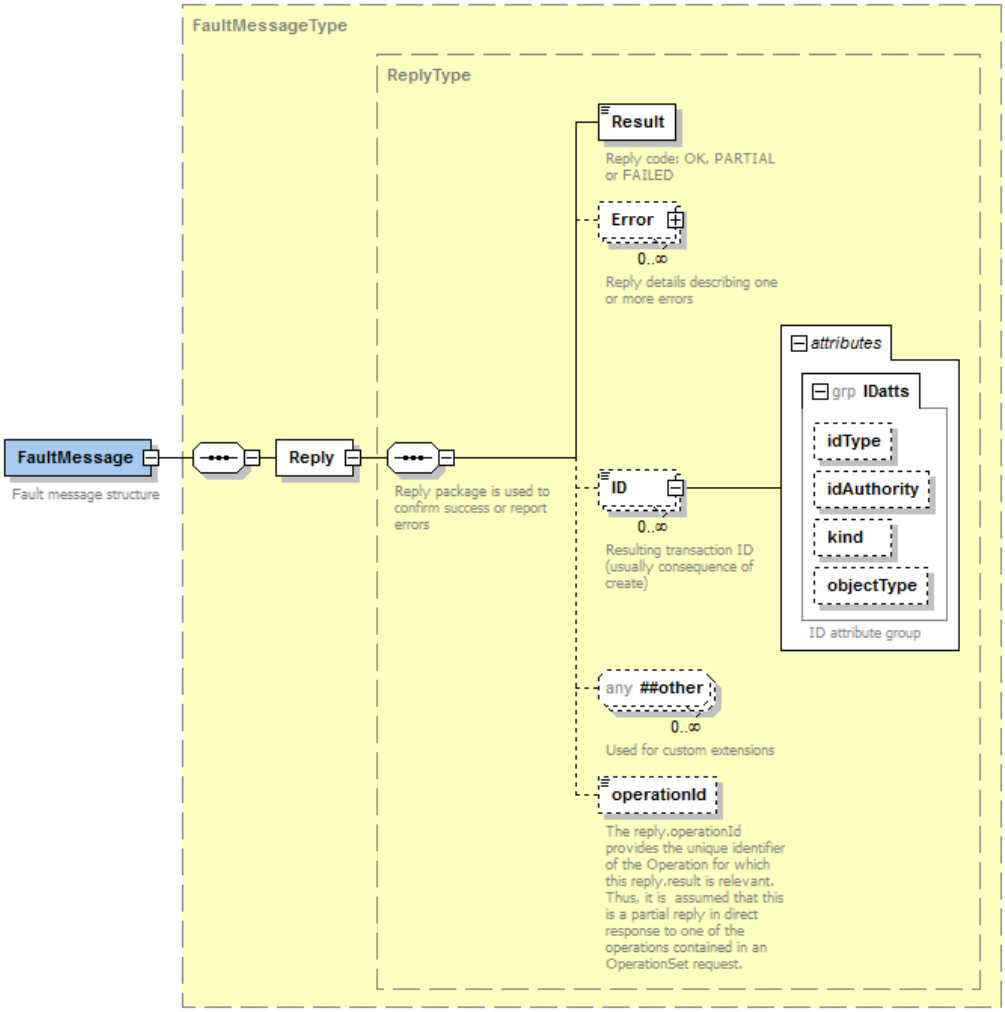


Figure 35 - Fault Message Structure

6.4 Payload Structures

This sub-clause describes two forms of payload structures: generic and type-specific. Where the common message envelope defines the payload as being generic (or type-independent), which is the common usage with both JMS and generic web services. However, the definition of WSDLs for strongly-typed web services may require a payload definition of a variant message envelope that is type specific.

There are some types of messages where a Payload must be provided, as would be the case for a request message with a verb of ‘create’ or ‘change’, some response messages and some event messages. Payloads typically contain XML documents that conform to a defined XML schema. However, there are exceptions to this rule. Some XML payloads could potentially not have useful XML schemas, as in the case of RDF files or dynamic query results, as well as non-XML formats such as CSV and PDF.

There may also be cases where a large payload must be compressed, in the event that it would become very large and otherwise consume significant network bandwidth. In order to accommodate a variety of payload format options the following payload structure is used.

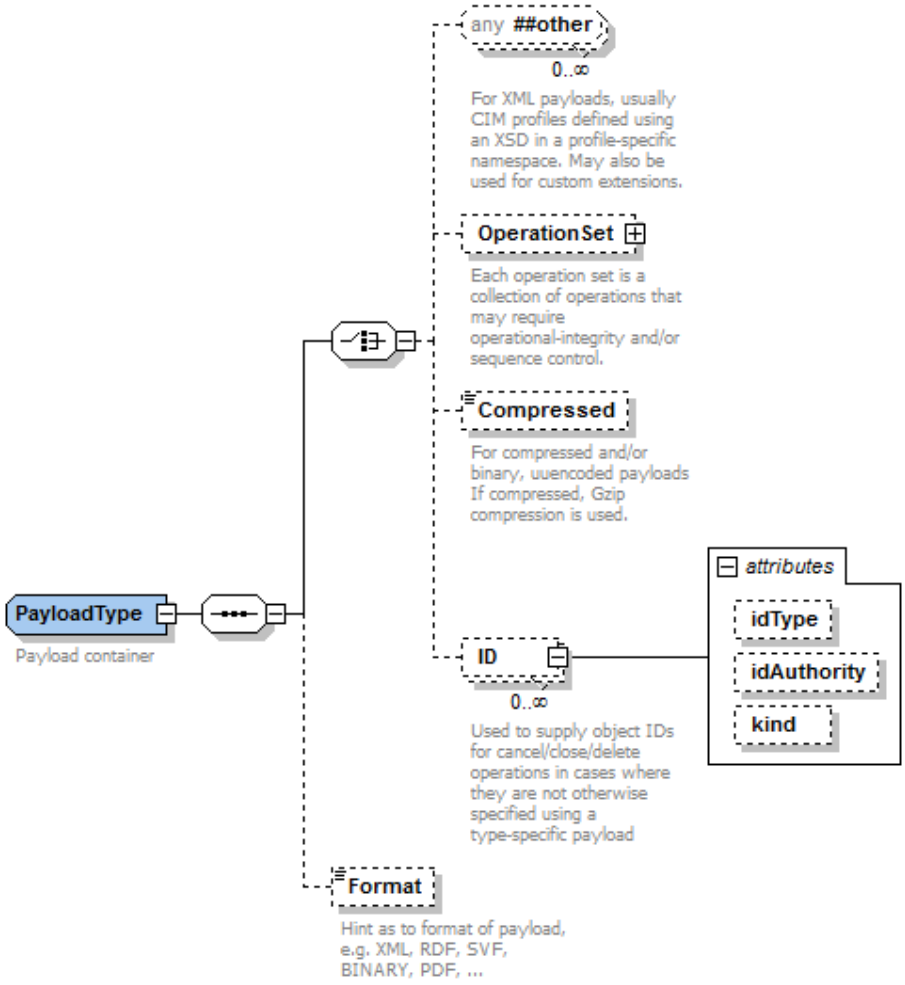


Figure 36 - Message Payload Container - Generic

Section 6.10 discusses the use of the OperationSet element for complex transactions in more detail.

When the generic message payload container is used, any type of XML document may be included, using the XML ‘any’ structure. While this provides options for loose-coupling, specific complex types defined by XML schemas (XSDs) can be used as well.

There are also some cases where a zipped, base64 encoded string is necessary, and would be passed using the ‘Compressed’ tag within the message. The base64 encoding must always be performed after compression. The ‘Compressed’ element is used even in caes where binary data is not compressed. The Gnu Zip compression shall be used in order to provide compatibility within both Java and Microsoft .Net implementations. A Java example is provided in Annex F. Specific examples of the usage of payload compression would be where:

- An XML payload, conforming to a recognized XML schema exceeds a predefined size (e.g. 1MB, 5MB, 10MB, …). This would be common for model exchanges and energy market transactions.

- A payload has a non-XML format, such as PDF, Excel spread sheet, CSV file or binary image.
- A payload is formatted using XML, but has no XML schema and exceeds a predefined size (e.g. 1MB, 5MB, 10MB, …). An example of this would be the case of dynamic XML generated as a consequence of a SQL XML query that would return an XML result set

When a payload is compressed and base64 encoded, it is stored within the Payload/Compressed message element as a string. Additionally, in order to support efficient transfer of binary formatted data, data can be base64 encoded but not compressed. This would be used for data classified as being 'high speed', where XML formatting would not meet performance needs.

The ID element and associated attributes can be used to supply object or transaction identifiers. This is useful in cases where a payload does not otherwise provide object identifiers as may commonly be the case for cancel, close or delete requests, responses to create requests or events using the canceled, closed or deleted verbs.

The Format element can be used to identify specific data formats, such as XML, RDF, SVG, BINARY, PDF, DOC, CSV, etc. This is especially useful if the payload is base64 encoded and potentially compressed. The use of this tag is optional, and would typically only be used when the payload is stored using the Payload/Compressed message element. The following table describes the relationships between elements in the Payload, showing a wide variety of payload options.

Any	***OperationSet***	***Compressed***	***Format***	***Interpretation***
XML	null	null	Null or 'XML'	Message payload is contained within the 'any', where the contents are described by the header Noun. This is the most common usage.
null	null	string	'XML'	Message payload is gzipped and base64 encoded in the Compressed element, where the contents are described by the header Noun.
null	XML	null	Null or 'XML'	Complex transaction is being conveyed using the OperationSet element, where the header Verb is 'execute' or 'executed'
XML	null	null	'RDF'	Message payload is an RDF document as conveyed using the 'any' element
null	null	string	'RDF'	Message payload is a compressed RDF document as conveyed in the Compressed element
null	null	string	'PDF'	Message payload is a compressed PDF document as conveyed in the Compressed element
null	null	string	'GZIP'	Message payload is a gzip archive of one or more files as conveyed in the Compressed element
null	null	string	'CSV'	Message payload is a CSV file being conveyed in the Compressed element
null	null	string	'XLS'	Message payload is an Excel file being conveyed in the Compressed element
null	null	string	'DOC'	Message payload is a Word document being conveyed in the Compressed element

null	null	string	'TEXT'	Message payload is a compressed text document as conveyed in the Compressed element
null	null	string	'JSON'	Message payload is a compressed JSON object as conveyed in the Compressed element
null	null	string	'BINARY'	Message payload is a binary structure that has been base64 encoded but not compressed. Any further aspects are application specific.
null	null	string	*other*	Message payload is a compressed file of some 'other' format as conveyed in the Compressed element. This allows the definition of custom Format values.

Figure 37 - Payload Usages

These payload options provide an alternative to the use of SOAP attachments. SOAP attachments are more difficult to secure since the SOAP envelope signature signs the SOAP body but does not sign the attachment. This also requires that the payload is processed separately from the rest of the SOAP message (e.g. the message is parsed to extract the payload, and then the payload is parsed and processed). However, we believe this implementation approach is less complex than using SOAP attachments.

6.5 Strongly-Typed Payloads

In cases where strongly-typed WSDLs are to be defined with operations specific for combinations of verb and noun, the common message envelope is redefined with the following two substitutions:

- Root element "Message" replaced with a WSDL operation name such as "CreateEndDeviceControls". This address a Web service "wire signature" issue if multiple operations reference a same XSD element.
- Payload contains a concrete message type element such as EndDeviceControls.

As a result, a message structure for CreateEndDeviceControl request service operation is then redefined as shown in the following diagram:

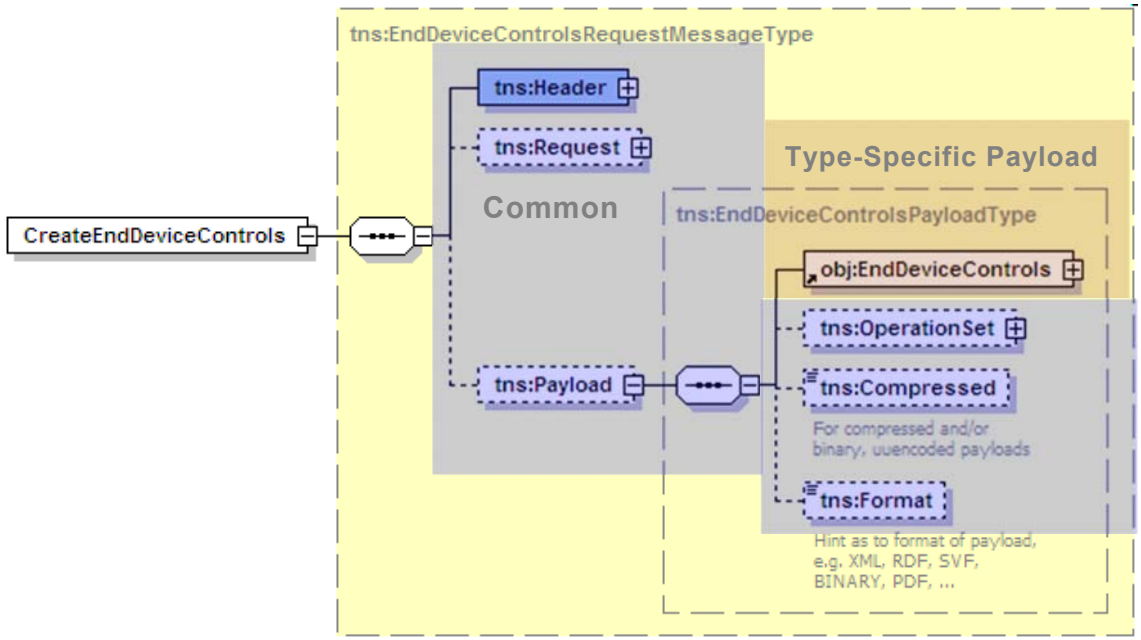


Figure 38 - Message Payload Container - Type Specific Example

Note the Message element is renamed to CreateEndDeviceControls for this service definition, and the Payload element contains a concrete object “EndDeviceControls” which is based on a CIM profile (or payload type). A key difference to be noted is the strong typing of the payload element, as opposed to use of the ‘any’ in the standard Message.xsd. This results in a type-specific version of Message.xsd per each IEC 61968 profile in use. See section 8.3 and annex C for further details.

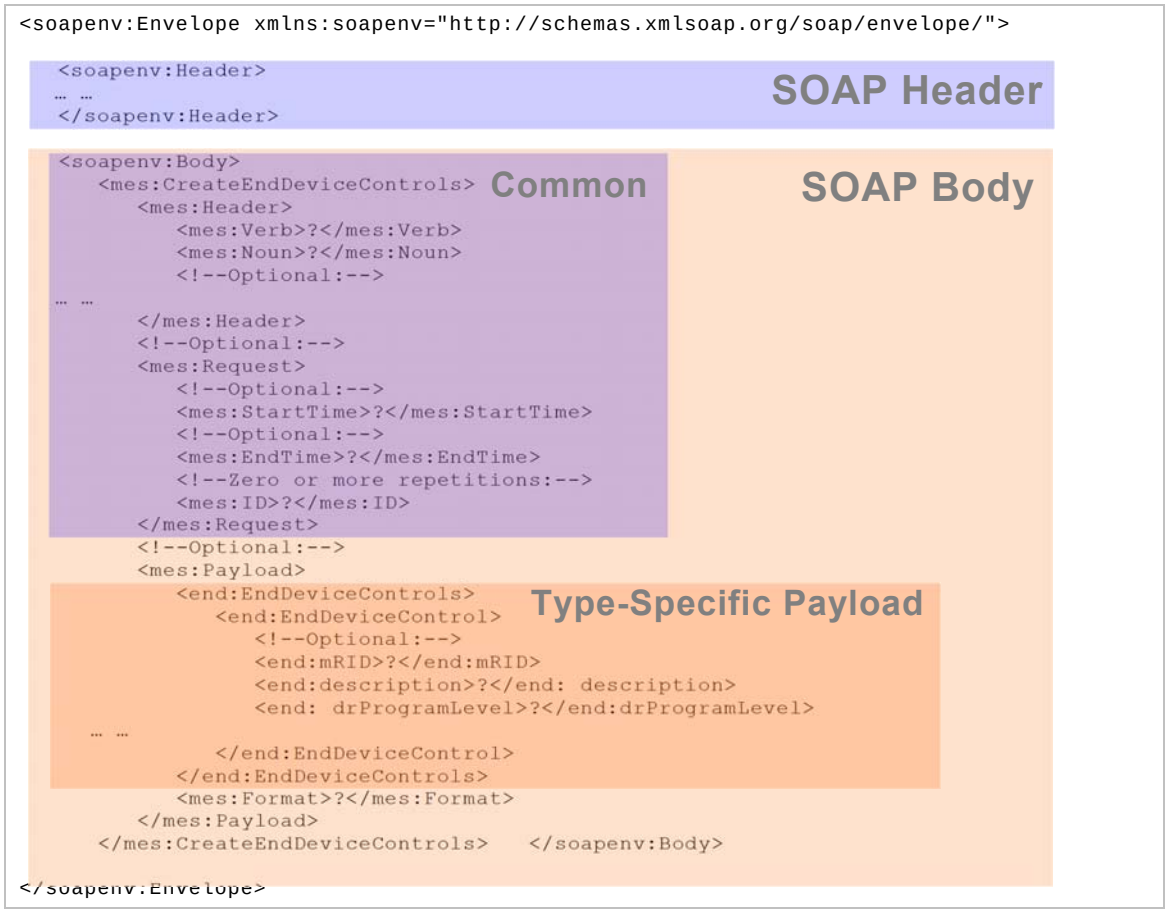
6.6 SOAP Message Envelope

SOAP has been widely used as a standard protocol specification for exchanging XML information using web services. It provides an envelope that contains a header and a body. How a SOAP message is structured can be defined in WSDL binding section as an example listed below:

```
<wsdl:binding name="EndDeviceControl_Binding" type="tns:EndDeviceControl">
   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="CreatedEndDeviceControl">
      <wsdl:documentation>CreatedEndDeviceControl binding</wsdl:documentation>
<soap:operation
soapAction="http://iec.ch/TC57/2010/EndDeviceControls/CreatedEndDeviceControls"
style="document"/>
      <wsdl:input name="CreatedEndDeviceControlRequest">
         <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="CreatedEndDeviceControlResponse">
         <soap:body use="literal"/>
      </wsdl:output>
      <wsdl:fault name="CreatedEndDeviceControlFault">
         <soap:fault name="CreatedEndDeviceControlFault" use="literal"/>
      </wsdl:fault>
   </wsdl:operation>
   …
```

In this WSDL, it is specified that where an input / output payload is located (soap:body - highlighted) and what binding style (=document - highlighted) it follows.

Based on the WSDL binding information, a SOAP message (see below) can be constructed. When using SOAP the message structure will appear within the context of the SOAP Body. This is shown in the following example.




```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header>
   … …
   </soapenv:Header>
   <soapenv:Body>
      <mes:CreateEndDeviceControls>
         <mes:Header>
            <mes:Verb>?</mes:Verb>
            <mes:Noun>?</mes:Noun>
            <!--Optional:-->
   … …
         </mes:Header>
         <!--Optional:-->
         <mes:Request>
            <!--Optional:-->
            <mes:StartTime>?</mes:StartTime>
            <!--Optional:-->
            <mes:EndTime>?</mes:EndTime>
            <!--Zero or more repetitions:-->
            <mes:ID>?</mes:ID>
         </mes:Request>
         <!--Optional:-->
         <mes:Payload>
            <end:EndDeviceControls>
               <end:EndDeviceControl>
                  <!--Optional:-->
                  <end:mRID>?</end:mRID>
                  <end:description>?</end: description>
                  <end: drProgramLevel>?</end:drProgramLevel>
      … …
               </end:EndDeviceControl>
            </end:EndDeviceControls>
            <mes:Format>?</mes:Format>
         </mes:Payload>
      </mes:CreateEndDeviceControls>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 39 - SOAP Envelope Example for Strong Typing

6.7 Request Processing

A request message is sent from a client to a service to initiate a query or transaction, where a response message is typically expected. The basic sequence of request processing is as follows:

1. Client constructs a request message using a common message envelope and specifying a verb and a noun. The noun identifies the type of the payload
2. The client sends the request message to the appropriate service interface. This can use transport technologies such as JMS or web services. The ESB implementation can transparently use intermediaries such as proxies, routers and adapters to transmit the request to the appropriate service instance.
3. The server accepts the message.
4. If the request message is invalid (e.g. incomplete, XML not well formed, etc.), a fault message may be returned to the client and processing terminates.
5. The service looks at the verb and noun combination, determining if the request can (or should) be processed, if not an error response message is sent to the client and processing terminates. This step can also consider checks for authentication and authorization.

6. The service performs the desired processing, parsing the payload as needed. The service may parse the payload using an appropriate XML schema (as would define the payload type described by the message noun), using XPath expressions, using XSL transformations or other mechanisms. The service may also consider parameters provided in the message header and request packages for processing.

7. A response message is constructed, where a payload is rendered of the type identified by the noun as needed. This would commonly be the case in response to a 'get' request. The message reply element should be used to convey either a Result of 'OK' or an error code as appropriate.

8. The response message is returned to the client. This can use web services, JMS or other transport technologies as expected by the client.

9. The client processes the response message, parsing the payload as needed. The client should examine the reply element of the message to see if the request was successful (i.e. Result='OK') or encountered one or more errors.

10. Processing is completed

It is important to note that there are a variety of failure scenarios that can occur between steps 2-8, where a client should be able to handle a fault or time out when waiting for the reply to a request.

6.8 Event Processing

An event message is a message that is published by a service (or more generally any event publisher) to potentially many listeners. Events may also be referred to as 'notifications'. Event listeners are consumers that have subscribed to one or more JMS topics of potential interest. In the case of web services, there would be an intermediary to send the events to subscribers listening via web services.

The basic sequence of event processing is as follows:

1. A service constructs an event message using a common message envelope and specifying a verb and a noun. The noun identifies the type of the payload, although not all event messages will have a payload.

2. The service sends the event message to the appropriate JMS topic. The ESB implementation can also transparently use intermediaries such as routers and adapters to transmit the event to the appropriate event listeners.

3. The listener accepts the message.

4. If the event message is invalid (e.g. incomplete, XML not well formed, etc.), processing terminates (typically after an error is logged).

5. The listener looks at the verb and noun combination, determining if the event can (or should) be processed, if not, processing terminates.

6. The listener performs the desired processing, parsing the payload as needed. The service may parse the payload using an appropriate XML schema (as would define the payload type described by the message noun), using XPath expressions, using XSL transformations or other mechanisms.

When transactional request messages (see 4.5~~4.4~~) are processed on the bus that use the verb 'create', 'change, 'close', 'cancel' or 'delete', a corresponding event should be published using the corresponding past tense verb (e.g. 'created', 'changed, 'closed', 'cancelled', 'deleted' or 'executed'). This should be issued after successful execution of the transaction by either the

service that processed the request or a bus component. In these cases, the payload is identical to the payload used for the corresponding request message used to invoke the transaction.

The delivery guarantees are a consequence of the definition of the specific JMS topic or queue, as well as the means by which a listener subscribes (e.g. durable subscription).

6.9 Message Correlation

One important aspect of asynchronous messaging patterns is the need to be able to correlate a request with a reply. The Header.CorrelationID is key to the association of requests with asynchronous replies. The following rules should be applied to messages to allow necessary 'linking':

- When a client provides a CorrelationID on a request, the value should be a either a hexadecimal UUID (e.g. '`D921A053-80C1-4DB6-960E-2603127B7B92`') or a generated sequence number (e.g. 100023, 100024, 100025,…) that is effectively unique within the client making the request.

- If a request message includes a CorrelationID, the response message should return the same CorrelationID.

- If no CorrelationID is provided on a request message but a MessageID is provided, the response message should set the CorrelationID to the value of the MessageID that was provided on the request. MessageID should also be UUID or generated sequence numbers.

- If no MessageID or CorrelationID is provided on a request message, there is no way to correlate an asynchronous response to a specific request. Consequentially the CorrelationID cannot be set in the response message in a manner that identifies a linkage to a specific request.

- If a service is generating events as a direct consequence of a specific request, the CorrelationID should be set on the corresponding event message as per the previous rules if possible, noting that this may not always be possible and it is therefore not a requirement. This would provide a correlation between the event and the transaction that caused it.

Refer to the discussion and CorrelationID usage example provided in section 5.2.5.

6.10 Complex Transaction Processing Using OperationSet

The purpose of this section is to describe the use of the OperationSet element provided by Message.xsd. This provides support for transactions. The Message.xsd message envelope has been extended to accommodate an OperationSet construct in both the payload and reply portions of Message.xsd.

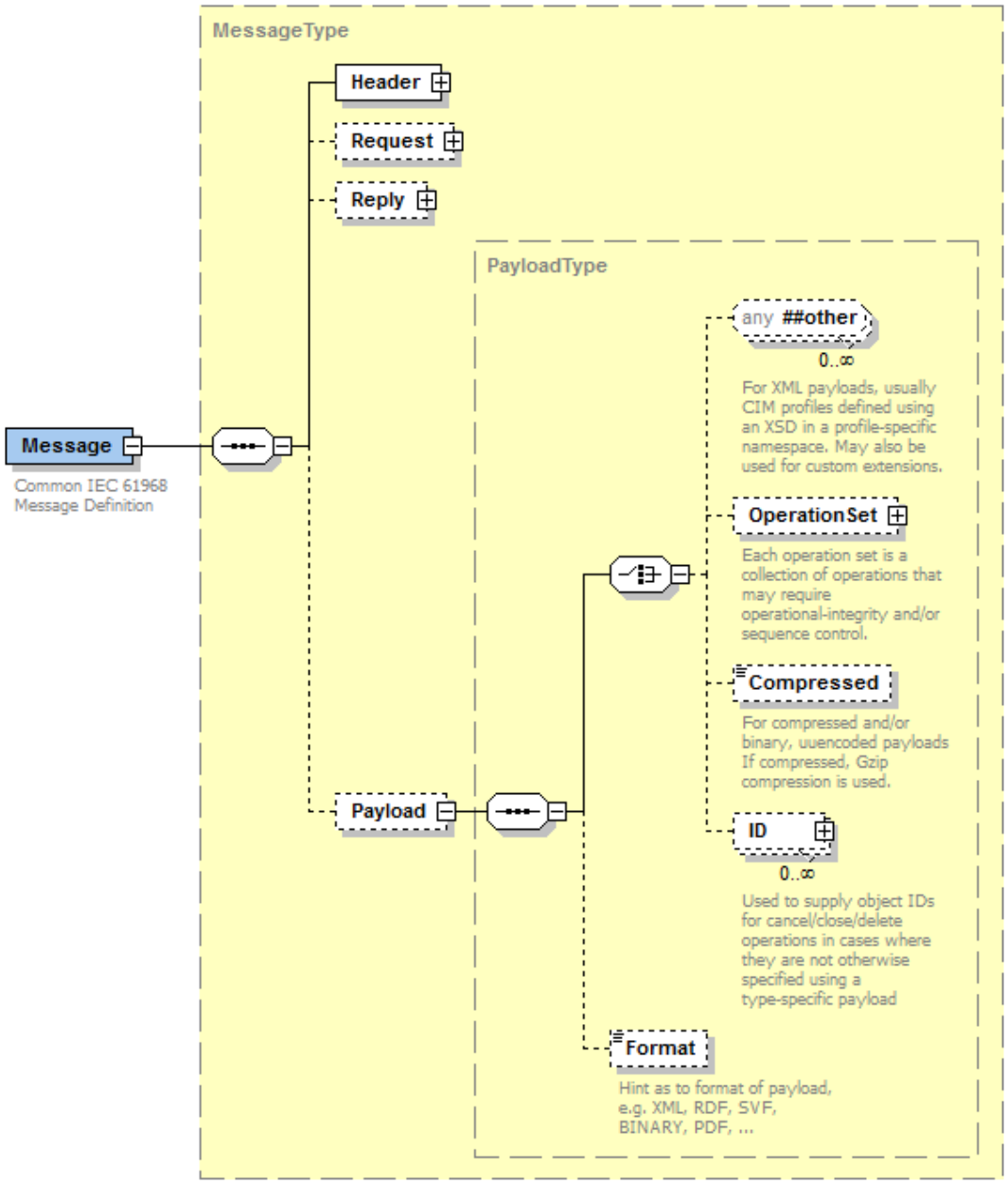


Figure 40 - Message OperationSet Element

There are two circumstances where the use of OperationSet might be necessary:

a. When modifying the configuration of a CIM object and the modification involves deleting one or more attributes or one or more instances of associated CIM objects. An example is removing a Register configuration from a Meter.
b. When performing two or more related actions that must be handled in a specific sequence and/or with overall transactional integrity (i.e., either all actions must succeed or all must be rolled back).

A message utilizing the OperationSet construct always has a Header verb of either ‘execute’ or ‘executed’ and a noun of ‘OperationSet’. An OperationSet in turn contains one or more Operation elements, and each OperationSet.Operation has an operationId which supplements the overall message CorrelationID to provide a fine-grained ability to correlate the contents of one or more reply messages with the individual operations in an OperationSet. Individual

Operation elements within an OperationSet have OperationSet-level verbs and nouns. Allowable verbs are create, created, change, changed, delete and deleted.

To support circumstance a) above, each Operation in an OperationSet also includes an elementOperation boolean. This Boolean is to be set to ‘true’ when the Operation verb is either ‘delete’ or ‘deleted’ and the intent is to delete individual attributes or individual instances of associated CIM classes from the object specified by the OperationSet noun (as opposed to deleting the entire CIM object specified by the Operation noun. If omitted, elementOperation is assumed to be ‘false’. It is emphasized that in this case, use of the Operation verb “delete” or “deleted” in combination with an elementOperation boolean set to ‘true’ effectively modifies (and does not delete) the CIM object specified by the Operation noun.

To support circumstance b) above, each OperationSet may have either an enforceMsgSequence boolean or an enforceTransactionalIntegrity boolean, or both. The enforceMsgSequence Boolean is to be set to ‘true’ when the Operations in the Operation set must be executed in ascending order of their operationID. The enforceTransactionalIntegrity boolean is to be set to ‘true’ if all Operations in the OperationSet must succeed. In this case, if all such Operations do not succeed, all must be rolled back. If either or both of these booleans are omitted, they are assumed to be ‘false’.

When modifying the configuration of a CIM object using any of the verbs ‘change’, ‘changed’, ‘delete’ or ‘deleted’, only the ID of the object being changed and the information that is being changed is to be included. This is true whether or not an OperationSet is being used. It is for this reason that almost all elements within the IEC 61968-9 Master Data Management Profiles are optional in the profiles.

It is recommended that only one OperationSet be used, as multiple OperationSets would place more burden upon consumers and potentially involve unnecessarily large messages.

It should also be noted that while this provides the means to convey transactions using XML schema-based data structures, it is also technically possible to leverage IEC 61970-552 for transactions based upon RDF.

6.10.1 OperationSet Element

The Figure 41 describes the OperationSet element in more detail. An OperationSet can:

- Require that each operation is sequentially executed by setting the enforceMsgSequence flag to ‘true’
- Require that transactional integrity be maintained (i.e. all or nothing), by setting the enforceTransactionalIntegrity flag to ‘true’
- Have one or more Operations, where each operation has a noun, verb and payload.

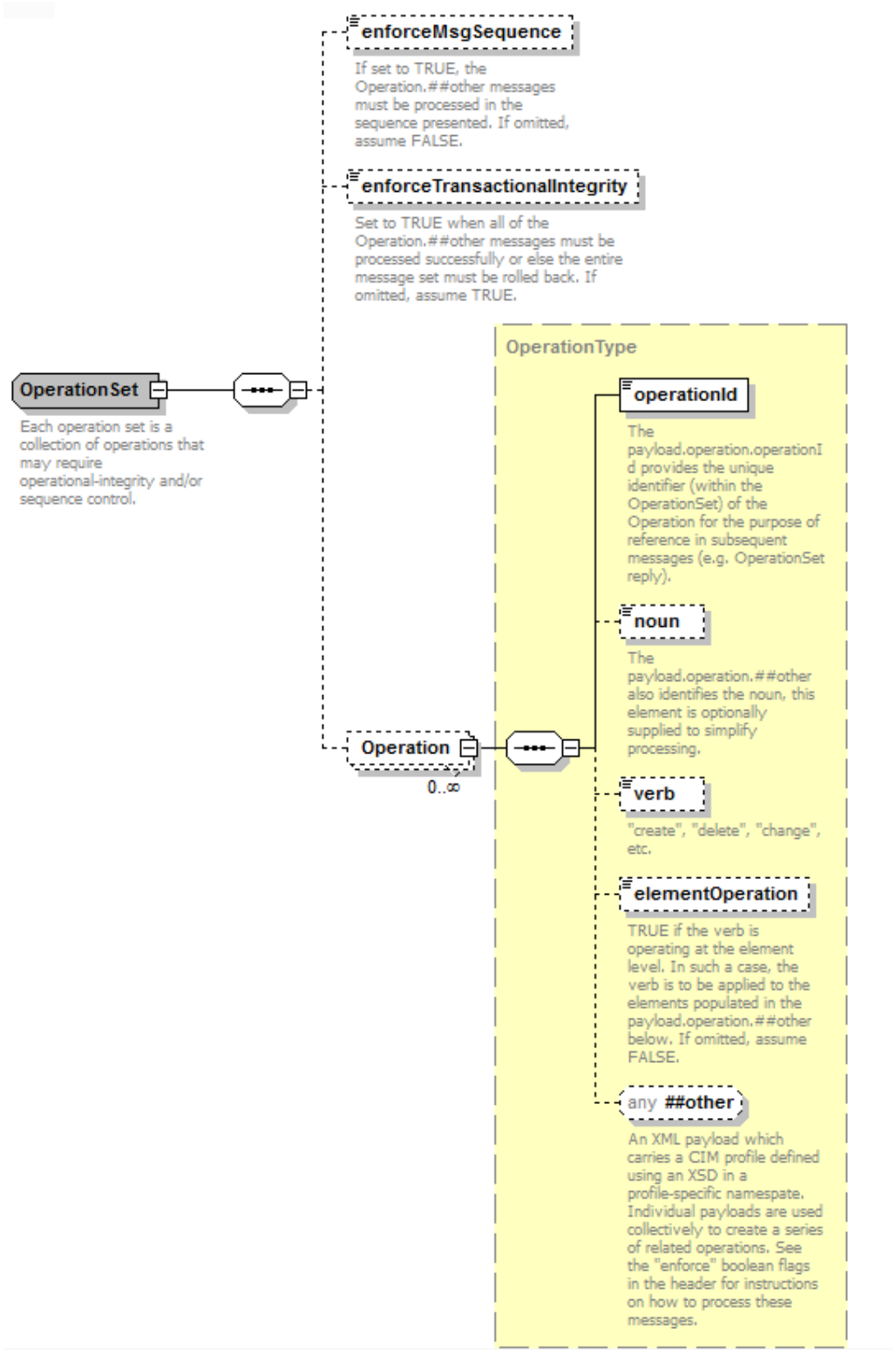


Figure 41 - OperationSet Details

Within the Operation element, the noun will identify the type of the any element. The elementOperation value will cause the transaction to either act upon the object or elements within the object. Examples provided will further describe usage.

6.10.2 Patterns

Any given transaction may be executed using either a request-response message pattern (Request stereotype and Response stereotype messages) or a published event message pattern (Event stereotype messages). Four exemplary sequence diagrams effectively illustrate the possible variations.

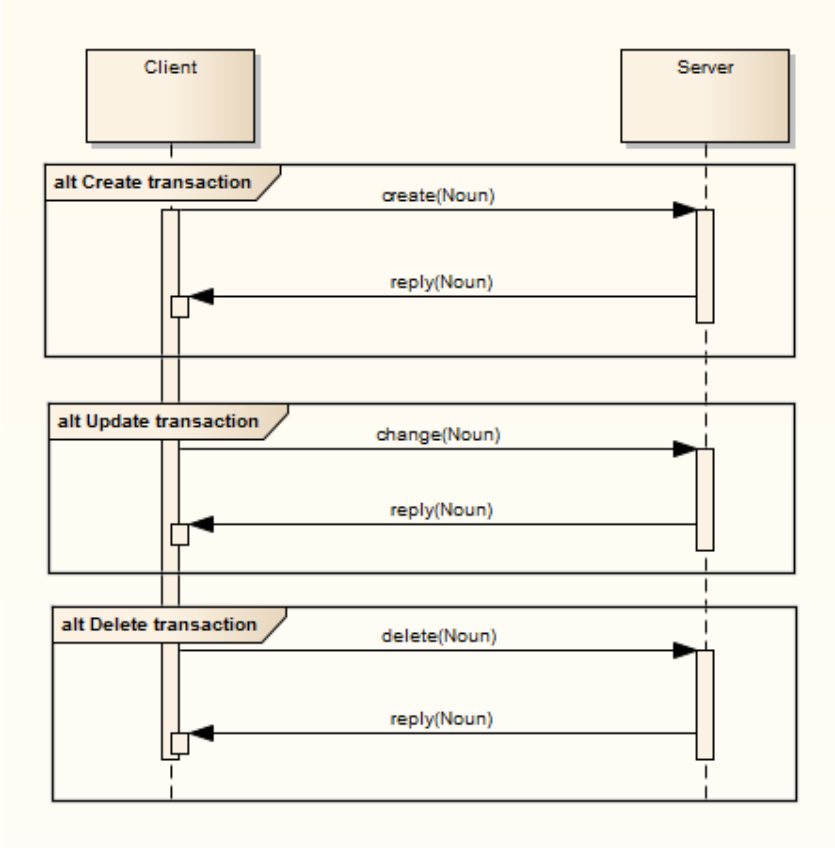


Figure 42 - Transactional Request/Response (non-OperationSet)

This request / response pattern can be used for transactions. Allowable verbs are ‘create’, ‘change’ and ‘delete’. Depending upon the scenario, there can be multiple replies to a given ‘create’, ‘change’, or ‘delete’ message. For example, a single create message can be issued to create multiple meters. In this case, the responding system can send a single reply message for all meters or multiple reply messages with the reply data for one or more meters in each message.

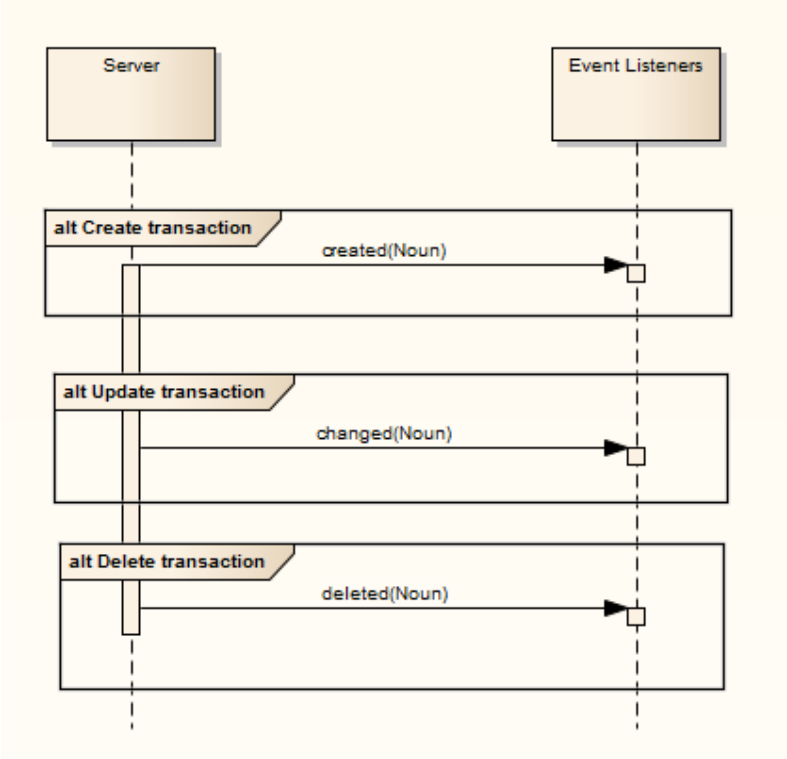


Figure 43 - Published Events (non-OperationSet)

This published event pattern can also be used for transactions. Allowable verbs are ‘created’, ‘changed’ and ‘deleted’. Using this pattern, an enterprise system may notify one or more other enterprise systems of events without requiring any acknowledgment or confirmation of successful processing.

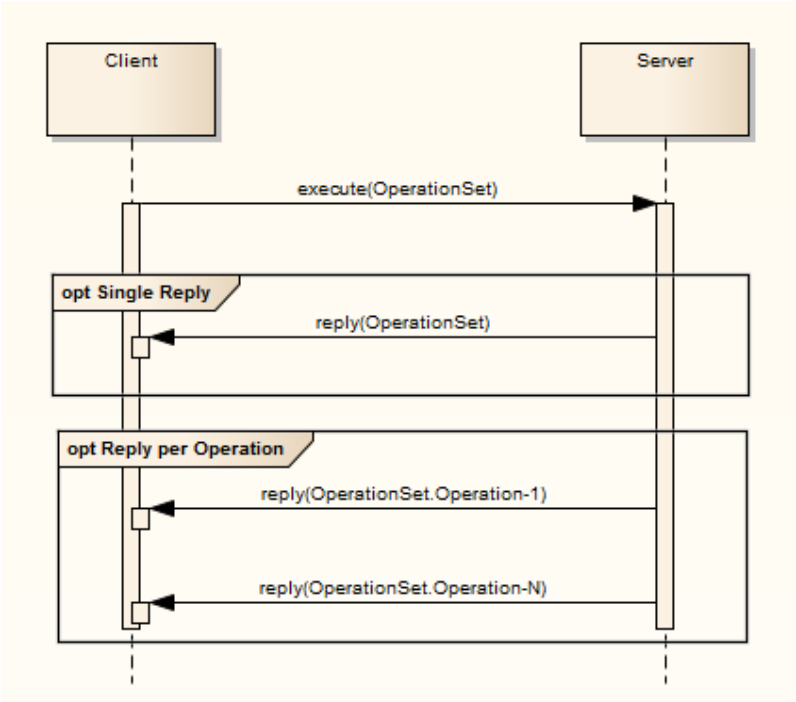


Figure 44 – Transactional Request/Response (OperationSet)

This request/response pattern can be used for any transaction involving an OperationSet. The verb in the message Header is always ‘execute’. The individual Operation(s) within the Operation set can have verbs and nouns consistent with the request / response transaction in Pattern 1. Depending upon the scenario, there can be multiple replies to a given execute / OperationSet transaction. For example, a single reply message can be sent for the entire OperationSet, or multiple reply messages can be sent, each with the reply data for one or more Operations in each message. The operationID element for each Operation in the request message is supplied in the reply message(s). This is used, in conjunction with the overall CorrelationID in the message Header(s) to correlate replies with their corresponding requests.

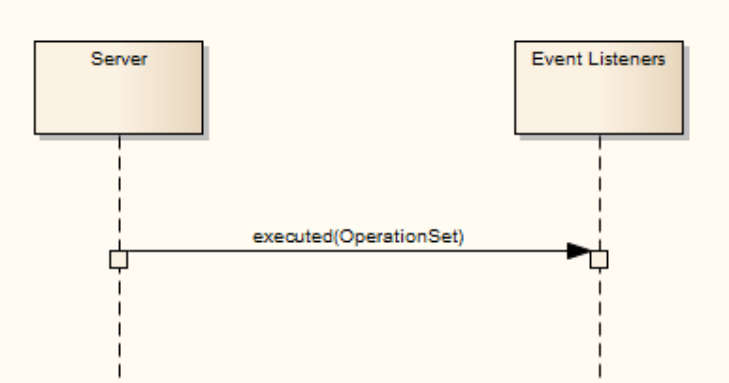


Figure 45 - Published Event (OperationSet)

This published event pattern can also be used for any transaction involving an OperationSet. The verb in the message Header is always “executed”. The individual Operation(s) within the Operation set can have verbs and nouns consistent with the published event transaction in Pattern 2. Using this pattern, an enterprise system may notify one or more other enterprise systems of OperationSet events without requiring any acknowledgment or confirmation of successful processing.

6.10.3 OperationSet Example

The following XML provides an example of a complex transaction that uses the Payload.OperationSet element.

```
<?xml version="1.0" encoding="UTF-8"?>
<RequestMessage
    xmlns = "http://iec.ch/TC57/2011/schema/message"
    xmlns:m = "http://iec.ch/TC57/2011/MeterConfig#"
    xmlns:up = "http://iec.ch/TC57/2011/UsagePointConfig#"
    xmlns:mdlc = "http://iec.ch/TC57/2011/MasterDataLinkageConfig#"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://iec.ch/TC57/2011/schema/message Message.xsd">
    <Header>
        <Verb>execute</Verb>
        <Noun>OperationSet</Noun>
        <Revision>2.0</Revision>
        <Timestamp>2012-12-20T09:30:47Z</Timestamp>
        <Source>CIS</Source>
        <AckRequired>true</AckRequired>
        <MessageID>D921A053-80C1-4DB6-960E-2603127B7B92</MessageID>
        <CorrelationID>D921A053-80C1-4DB6-960E-2603127B7B92</CorrelationID>
    </Header>
    <Payload>
        <OperationSet>
            <enforceMsgSequence>true</enforceMsgSequence>
            <enforceTransactionalIntegrity>true</enforceTransactionalIntegrity>
            <Operation>
                <operationId>1</operationId>
                <noun>MeterConfig</noun>
                <verb>create</verb>
                <mdlc:MeterConfig>
                    <mdlc:Meter>
                        <mdlc:formNumber>2S</mdlc:formNumber>
                        <mdlc:ConfigurationEvents>
```

```
                        <mdlc:createdDateTime>2012-12-
20T09:30:47Z</mdlc:createdDateTime>
                        <mdlc:effectiveDateTime>2012-12-
21T00:00:00Z</mdlc:effectiveDateTime>
                        <mdlc:Names>
                            <mdlc:name>C34531</mdlc:name>
                            <mdlc:NameType>
                                <mdlc:name>MeterBadgeNumber</mdlc:name>
                                <mdlc:NameTypeAuthority>
                                    <mdlc:name>UtilityXYZ</mdlc:name>
                                </mdlc:NameTypeAuthority>
                            </mdlc:NameType>
                        </mdlc:Names>
                    </mdlc:ConfigurationEvents>
                </mdlc:Meter>
            </mdlc:MeterConfig>
        </Operation>
        <Operation>
            <operationId>2</operationId>
            <noun>UsagePointConfig</noun>
            <verb>create</verb>
            <up:UsagePointConfig>
                <up:UsagePoint>
                    <up:amiBillingReady>amiCapable</up:amiBillingReady>
                    <up:connectionState>connected</up:connectionState>
                    <up:isSdp>true</up:isSdp>
                    <up:isVirtual>false</up:isVirtual>
                    <up:phaseCode>B</up:phaseCode>
                    <up:readCycle>ReadCycleJ</up:readCycle>
                    <up:ConfigurationEvents>
                        <up:createdDateTime>2012-12-
20T09:30:47Z</up:createdDateTime>
                        <up:effectiveDateTime>2012-12-
21T00:00:00Z</up:effectiveDateTime>
                    </up:ConfigurationEvents>
                    <up:Names>
                        <up:name>UP43639</up:name>
                        <up:NameType>
                            <up:name>ServiceDeliveryPointID</up:name>
                            <up:NameTypeAuthority>
                                <up:name>UtilityXYZ</up:name>
                            </up:NameTypeAuthority>
                        </up:NameType>
                    </up:Names>
                </up:UsagePoint>
            </up:UsagePointConfig>
        </Operation>
        <Operation>
            <operationId>3</operationId>
            <noun>MasterDataLinkageConfig</noun>
            <verb>create</verb>
            <mdlc:MasterDataLinkageConfig>
                <mdlc:ConfigurationEvent>
                    <mdlc:createdDateTime>2012-12-
17T09:30:47Z</mdlc:createdDateTime>
                    <mdlc:effectiveDateTime>2012-12-
21T00:00:00Z</mdlc:effectiveDateTime>
                </mdlc:ConfigurationEvent>
                <mdlc:Meter>
                    <mdlc:Names>
                        <mdlc:name>C34531</mdlc:name>
                        <mdlc:NameType>
                            <mdlc:name>MeterBadgeNumber</mdlc:name>
                            <mdlc:NameTypeAuthority>
                                <mdlc:name>UtilityXYZ</mdlc:name>
                            </mdlc:NameTypeAuthority>
                        </mdlc:NameType>
                    </mdlc:Names>
                </mdlc:Meter>
                <mdlc:UsagePoint>
                    <mdlc:Names>
                        <mdlc:name>UP43639</mdlc:name>
                        <mdlc:NameType>
                            <mdlc:name>ServiceDeliveryPointID</mdlc:name>
                            <mdlc:NameTypeAuthority>
                                <mdlc:name>UtilityXYZ</mdlc:name>
                            </mdlc:NameTypeAuthority>
                        </mdlc:NameType>
```

```
                    </mdlc:Names>
                </mdlc:UsagePoint>
            </mdlc:MasterDataLinkageConfig>
        </Operation>
    </OperationSet>
</Payload>
</RequestMessage>
```

The example complex transaction has three operations that do the following:

- Perform a ‘create MeterConfig’
- Perform a ‘create UsagePointConfig’
- Performs ‘create MasterDataLinkageConfig’

The XML identifies namespaces for MeterConfig, UsagePointConfig and MasterDataLinkageConfig as defined by IEC 61968-9.

6.11 **Representation of Time**

The ISO 8601 standard is used to define the representations of time values that are conveyed through interfaces. This avoids issues related to time zones and daylight savings time changes.

Timestamps in messages published by a server process should use a prevailing time, using the following example format: *2007-03-27T14:00:00-05:00* (as time changes from CDT to CST, the *-05:00* would change to *-06:00*).

Timestamps in messages sent by a client process could use any ISO 8601 compliant timestamp.

It is extremely important to note that the use of ISO 8601 timestamps within message definitions for the external interfaces defined by this document in no way constrains other representations of time that may include:

- User interfaces, where local time or market hours may be used as desired
- Reports, where reports would be generated using an appropriate local time
- Internal integration, where an application may internally require some other time structure

6.12 **Other Conventions and Best Practices**

The following are other conventions that must be followed by this specification:

- Within XML definitions, tags should be namespace qualified. For example, an XML tag of ‘<tag>’ should be prefixed by a specific namespace reference, e.g. ‘<ns:tag>’. This will help to eliminate ambiguity. *(Note that many examples in this document are not namespace qualified for brevity and to aid legibility)*
- Quantities should be expressed using SI units where appropriate.

6.13 **Technical Interoperability**

Open standards are a key part of the strategy to achieve technical interoperability. Standards of particular interest include:

- W3C standards
- OASIS WS-* standards
- IEC Common Information Model and related standards (e.g. IEC 61970-301 and IEC 61968-11)
- Java Message Service

It is very important that the implementation of Web Service interfaces not be dependent upon any specific proprietary, third party products. Another key requirement is that implementation of web service clients must be possible using both Java and .Net development tools.

6.14 **Service Level Agreements**

Different categories of services will have different service level agreements (SLAs). The SLAs for some services are directly impacted by the variability in the amount of data that can be transferred.

The response time periods specified for each interface covered by an SLA typically will vary to some degree, based upon factors such as network and system loading. Consequentially, each SLA should be stated in a manner such that each SLA will be honoured X% of the time where X is often in the range of 90-100%.

One use of SLAs is to identify timeout periods for request handling.

6.15 **Auditing, Monitoring and Management**

The ESB will typically have capabilities for auditing, monitoring and management. There may also be common services that are used for the implementation of integration components within the ESB. Example functionality would often include:

- Logging
- Generation of unique identifiers
- Generation of signatures
- User authentication and authorization
- Identification of on-line service instances (where there may be multiple instances)

7 Payload Specifications

Each noun used in a message identifies a payload type. Payload types are typically derived from the IEC CIM or other semantic models. Payload types used by the parts of IEC 61968 are always derived from the IEC CIM and have design artefacts (e.g. XSDs) that describe their structure. Cases where XSDs are not required include:

- Messages using RDF payloads as defined by IEC 61968-13 and IEC 61970-452.
- Messages using payloads as defined by IEC 61970-453.
- Response messages from services that dynamically generate XML (as in the case of SQL XML result sets).
- Non-XML compressed and encoded payloads.
- Encoded binary data (where XML formatting is not efficient as in the case of 'high speed data')

If an XSD is not available to describe the payload, it is the responsibility of the sender and receiver(s) to agree upon the specific formatting.

The CIM logical information model is described as a set of UML packages. The following diagram shows the use of the CIM from the perspectives of UML modelling and generation of design artefacts needed by integration tools. It illustrates the relationships between information models and contextual profiles that are used in conjunction with assembly rules in order to derive design artefacts.

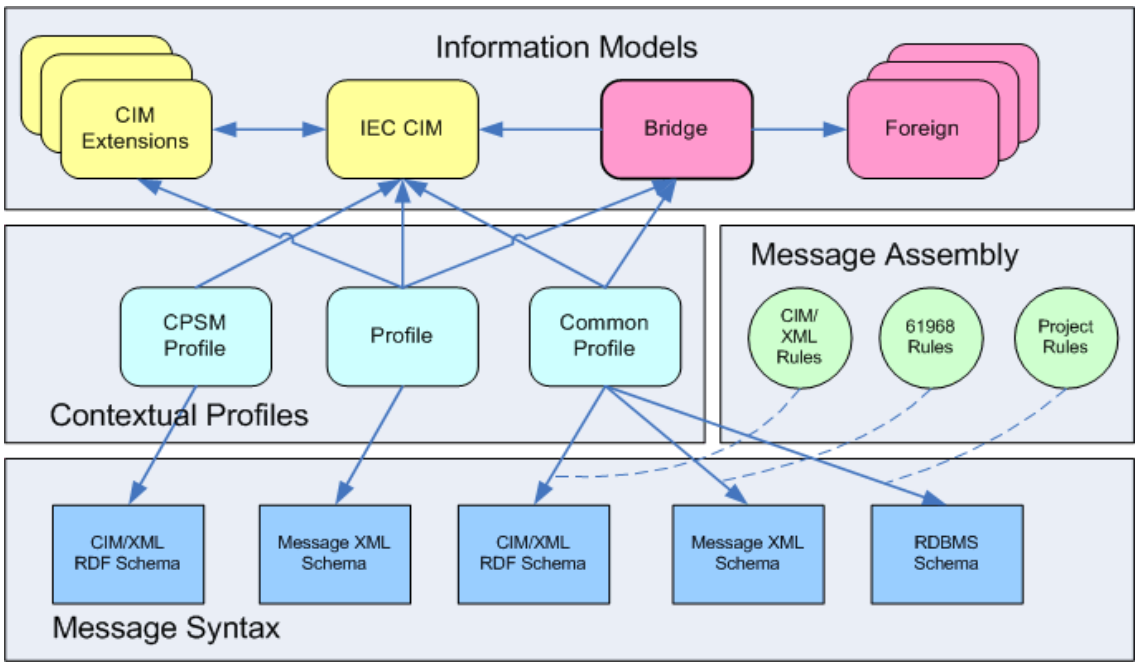


Figure 46 - Information Models, Profiles and Messages

The following figure shows an example contextual profile design within CIMTool (www.cimtool.org).

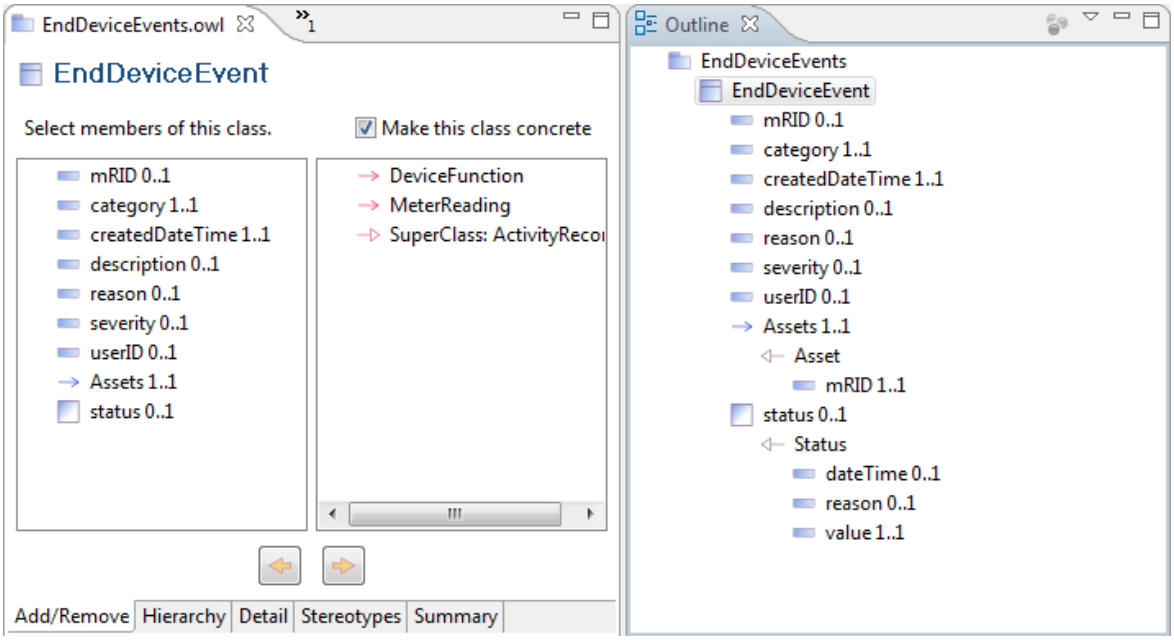


Figure 47 - Contextual Profile Design in CIMTool

When realizing a profile as a design artefact in the form of an XML Schema, it is important to recognize that there are many options related to the realization that may affect interoperability, these include:

- Use of namespaces
- Object references ‘by value’ or ‘by reference’
- Flat or hierarchical complex type definitions
- Required vs. Optional elements
- Enumerations

Another important point is that the noun must ***not*** be a name of a CIM UML class, otherwise it is not possible to have a valid XSD that includes that UML class in the profile/payload type.

The following diagram describes the structure of a simple example payload as described by an XML Schema that could be conveyed within the ‘any’ of the payload.

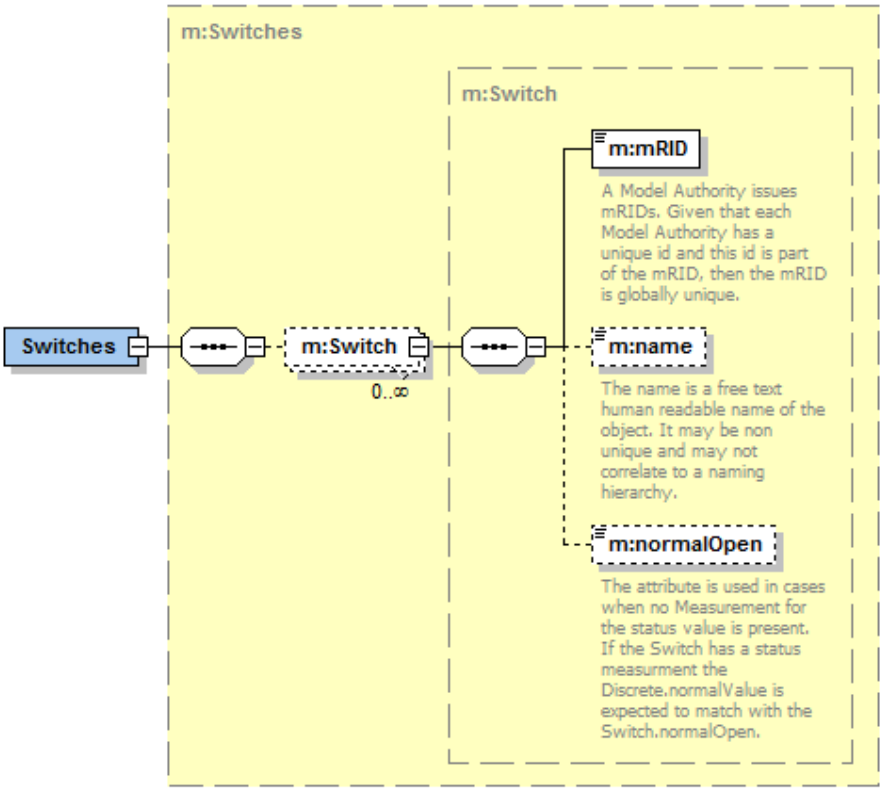


Figure 48 - Example Message Payload Schema

The example payload is described by the following XML Schema definition. XML Schemas for payloads can be generated in a variety of ways. One example is the use of CIMTool, where the CIM UML model is used as the domain model for the message definition. Note that the XML Schema for a message payload minimally defines a top level element.

The important point is that the name of the top level element must be the same as the noun that is used in the message header. In the following XSD the payload definition would be used in conjunction with the noun ‘Switches’.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:a="http://www.iec.ch/2008/Message#"
targetNamespace="http://iec.ch/TC57/2011/CIM-schema-cim12#"
elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns="http://iec.ch/TC57/2011/Message#" xmlns:m="http://iec.ch/TC57/2007/CIM-schema-
cim12#">
<xs:element name="Switches" type="m:Switches"/>
   <xs:complexType name="Switches">
      <xs:sequence>
         <xs:element name="Switch" type="m:Switch" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Switch">
      <xs:annotation>
         <xs:documentation>A generic device  designed to close, or open, or both, one
or more electric circuits.</xs:documentation>
      </xs:annotation>
      <xs:sequence>
         <xs:element name="mRID" minOccurs="1" maxOccurs="1" type="xs:string">
            <xs:annotation>
               <xs:documentation>A Model Authority issues mRIDs. Given that each Model
Authority has a unique id and this id is part of the mRID, then the mRID is globally
unique.</xs:documentation>
            </xs:annotation>
```

```
        </xs:element>
        <xs:element name="name" minOccurs="0" maxOccurs="1" type="xs:string">
           <xs:annotation>
              <xs:documentation>The name is a free text human readable name of the
object. It may be non unique and may not correlate to a naming
hierarchy.</xs:documentation>
            </xs:annotation>
        </xs:element>
        <xs:element name="normalOpen" minOccurs="0" maxOccurs="1" type="xs:boolean">
           <xs:annotation>
              <xs:documentation>The attribute is used in cases when no Measurement for
the status value is present. If the Switch has a status measurment the
Discrete.normalValue is expected to match with the
Switch.normalOpen.</xs:documentation>
           </xs:annotation>
        </xs:element>
     </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Figure 49 - Example Payload XML Schema

From the previous XML Schema, the following XML payload example is possible (as was used for examples in clause 6):

```
<m:Switches xsi:schemaLocation="http://www.iec.ch/TC57/2011/CIM-schema-cim12#
Switches.xsd" xmlns:m="http://www.iec.ch/TC57/2011/CIM-schema-cim12#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <m:Switch>
      <m:mRID>35378383838</m:mRID>
      <m:name>SW1</m:name>
      <m:normalOpen>true</m:normalOpen>
   </m:Switch>
   <m:Switch>
      <m:mRID>363482488448</m:mRID>
      <m:name>SW2</m:name>
      <m:normalOpen>true</m:normalOpen>
   </m:Switch>
   <m:Switch>
      <m:mRID>894094949444</m:mRID>
      <m:name>SW3</m:name>
      <m:normalOpen>false</m:normalOpen>
   </m:Switch>
</m:Switches>
```

Figure 50 - Example Message XML

Specific payload formats should be defined by an interface specification using XML Schemas, as are provided by IEC 61968 parts 3 through 9. For implementations outside the scope of IEC 61968, payload definitions can be defined as needed. In most cases the message noun takes a simple form such as ‘Switches’, ‘BidSets’, ‘TroubleTickets’ or ‘WorkOrders’. However it is also possible to use a prefix which identifies a message context in the following form:

```
<context><noun>
```

This would allow for stereotypes of the basic noun definition to be used to define additional restrictions appropriate for the message context. Examples would be ‘GetMeterReadings’ and ‘GetEndDeviceAssets’.

8 Interface Specifications

8.1 General

This purpose of this clause is to describe interface definitions. There are three perspectives provided here:

- What is needed by a user of the interface to supplement the information provided by a specific definition language or design artefacts
- Web services artefacts and implementation details
- JMS implementation details

8.2 Application-Level Specifications

Specific interfaces are defined using a sequence of specific combinations of verbs and nouns (i.e. payload types). For example a request message with verb and noun of 'get MeterReadings' would result in a response message of 'reply MeterReadings'. Given the potential complexity and options available for a given integration, the details of the interactions should be further documented. Application-level specifications based upon IEC 61968-100 will often require more detailed specifications that are typically beyond the capabilities of XML schemas and WSDLs. The following are simple examples of the documentation for messages that might be provided by an application-level specification.

The messages for a request would use the following message fields:

Message Element	Value
Header/Verb	get
Header/Noun	*Name of payload type*
Header/Source	*System or application initiating request*
Header/UserID	*Optional: ID of user*
Request/?	*Optional: Other request parameters may be specified as needed*

The corresponding response messages would use the following message fields:

Message Element	Value
Header/Verb	reply
Header/Noun	*Defined payload type name*
Reply/result	*Reply code, success=OK, partial success=PARTIAL, error=FAILED*
Reply/Error	*Optional: May be any number of error messages*
Payload	*Defined payload type*

In the cases of payloads that would otherwise be very large (as an example, over some threshold such as 1 megabyte), the payloads would be zipped, base64 encoded and stored within the 'Payload/Compressed' tag.

Specific nouns, verbs (as defined in Annex B) and payload formats should be defined by an interface specification, as are provided by IEC 61968 parts 3-9. Use case sequence diagrams are also commonly used to describe information exchange patterns in terms of verbs and nouns.

A more thorough description of the usage of an interface, potentially as part of a more complex business process would be described using a sequence diagram. The following sequence diagram provides an example of information exchange using verbs and nouns. The diagram convention uses '<verb>(<Noun>)' for each flow between components.

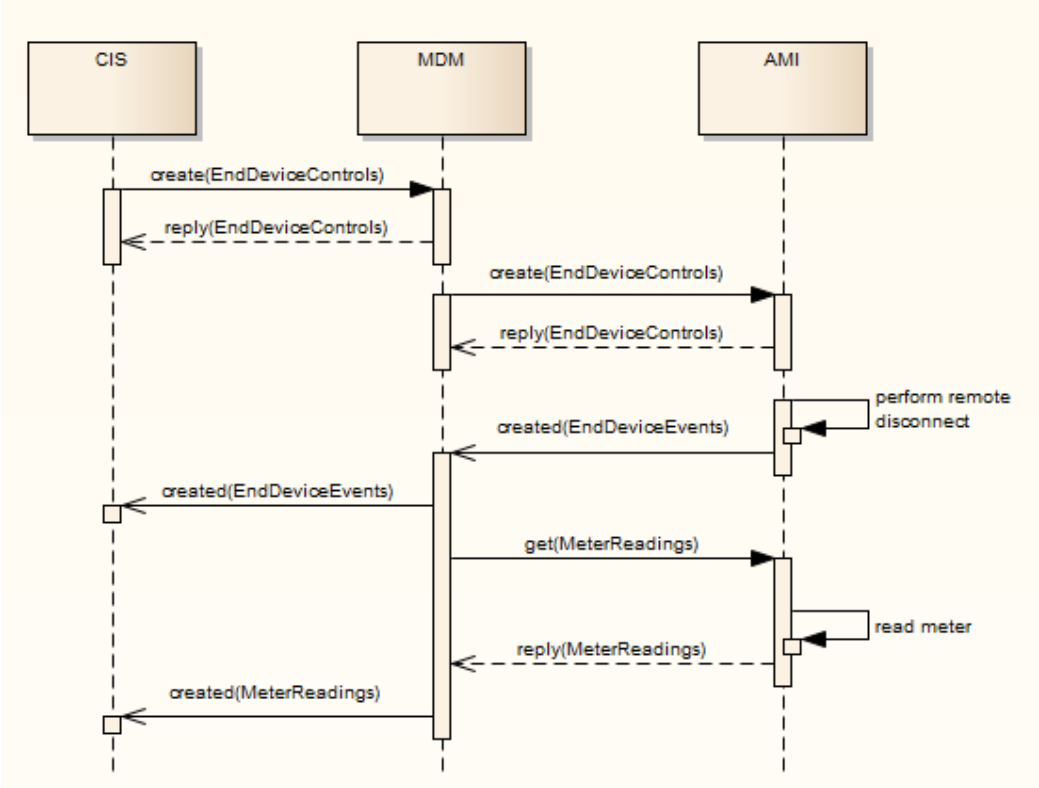


Figure 51 - Example Complex Business Process

The above figure is a sequence diagram that describes a complex business process that combines several different types of messages and integration patterns.

8.3 Web Service Interfaces

8.3.1 General

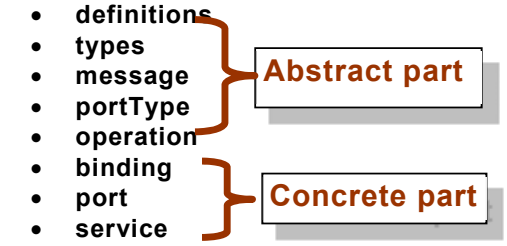
This sub-clause describes the definition of interfaces using web services. This describes the use of a document-wrapped style which maximizes interoperability. This also prescribes operations which are names using verb/noun combinations, with type-specific payload definitions.

There are two approaches described by this standard for web services:

- Strongly-typed, with procedure for WSDL generation defined in detail in Annex C
- Generic web services, where a generic WSDL is described in Annex D

8.3.2 WSDL Structure

Typically a WSDL (v1.1) is made of two parts with following tags.



WSDL example document can be found in WSDL template in Annex 4.
For strongly-typed web services, the web service design practices are summarized below:

- Standard SOAP binding is used.
- XSD as data type is typically imported instead of being embedded for better version control
- Wire signature issue is avoided by redefining element names such as CreateEndDeviceControl and ChangeEndDeviceControl using a single XSD complexType
- Wrapped Document style is used
- Operation name follows the Verb + Noun naming convention which allows avoiding contend-based routing

8.3.3 Document Style SOAP Binding

The document style using SOAP body is the most common practice in WSDL design. It can fully utilize the benefits of an XML schema for payload validation. Below is an example of the binding section in a Document style WSDL for the EndDeviceControl information exchange:

```
<wsdl:binding name="EndDeviceControls_Binding" type="tns:EndDeviceControls_Port">
   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

   <wsdl:operation name="CreateEndDeviceControls">
      <soap:operation
soapAction="http://iec.ch/TC57/2010/EndDeviceControls/CreateEndDeviceControls"
style="document"/>
      <wsdl:input name="CreateEndDeviceControlsRequest">
         <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="CreateEndDeviceControlsResponse">
         <soap:body use="literal"/>
      </wsdl:output>
      <wsdl:fault name="CreateEndDeviceControlsFault">
         <soap:fault name="CreateEndDeviceControlsFault" use="literal"/>
      </wsdl:fault>
   </wsdl:operation>
…
```

Note that both <soap:binding> and <soap:operation> styles are defined as “document” and are highlighted in gray. Also <soap:body> is used for both input and output operations. A “document” style means an XML document is included in a soap message. In this case, it is directly placed in the <soap:body>.

If a wsdl:*operation* name is the same as the input element name, this WSDL becomes a wrapped document style WSDL. Wrapped document style originates from Microsoft to mimic a RPC style. In a RPC style, a payload is always wrapped by its operation name.

The characteristics of the wrapped pattern are listed below:

- The input message has a single *part*
- The *part* is an element
- The element has the same name as the *operation*
- The element's complex type has no attributes.

Any WSDL that does not meet the criterion above is an unwrapped WSDL. There are pros and cons for both wrapped and unwrapped patterns, but wrapped document style is recommended in this profile for the sake of interoperability.

Here is a sample WSDL on the wrapped document style:

```
  … …

<wsdl:message name="CreateEndDeviceControlsRequestMessage">
   <wsdl:part name="CreateEndDeviceControlsRequestMessage"
element="message:CreateEndDeviceControls"/>
</wsdl:message>
… …

<wsdl:portType name="EndDeviceControls_Port">

   <wsdl:operation name="CreateEndDeviceControls">
      <wsdl:input name="CreateEndDeviceControlsRequest"
message="tns:CreateEndDeviceControlsRequestMessage"/>
      <wsdl:output name="CreateEndDeviceControlsResponse"
message="tns:ResponseMessage"/>
      <wsdl:fault name="CreateEndDeviceControlsFault" message="tns:FaultMessage"/>
   </wsdl:operation>
   … …
   </wsdl:operation>
</wsdl:portType>
```

8.3.4 Strongly-typed Web Services

The strongly-typed web service integration pattern is intended for use to implement semantic-based interfaces in support of a SOA integration strategy. The strongly-typed pattern has the following characteristics:

1. Uses SOAP-based web services, where fine-grained WSDL's are used to define a contract.
2. Enables stronger payload validation by defining operation messages using strongly typed payloads.

8.3.4.1 Service and Operation Naming (Normative)

In the IEC 61968-100 strongly-typed web service implementation, the following service names are used to reflect the role of the service in the enterprise:

- **Send**
  To provide (send) information (business object) for public (enterprise) consumption. To be invoked by the system of record for the business object and only when the state of the business object has changed. This is used in conjunction with the verbs created, changed, closed, canceled and deleted.

- **Receive**
  To consume (receive) information (business object) from an external source. This is used in conjunction with the verbs created, changed, closed, canceled and deleted.

- **Request**
  To request another party to perform a specific service. This is used in conjunction with the verbs get, create, change, close, cancel and delete.

- **Execute**

To run a service provided to the public, which may include a state change request or a query request. This is used in conjunction with the verbs create, change, close, cancel and delete.

- **Reply**
  To reply with the result of the execution of a service (by the Execute service). This is used in conjunction with the verbs created, changed, closed, canceled and deleted.

- **Show**
  To provide (show) information (business object) for public (enterprise) consumption, when the state of the business object is not changed, by the system of record or other system that has a copy of the same business object.

- **Retrieve**
  To request specific data of a business object to be provided.

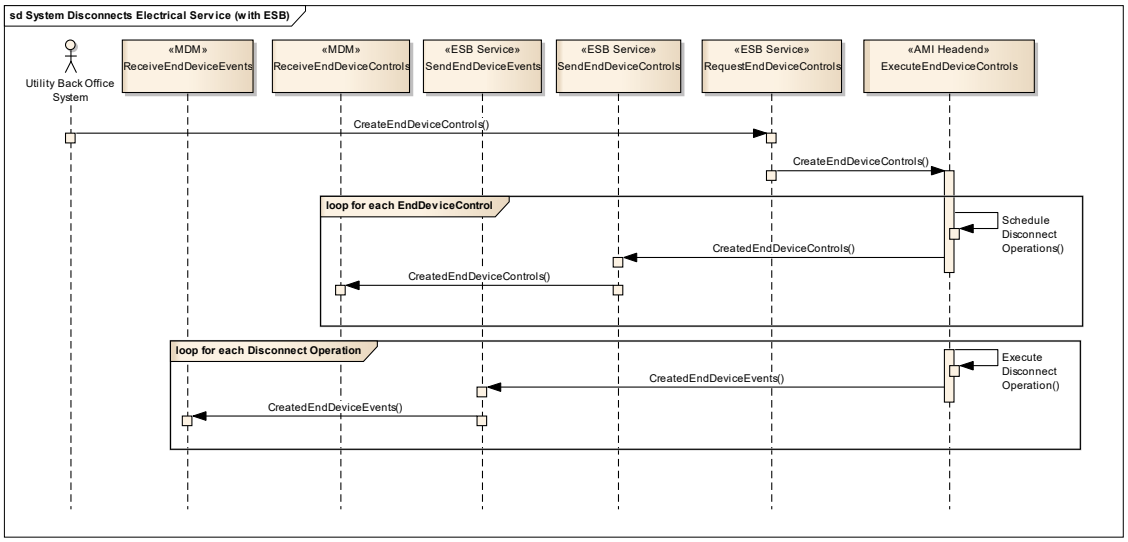
Using the service name and operation patterns, information objects and verbs, a service/operation naming convention for strongly-typed web services is described as below:

- **Service name**:
  To follow ***<Service pattern name>+<Information Object>*** such as ExecuteEndDeviceControls

- **Operation name**:
  To follow ***<Verb>+<Information Object>*** such as CreatedEndDeviceControls

8.3.4.2 Strongly-typed Web Service Integration Example

Below is an example usage of the strongly-typed web services to implement connect/disconnect functionality between an MDM and AMI system utilizing an ESB.



1. Meter Data Management System
    a. Services Implemented
        i. ReceiveEndDeviceControls: processes event stereotype messages (notifications) of EndDeviceControls activity.
        ii. ReceiveEndDeviceEvents: processes event stereotype messages (notifications) of EndDeviceEvent activity.
2. Enterprise Service Bus
    a. Services Implemented
        i. RequestEndDeviceControls: processes requests to perform some activity with EndDeviceControls. Routes to appropriate endpoint(s) for execution.
        ii. ReplyEndDeviceControls: processes replys regarding event stereotype messages (notifications) of EndDeviceControls activity. Publishes to the appropriate endpoint(s).
        iii. SendEndDeviceEvents: processes event stereotype messages (notifications) of EndDeviceEvent activity. Publishes to the appropriate endpoint(s).
3. AMI System
    a. Services Implemented
        i. ExecuteEndDeviceControls: acts on (executes) sets of EndDeviceControls.

8.4 JMS

8.4.1 General

This subclause describes the use of JMS. Messages communicated using JMS will use topics and/or queues. The differences and similarities between topics and queues are summarized as follows:

- Topics are used when the destination of a message is potentially more than one process
- Queues are used when the destination of a message is at most one process

- If supported by the JMS provider, topics and queues may be organized and named hierarchically
- Except for the use of a durable subscription, a process can only receive a copy of a message published to a topic if it is running and has an active subscription
- Message published to queues will remain on the queue until de-queued by a receiving process (noting that there may be options for expiration and queue persistence by the specific JMS implementation)
- A Queue is in effect a special case of a durable topic subscription, where only one process consumes a message.

8.4.2 Topic and Queue Naming

When naming a topic or queue, the top-level should identify ‘context’. Examples of context can be PRODUCTION, TESTING, DEVELOPMENT or TRAINING. The purpose is to insure that messages of different contexts are logically segregated if not physically segregated. For example, it is critical that there is no opportunity for a message related to a TRAINING activity to be injected into a PRODUCTION activity. The use of both physical and logical segregation is desirable.

The following diagram describes a possible organization of topics and queues. This organisation is an example only and not normative.

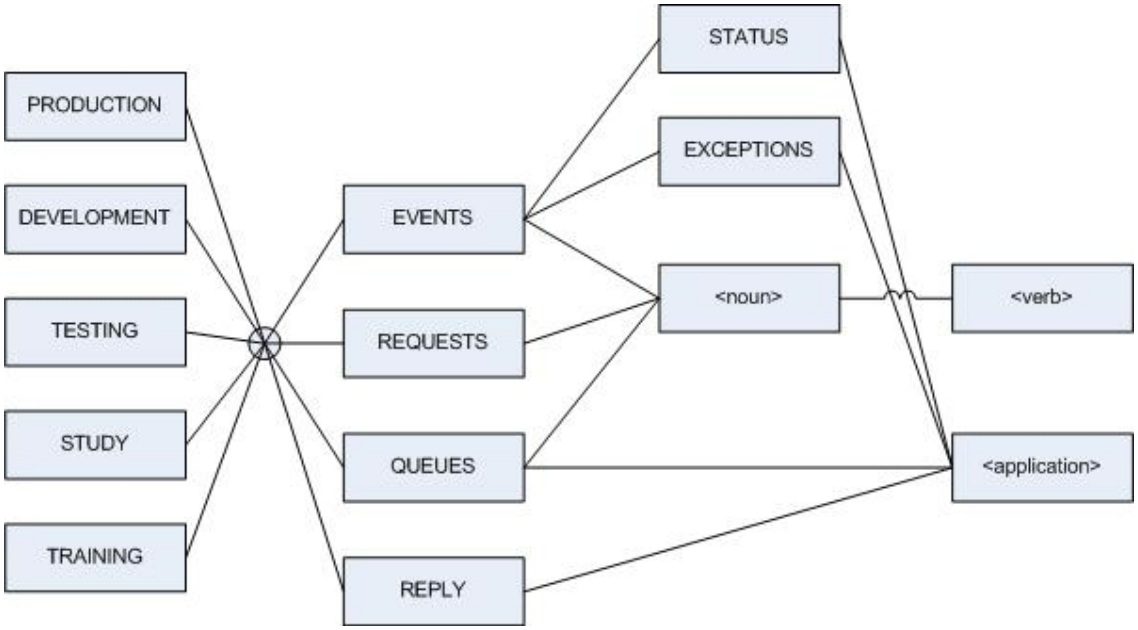


Figure 52 - Example Organization of Topics and Queues

This topic/queue organization would result in names such as:

- PRODUCTION.EVENTS.STATUS
- PRODUCTION.EVENTS.BidSet.created
- PRODUCTION.REQUESTS.BidSet.create
- STUDY.EVENTS.Contingency.created

It is important to note that JMS implementations typically allow for the use of wild cards in subscriptions. The intent of the described organization is meant to allow that feature to be leveraged. Additionally, it is possible to extend the topic definitions to provide for more granular

subscriptions. For example, a topic in the form <context>.EVENTS.<Noun> could be augmented to include a specific object ID for a specific project implementation.

8.4.3 JMS Message Fields

It is also important to note that some JMS header fields are related to field in the 61968 message header. The following table shows where some JMS header fields may be mapped to 61968-100 header fields as a best practice, but it is not required.

JMS Header Field	Set By	61968-100 Header Field
JMSDestination	send or publish method	NA
JMSDeliveryMode	send or publish method	NA
JMSExpiration	send or publish method	NA
JMSPriority	send or publish method	NA
JMSMessageID	send or publish method	MessageID
JMSTimestamp	send or publish method	TimeStamp
JMSCorrelationID	Client	CorrelationID
JMSReplyTo	Client	ReplyAddress
JMSType	Client	NA
JMSRedelivered	JMS provider	NA

9 Security

Security is a key issue for most implementations. Security requirements may be different depending upon the specific integration scenario. Some of the different example scenarios include:

- Intra-application integration of components within a controlled environment
- Inter-application integration within a controlled environment
- Inter-application integration across an enterprise
- Business-to-business integration between trusted partners using a trusted infrastructure
- Extra-enterprise integration, including enterprise application to device integration
- Publicly accessible services

There are many approaches and mechanisms that can be employed, depending upon the requirements.

The use of a SOAP envelope (which can be used with JMS as well as web services) provides benefits, where many products will leverage SOAP Headers for security purposes.

In cases where messages use a public network, security is a significant concern, although there are other situations where security can be a significant concern. Security can include authentication, authorization, encryption and non-repudiation. The details of the implementation of security are outside of the scope of this standard.

There are two basic steps in securing messaging interactions. First, the transport layer is secured. The second step is to secure the message itself. The transport layer is typically secured through the use of Secure Socket Layer (SSL) and Transport Layer Security (TLS). Besides creating a secure communication channel between a client and a service, message exchanges require that security information be embedded within the message itself. This is often the case when a message needs to be processed by several intermediary nodes before it reaches the target service or when a message must be passed among several services to be processed.

It is important to note that message-level security is very useful in XML document-centric applications, since different sections of the XML document may have different security requirements or be intended for different users.

10 Version Control

It is important to recognize that new versions of interfaces may be provided over time, largely as a consequence of:

- Staging of initial implementation
- New requirements
- Upgrades to vendor products

Wherever possible, interfaces will be evolved through augmentation, where a newer version of an interface is compatible with a previous version of an interface. However, this will not always be possible. New versions of interfaces will be manifested by:

- Changes to WSDLs
- Changes to XML Schemas
- Changes to software implementations

In line with OASIS guidelines for namespaces, it is strongly desirable preserve namespaces, especially when definitions have backward compatibility. New namespaces should only be created when a definition cannot be backwards compatible. There are two types of updates in terms of version control:

- Major version update:
  In this case major update has been made in an XSD and its backward compatibility has been broken as a result.
- Minor version update:
  In this case backward compatibility is intact. One example of such minor update is a new element added but as an optional field.

A naming convention for version control of message payloads is proposed here to use XSD targetNamespace, version attribute, and annotation as below:

- targetNamespace="http://iec.ch/TC57/**yyyy/<Payload Type Name>**"
- version="**<Major version>.<Minor version>**".
- Annotation added for detail description such as "Version 1.0 created in 2009/02".

Here are two examples for major and minor XSD updates, respectively.

In this example a 2009/02 version has a major update, its targetNamespace and version can be changed from:

```
<xs:schema ... targetNamespace="http://iec.ch/TC57/2010/EndDeviceControls"
version="1.0">
   <xs:annotation>
      <xs:documentation>
         Major version 1.0 created in 2010/11
      </xs:documentation>
   </xs:annotation>
```

To

```
<xs:schema ... targetNamespace="http://iec.ch/TC57/2011/EndDeviceControls"
version="2.0">
   <xs:annotation>
      <xs:documentation>
         Major version 2.0 created in 2011/03
      </xs:documentation>
   </xs:annotation>
```

However if an update is minor, its targetNamespace and version can be changed as follows, from:

```
<xs:schema ... targetNamespace="http://iec.ch/TC57/2010/EndDeviceControls"
version="1.0">
   <xs:annotation>
      <xs:documentation>
         Major version 1.0 created in 2010/11
      </xs:documentation>
   </xs:annotation>
```

To the following example with a minor version:

```
<xs:schema ... targetNamespace="http://iec.ch/TC57/2010/EndDeviceControls"
version="1.1">
   <xs:annotation>
      <xs:documentation>
         Major version 1.0 created in 2010/11
Minor version 1.1 created in 2010/12
      </xs:documentation>
   </xs:annotation>
```

The “version” attribute does not apply to XML validation against an XSD so its content change (2nd example, minor change) does not break validation against previous XSD version.

For versioning of Message.xsd, similar rules would apply.

Annex A (Normative): XML Schema for Common Message Envelope

The following XML schema is used to define a common message envelope (CME) for request, response and event messages as referenced throughout this standard.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Common Message Specification for IEC 61968 -->
<!--            Change Log                      -->
<!-- 2010/12/15 Added OperationSet to Payload   -->
<!-- 2011/03/09 Corrected FaultMessageType      -->
<!-- 2011/03/09 Baseline for version control    -->
<!-- 2011/03/10 Created type definitions for OperationSet and Operation to improve
compatibility with SoapUI  -->
<!-- 2011/05/06 Removed deprecated verbs, added 'executed'  -->
<!-- 2011/05/06 Changed base namespace to follow WG14 convention of 'iec.ch'  -->
<!-- 2012/02/10 Added relatedObject to Error element  -->
<!-- 2012/02/11 Created a new ObjectType for use in Error element  -->
<!-- 2012/02/11 Removed enumeration for Header.Context  -->
<!-- 2012/02/12 Added note that Error.object.Name elements are deprecated  -->
<!-- 2012/02/12 Added more comments to message elements  -->
<!-- 2012/02/16 Corrected comment for Reply.Error.level  -->
<!-- 2012/02/16 Revised comment for Reply.Error.code  -->
<!-- 2012/02/22 Added ID to Payload for optional use by close/cancel/delete  -->
<!-- 2012/02/22 Extended ID elements to have attributes for idType, idAuthority,
iSmRID -->
<!-- 2012/02/22 Extended ErrorType elements to use ID and relatedID elements, with
deprecation of object -->
<!-- 2012/02/24 Added kind attribute to ID elements in place of iSmRID -->
<!-- 2012/03/19 Corrected ID and relatedID definitions in ErrorType -->
<!-- 2012/03/20 Revised ID elements to use an attribute group -->
<!-- 2012/03/21 Corrected Payload.ID elements -->
<!-- 2012/04/03 Corrected Reply.Error.object.Name -->
<!-- 2012/04/03 Corrected Header.User.Organization made optional -->
<!-- 2012/06/08 Updated IDatts attribute group to include objectType attribute as
string -->
<!-- 2012/10/14 corrections and revisions to annotations for FDIS -->
<xs:schema xmlns="http://iec.ch/TC57/2011/schema/message"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://iec.ch/TC57/2011/schema/message"
elementFormDefault="qualified" attributeFormDefault="unqualified" version="1.0.0">
  <xs:complexType name="RequestType">
    <xs:annotation>
      <xs:documentation>Request type definition</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:annotation>
        <xs:documentation>Request package is typically used to supply parameters for
'get' requests</xs:documentation>
      </xs:annotation>
      <xs:element name="StartTime" type="xs:dateTime" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Start time of interest</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="EndTime" type="xs:dateTime" minOccurs="0">
        <xs:annotation>
          <xs:documentation>End time of interest</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Option" type="OptionType" minOccurs="0" maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Request type specialization</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="ID" minOccurs="0" maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Object ID for request</xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:string">
              <xs:attributeGroup ref="IDatts"/>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
```

```
      </xs:element>
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>This can be a CIM profile defined as an XSD with a CIM-
specific namespace This may also be used for custom extensions.</xs:documentation>
        </xs:annotation>
      </xs:any>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ReplyType">
    <xs:annotation>
      <xs:documentation>Reply type definition</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:annotation>
        <xs:documentation>Reply package is used to confirm success or report
errors</xs:documentation>
      </xs:annotation>
      <xs:element name="Result">
        <xs:annotation>
          <xs:documentation>Reply code: OK, PARTIAL or FAILED</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="OK"/>
            <xs:enumeration value="PARTIAL"/>
            <xs:enumeration value="FAILED"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="Error" type="ErrorType" minOccurs="0" maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Reply details describing one or more
errors</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="ID" minOccurs="0" maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Resulting transaction ID (usually consequence of
create)</xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:string">
              <xs:attributeGroup ref="IDatts"/>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Used for custom extensions</xs:documentation>
        </xs:annotation>
      </xs:any>
      <xs:element name="operationId" type="xs:integer" minOccurs="0">
        <xs:annotation>
          <xs:documentation>The reply.operationId provides the unique identifier of
the Operation for which this reply.result is relevant. Thus, it is  assumed that this
is a partial reply in direct response to one of the operations contained in an
OperationSet request.</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="PayloadType">
    <xs:annotation>
      <xs:documentation>Payload container</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:choice>
        <xs:any namespace="##other" processContents="skip" minOccurs="0"
maxOccurs="unbounded">
          <xs:annotation>
            <xs:documentation>For XML payloads, usually CIM profiles defined using an
XSD in a profile-specific namespace. May also be used for custom
extensions.</xs:documentation>
```

```
          </xs:annotation>
        </xs:any>
        <xs:element name="OperationSet" type="OperationSet" minOccurs="0">
          <xs:annotation>
            <xs:documentation>Each operation set is a collection of operations that
may require operational-integrity and/or sequence control.</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="Compressed" type="xs:string" minOccurs="0">
          <xs:annotation>
            <xs:documentation>For compressed and/or binary, uuencoded payloads If
compressed, Gzip compression is used.</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="ID" minOccurs="0" maxOccurs="unbounded">
          <xs:annotation>
            <xs:documentation>Used to supply object IDs for cancel/close/delete
operations in cases where they are not otherwise specified using a type-specific
payload</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attributeGroup ref="IDatts"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:choice>
      <xs:element name="Format" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Hint as to format of payload, e.g. XML, RDF, SVF, BINARY,
PDF, ...</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="OperationType">
    <xs:annotation>
      <xs:documentation>For master data set synchronization XML
payloads.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="operationId" type="xs:integer">
        <xs:annotation>
          <xs:documentation>The payload.operation.operationId provides the unique
identifier (within the OperationSet) of the Operation for the purpose of reference in
subsequent messages (e.g. OperationSet reply).</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="noun" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>The payload.operation.##other also identifies the noun,
this element is optionally supplied to simplify processing.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="verb" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>"create", "delete", "change", etc.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="elementOperation" type="xs:boolean" default="false"
minOccurs="0">
        <xs:annotation>
          <xs:documentation>TRUE if the verb is operating at the element level. In
such a case, the verb is to be applied to the elements populated in the
payload.operation.##other below. If omitted, assume FALSE.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:any namespace="##other" processContents="skip" minOccurs="0">
        <xs:annotation>
          <xs:documentation>An XML payload which carries a CIM profile defined using
an XSD in a profile-specific namespace. Individual payloads are used collectively to
create a series of related operations. See the "enforce" boolean flags in the header
for instructions on how to process these messages.</xs:documentation>
        </xs:annotation>
      </xs:any>
    </xs:sequence>
```

```
  </xs:complexType>
  <xs:complexType name="OperationSet">
    <xs:annotation>
      <xs:documentation>Each operation set is a collection of operations that may
require operational-integrity and/or sequence control.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="enforceMsgSequence" type="xs:boolean" minOccurs="0">
        <xs:annotation>
          <xs:documentation>If set to TRUE, the Operation.##other messages must be
processed in the sequence presented. If omitted, assume FALSE.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="enforceTransactionalIntegrity" type="xs:boolean"
minOccurs="0">
        <xs:annotation>
          <xs:documentation>Set to TRUE when all of the Operation.##other messages
must be processed successfully or else the entire message set must be rolled back. If
omitted, assume FALSE.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Operation" type="OperationType" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ReplayDetectionType">
    <xs:annotation>
      <xs:documentation>Used to detect and prevent replay attacks</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Nonce" type="xs:string"/>
      <xs:element name="Created" type="xs:dateTime"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="UserType">
    <xs:annotation>
      <xs:documentation>User type definition</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="UserID" type="xs:string">
        <xs:annotation>
          <xs:documentation>User identifier</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Organization" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>User parent organization identifier</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="HeaderType">
    <xs:annotation>
      <xs:documentation>Message header type definition</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:annotation>
        <xs:documentation>Message header contains control and descriptive information
about the message.</xs:documentation>
      </xs:annotation>
      <xs:element name="Verb">
        <xs:annotation>
          <xs:documentation>This enumerated list of verbs that can be used to form
message types in compliance with the IEC 61968 standard.</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="cancel"/>
            <xs:enumeration value="canceled"/>
            <xs:enumeration value="change"/>
            <xs:enumeration value="changed"/>
            <xs:enumeration value="create"/>
            <xs:enumeration value="created"/>
            <xs:enumeration value="close"/>
            <xs:enumeration value="closed"/>
            <xs:enumeration value="delete"/>
            <xs:enumeration value="deleted"/>
            <xs:enumeration value="get"/>
```

```
            <xs:enumeration value="reply"/>
            <xs:enumeration value="execute"/>
            <xs:enumeration value="executed"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="Noun" type="xs:string">
        <xs:annotation>
          <xs:documentation>The Noun of the Control Area identifies the main subject
of the message type, typically a real world object defined in the
CIM.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Revision" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Revision level of the message type.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="ReplayDetection" type="ReplayDetectionType" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Use to introduce randomness in the message to enhance
effectiveness of encryption</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Context" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Intended context for information usage, e.g. PRODUCTION,
TESTING, TRAINING, ...</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Timestamp" type="xs:dateTime" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Application level relevant time and date for when this
instance of the message type was produced.  This is not intended to be used by
middleware for message management.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Source" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Source system or application that sends the
message</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="AsyncReplyFlag" type="xs:boolean" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Indicates whether or not reply should be
asynchronous</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="ReplyAddress" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Address to be used for asynchronous replies, typically a
URL/topic/queue.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="AckRequired" type="xs:boolean" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Indicates whether or not an acknowledgement is
required</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="User" type="UserType" minOccurs="0">
        <xs:annotation>
          <xs:documentation>User information of the sender</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="MessageID" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Unique message ID to be used for tracking
messages</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="CorrelationID" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>ID to be used by applications for correlating
replies</xs:documentation>
        </xs:annotation>
      </xs:element>
```

```
      <xs:element name="Comment" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Optional comment</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Property" type="MessageProperty" minOccurs="0"
maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Message properties can be used to identify information
needed for extended routing and filtering capabilities</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Used to allow custom extensions</xs:documentation>
        </xs:annotation>
      </xs:any>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Message" type="MessageType">
    <xs:annotation>
      <xs:documentation>Common IEC 61968 Message Definition</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:complexType name="MessageProperty">
    <xs:annotation>
      <xs:documentation>Message properties can be used for extended routing and
filtering</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Value" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="RequestMessage" type="RequestMessageType">
    <xs:annotation>
      <xs:documentation>Request message structure</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="ResponseMessage" type="ResponseMessageType">
    <xs:annotation>
      <xs:documentation>Response message structure</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="EventMessage" type="EventMessageType">
    <xs:annotation>
      <xs:documentation>Event message structure. </xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:complexType name="MessageType">
    <xs:annotation>
      <xs:documentation>Generic Message Type</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Header" type="HeaderType"/>
      <xs:element name="Request" type="RequestType" minOccurs="0"/>
      <xs:element name="Reply" type="ReplyType" minOccurs="0"/>
      <xs:element name="Payload" type="PayloadType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="RequestMessageType">
    <xs:annotation>
      <xs:documentation>Request Message Type, which will typically result in a
ResponseMessage to be returned. This is typically used to initiate a transaction or a
query request.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Header" type="HeaderType"/>
      <xs:element name="Request" type="RequestType" minOccurs="0"/>
      <xs:element name="Payload" type="PayloadType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ResponseMessageType">
    <xs:annotation>
      <xs:documentation>Response MessageType, typically used to reply to a
RequestMessage</xs:documentation>
    </xs:annotation>
```

```
    <xs:sequence>
      <xs:element name="Header" type="HeaderType"/>
      <xs:element name="Reply" type="ReplyType"/>
      <xs:element name="Payload" type="PayloadType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="FaultMessageType">
    <xs:annotation>
      <xs:documentation>Fault Message Type, which is used in cases where the incoming
message (including the header) cannot be parsed</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Reply" type="ReplyType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="EventMessageType">
    <xs:annotation>
      <xs:documentation>Event Message Type, which is used to indicate a condition of
potential interest. Note that the Payload may be required in the
future.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Header" type="HeaderType"/>
      <xs:element name="Payload" type="PayloadType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ErrorType">
    <xs:annotation>
      <xs:documentation>Error Structure</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="code" type="xs:string">
        <xs:annotation>
          <xs:documentation>Defined error code, as defined by IEC 61968-100, related
standards or local implementation</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="level" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Severity level, e.g. INFORM, WARNING, FATAL,
CATASTROPHIC</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="INFORM"/>
            <xs:enumeration value="WARNING"/>
            <xs:enumeration value="FATAL"/>
            <xs:enumeration value="CATASTROPHIC"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="reason" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Description of the error</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="details" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Free form detailed text description of
error</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="xpath" type="xs:QName" minOccurs="0">
        <xs:annotation>
          <xs:documentation>XPath expression to identify specific XML
element</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="stackTrace" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Stack trace as generated by software upon
exception</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Location" type="LocationType" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Location of exception within software</xs:documentation>
        </xs:annotation>
```

```
      </xs:element>
      <xs:element name="ID" minOccurs="0">
        <xs:annotation>
          <xs:documentation>ID of object</xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:string">
              <xs:attributeGroup ref="IDatts"/>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="relatedID" minOccurs="0">
        <xs:annotation>
          <xs:documentation>ID of related object, used in cases where there is an
error between the relationship of two objects</xs:documentation>
        </xs:annotation>
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:string">
              <xs:attributeGroup ref="IDatts"/>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="object" type="ObjectType" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Deprecated</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="operationId" type="xs:integer" minOccurs="0">
        <xs:annotation>
          <xs:documentation>The reply.operationId provides the unique identifier of
the Operation for which this reply.result.error is relevant. Thus, it is  assumed that
this is an error from one of the operations contained in an OperationSet
request.</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="OptionType">
    <xs:annotation>
      <xs:documentation>Request options</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="value" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LocationType">
    <xs:annotation>
      <xs:documentation>Process location where error was
encountered</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="node" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Name of the pipeline/branch/route node where error
occurred</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="pipeline" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Name of the pipeline where error occurred (if
applicable)</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="stage" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Name of the stage where error occurred (if
applicable)</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ObjectType">
    <xs:annotation>
```

```
      <xs:documentation>Used to identify an object of interest</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="mRID" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>A UUID-based name for the object</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Name" type="Name" minOccurs="0" maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>The Name structure is deprecated. It will be completely
removed in the next edition</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="objectType" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Type of object</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="NameType">
    <xs:annotation>
      <xs:documentation>From CIM</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="description" type="xs:string" minOccurs="0"/>
      <xs:element name="NameTypeAuthority" type="NameTypeAuthority" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Name">
    <xs:annotation>
      <xs:documentation>From CIM</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="NameType" type="NameType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="NameTypeAuthority">
    <xs:annotation>
      <xs:documentation>From CIM</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="description" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="FaultMessage" type="FaultMessageType">
    <xs:annotation>
      <xs:documentation>Fault message structure</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:simpleType name="IDKindType">
    <xs:annotation>
      <xs:documentation>ID Kind Type</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="name"/>
      <xs:enumeration value="uuid"/>
      <xs:enumeration value="transaction"/>
      <xs:enumeration value="other"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:attributeGroup name="IDatts">
    <xs:annotation>
      <xs:documentation>ID attribute group</xs:documentation>
    </xs:annotation>
    <xs:attribute name="idType" type="xs:string"/>
    <xs:attribute name="idAuthority" type="xs:string"/>
    <xs:attribute name="kind" type="IDKindType"/>
    <xs:attribute name="objectType" type="xs:string"/>
  </xs:attributeGroup>
</xs:schema>
```

Annex B (Normative): Verbs

The following table provides normative definitions of verbs to be used in message headers, as defined by the IEC 61968-1 standard. These are realized as enumerated values within Messsage.xsd.

Verbs	Meaning	Message Structure
create	The ‘create’ verb is used to publish a request to the master system to create a new object. The master system may in turn publish the new object as an event using the verb ‘created’. The master system may also use the verb ‘reply’ to respond to the ‘create’ request, indicating whether the request has been processed successfully or not.	Request message will include HeaderType and Payload structures.
change	The ‘change’ verb is used to publish a request to the master system to make a change to an object based on the information in the message. The master system may in turn publish the changed object as an event using the verb ‘changed’ to notify that the object has been changed since last published. The master system may also use the verb ‘reply’ to respond to the ‘change’ request, indicating whether the request has been processed successfully or not.	Request message will include HeaderType, RequestType and optionally Payload structures. The requestType structure will potentially identify specific object IDs.
cancel	The’cancel’ verb is used to publish a request to the master system to cancel the object, most commonly in the cases where the object represents a business document. The master system may in turn publish the cancelled message as an event using the verb ‘canceled’ to notify that the document has been cancelled since last published. The master system may also use the verb ‘reply’ to respond to the ‘cancel’ request, indicating whether the request has been processed successfully or not. The ‘cancel’ verb is used when the business content of the document is no longer valid due to error(s).	Request message will include HeaderType, RequestType and optionally Payload structures. The requestType structure will potentially identify specific object IDs.
close	The ‘close’ verb is used to publish a request to the master system to close the object, most commonly in cases where the object represents a business document. The master system may in turn publish the closed message as an event using the verb ‘closed’ to notify that the document has been closed since last published. The master system may also use the verb ‘reply’ to respond to the ‘close’ request, indicating whether the request has been processed successfully or not. The ‘close’ verb is used when the business document reaches the end of its life cycle due to successful completion of a business process.	Request message will include HeaderType, RequestType and optionally Payload structures. The requestType structure will potentially identify specific object IDs.
delete	The ‘delete’ verb is used to publish a request to the master system to delete one or more objects. The master system may in turn publish the closed message as an event using the verb ‘deleted’ to notify that the object has been deleted since last published. The master system may also use the verb ‘reply’ to respond to the ‘delete’ request, indicating whether the request has been processed successfully or not. The ‘delete’ verb is used when the business object should no longer be kept in the integrated systems either due to error(s) or due to archiving needs. However, the master system will most likely retain a historical record of the object after deletion.	Request message will include HeaderType, RequestType and optionally Payload structures. The requestType structure will potentially identify specific object IDs.
execute	This is used when the message is conveying a complex transaction that involves a variety of create, delete and/or change operations through the use of the Payload.OperationSet element..	See Payload.OperationSet in Message.xsd.
get	The ‘get’ verb is used to issue a query request to the master system to return a set of zero or more objects that meet a specified criteria. The master system may in turn return zero or more objects using the ‘reply’ verb in a response message.	Request message will include HeaderType and RequestType structures. The requestType structure will potentially identify specific parameters to qualify the request, such as object IDs.

Verbs	Meaning	Message Structure
created	The ‘created’ verb is used to publish an event that is a notification of the creation of a object as a result of either an external request or an internal action within the master system of that object. This message type is usually subscribed by interested systems and could be used for mass updates. There is no need to reply to this message type.	Event message will include HeaderType and Payload structures.
changed	The ‘changed’ verb is used to publish an event that is a notification of the change of an objectt as a result of either an external request or an internal action within the master system of that object. This could be a generic change in the content of the object or a specific status change such as “approved”, “issued” etc. This message type is usually subscribed by interested systems and could be used for mass updates. There is no need to reply to this message type.	Event message will include HeaderType and Payload structures.
closed	The ‘closed’ verb is used to publish an event that is a notification of the normal closure of an objectt as a result of either an external request or an internal action within the master system of that object. This message type is usually subscribed by interested systems and could be used for mass updates. There is no need to reply to this message type.	Event message will include HeaderType and Payload structures.
canceled	The ‘canceled’ verb is used to publish an event that is a notification of the cancellation of an objectt as a result of either an external request or an internal action within the master system of that object. This message type is usually subscribed by interested systems and could be used for mass updates. There is no need to reply to this message type.	Event message will include HeaderType and Payload structures.
deleted	The ‘deleted’ verb is used to publish an event that is a notification of the deletion of an object as a result of either an external request or an internal action within the master system of that object. This message type is usually subscribed by interested systems and could be used for mass updates. There is no need to reply to this message type.	Event message will include HeaderType and Payload structures.
executed	This provides for an event that indicates the execution of a complex transaction that uses the Payload.OperationSet element.	See Payload.OperationSet in Message.xsd.
reply	There are two primary usages of the ‘reply’ verb, but in both cases it is only used in response to request messages, whether the pattern used is synchronous or asynchronous. The first usage is to indicate the success, partial success or failure of a transactional request to the master system to create, change, delete, cancel, or close a document. The second usage is in response to a ‘get’ request, where objects of interest may be returned in the response.	Used only for response messages. For responses to transactional requests, the message will contain HeaderType and ReplyType structures. For responses to get requests, the message will contain HeaderType, ReplyType and potentially Payload structures.

Annex C (Normative): Procedure for Strongly Typed WSDL Generation

C.1 General

The purpose of this annex is to describe the process for the generation of WSDLs and related artifacts. The following diagram provides an overview of the process as needed to create and reference specific design artifacts.

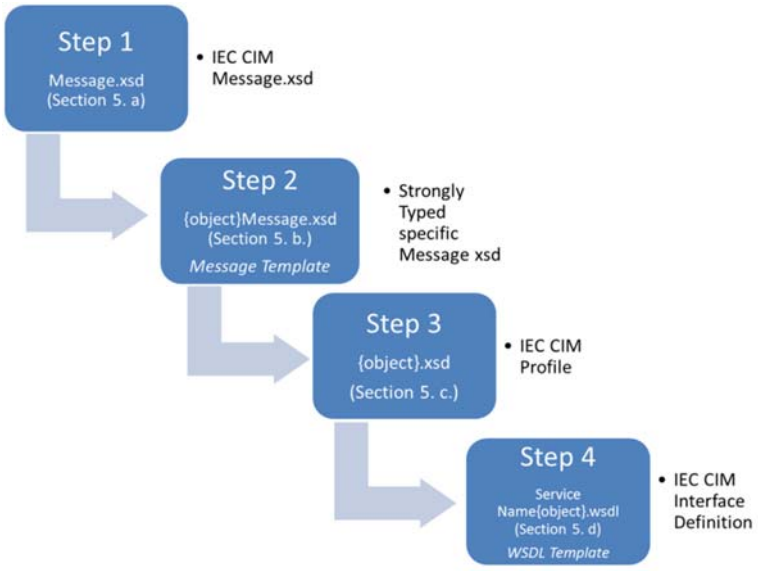
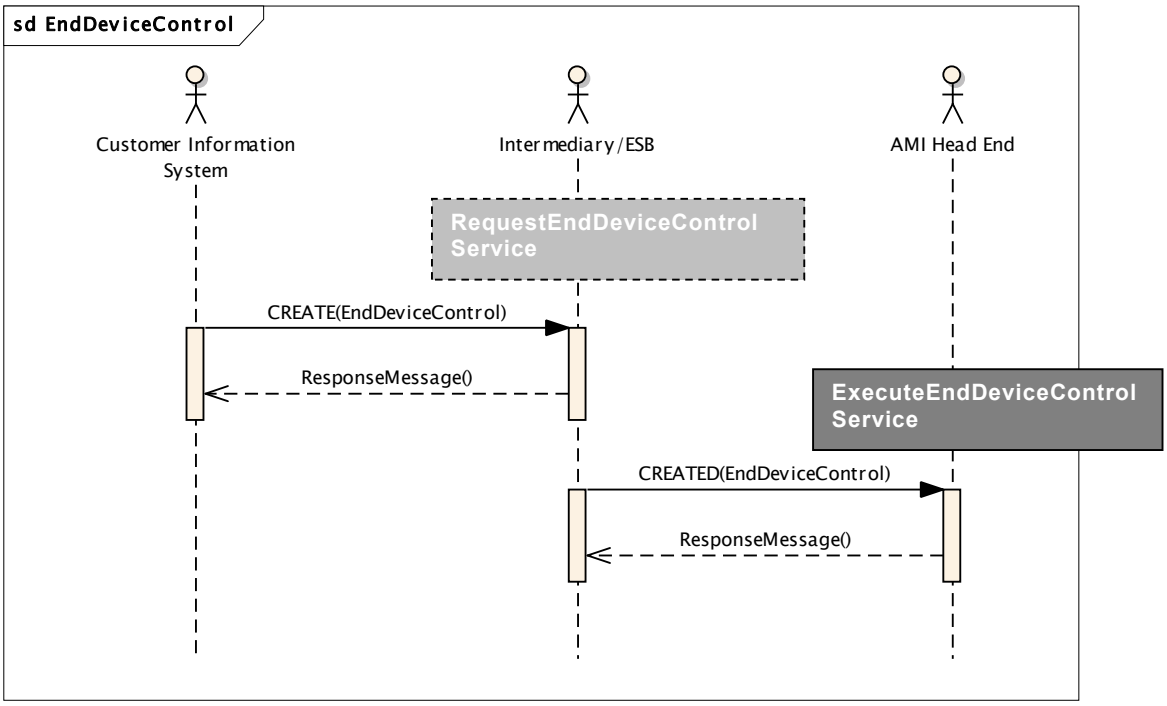


Figure C.1 - Process for WSDL Generation

Using templates, the process allows the creation of a WSDL that defines a set of operations for a given type of object (i.e. a specific noun). Each operation represents a combination of verb and noun. This WSDL will then reference a type specific message envelope, which references both the standard 61968 envelope structure definitions and a profile definition for the given noun.

C.2 WSDL Definition Steps

Step 1) – Sequence Diagram Based on Use Case
A sequence diagram shows message flow and service providers and consumers based on a use case. It presents messages in sequence base on integration requirements. The following diagram shows an EndDeviceControl message flow from CIS to HeadEnd via an intermediary ESB. CIS, in this case, is a requestor of EndDeviceControl so its message to ESB has a present tense verb “Create”. The message name follows the operation name convention <Verb>+<Information Object> as described in section 8.3.4 . The verb is “Create” and the information object is “EndDeviceControl” in this case.



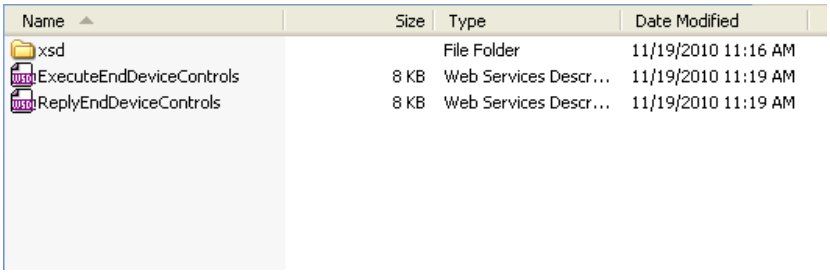
Step 2) – Service Semantics

Based on the sequence diagram, two services are provided for the entire message flow, one by ESB and one by HeadEnd. The naming of the two services are based on their service pattern (see secion 8.3.4) such as the role of service provided by the ESB is to "Request" end device control and the pattern of the service provided by HeadEnd is to "Execute" end device control. Each service will have the same operation as indicated in the sequence diagram as messages. As a result, both services have operation "CreatedEndDeviceControl" but have different service name, one is called "RequestEndDeviceControl" and the other named "ExecuteEndDeviceControl".

Step 3) Create Folder Structure

Under the preferred root folder, create an xsd folder for the xsd templates and CIM profiles. The WDSL's artifacts will be one-level above relative to the xsd folder.

The WSDLs reference the following folder structure.



The WSDLs are located in the root directory. The referenced XSD files are located in the XSD directory. A CIM profile xsd (EndDeviceControls.xsd), common message.xsd and {object}Message.xsd are located in this folder. The schema location is specified in wsdl:types section as the example for EndDeviceControlMessage listed below

```
<wsdl:types>
   <xs:schema targetNamespace="http://iec.ch/TC57/2011/schema/message"
```

```
                    elementFormDefault="qualified">
                    <xs:include schemaLocation="xsd/EndDeviceControlsMessage.xsd"/>
    </xs:schema>
  </wsdl:types>
```

The common message envelope, Message.xsd, can be found in Annex A.

Step 4) – Message Payload Definition & WSDL Generation

Service definition follows clause 8~~9~~. Document literal style is used in SOAP binding. As for a large payload MTOM is can be utilized. The template for a wrapped document style WSDL definition can be found in WSDL Template section in this Appendix.

This example demonstrates how to generate a WSDL for the *Execute* integration pattern. The same process can be used for any integration pattern by replacing Execute with the service naming pattern needed.

a. Copy the Message template xsd (see Message Template subclause in this Annex A) and place it in the correct folder directory
b. Copy the Message template (see Message Template subclause in this Annex C) and replace the {Information_Object_Name} variable with the correct noun.
    a. Save as {Information_Object_Name}Message.xsd (i.e. EndDeviceControlsMessage.xsd). This file is saved in the /xsd directory.

    There are two types of Object Message xsd templates, these include:
    - Send/Receive/Reply/Request/Execute
    - Get/Reply
c. Place the IEC CIM Profile xsd into the xsd folder that was created in Step 3.
d. Copy the Execute wsdl template (see WSDL Template subclause in this Annex) and replace the {Pattern_Name} variable with the correct pattern and replace the {Information_Object_Name} variable with the correct noun
    a. Save as Execute{Information_Object_Name}.wsdl (i.e. ExecuteEndDeviceControls.wsdl). This file is saved in the root directory.

    There are two types of Object Message xsd templates, these include:
    - Request/Execute
    - Send/Receive/Reply

Note that this is an example for *Execute* pattern but the steps are identical for other service patterns such as Reply for example.

C.3 Message Templates

The WSDL will reference a type-specific set of message structures, which in turn leverage the standard type-independent Message.xsd as described in Annex A. Occurrences of {Information_Object_Name} within the template would be replaced with a specific profile name.

Two message templates are provided in this subclause.

1) Message XSD Template for:

- **Send/Receive/Reply**
- **Request/Execute**

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://iec.ch/TC57/2011/{Information_Object_Name}Message"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msg="http://iec.ch/TC57/2011/schema/message"
xmlns:obj="http://iec.ch/TC57/2011/{Information_Object_Name}#"
targetNamespace="http://iec.ch/TC57/2011/{Information_Object_Name}Message"
elementFormDefault="qualified" attributeFormDefault="unqualified" version="1.0.0">
  <!-- Base Message Definitions -->
  <xs:import namespace="http://iec.ch/TC57/2011/schema/message"
schemaLocation="Message.xsd"/>
  <!-- CIM Information Object Definition -->
  <xs:import namespace="http://iec.ch/TC57/2011/{Information_Object_Name}#"
schemaLocation="{Information_Object_Name}.xsd"/>
  <!-- PayloadType Definition -->
  <xs:complexType name="{Information_Object_Name}PayloadType">
    <xs:sequence>
      <xs:element ref="obj:{Information_Object_Name}"/>
      <xs:element name="OperationSet" type="msg:OperationSet" minOccurs="0"/>
      <xs:element name="Compressed" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>For compressed and/or binary, uuencoded
payloads</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Format" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Hint as to format of payload, e.g. XML, RDF, SVF, BINARY,
PDF, ...</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <!-- Message Types -->
  <!-- RequestMessageType -->
  <xs:complexType name="{Information_Object_Name}RequestMessageType">
    <xs:sequence>
      <xs:element name="Header" type="msg:HeaderType"/>
      <xs:element name="Request" type="msg:RequestType" minOccurs="0"/>
      <xs:element name="Payload" type="tns:{Information_Object_Name}PayloadType"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <!-- ResponseMessageType -->
  <xs:complexType name="{Information_Object_Name}ResponseMessageType">
    <xs:sequence>
      <xs:element name="Header" type="msg:HeaderType"/>
      <xs:element name="Reply" type="msg:ReplyType"/>
      <xs:element name="Payload" type="tns:{Information_Object_Name}PayloadType"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <!-- EventMessageType -->
  <xs:complexType name="{Information_Object_Name}EventMessageType">
    <xs:sequence>
      <xs:element name="Header" type="msg:HeaderType"/>
      <xs:element name="Payload" type="tns:{Information_Object_Name}PayloadType"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <!-- FaultMessageType -->
  <xs:complexType name="{Information_Object_Name}FaultMessageType">
    <xs:sequence>
      <xs:element name="Reply" type="msg:ReplyType"/>
```

```
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Create{Information_Object_Name}"
type="tns:{Information_Object_Name}RequestMessageType"/>
  <xs:element name="Change{Information_Object_Name}"
type="tns:{Information_Object_Name}RequestMessageType"/>
  <xs:element name="Cancel{Information_Object_Name}"
type="tns:{Information_Object_Name}RequestMessageType"/>
  <xs:element name="Close{Information_Object_Name}"
type="tns:{Information_Object_Name}RequestMessageType"/>
  <xs:element name="Delete{Information_Object_Name}"
type="tns:{Information_Object_Name}RequestMessageType"/>
  <xs:element name="Created{Information_Object_Name}"
type="tns:{Information_Object_Name}EventMessageType"/>
  <xs:element name="Changed{Information_Object_Name}"
type="tns:{Information_Object_Name}EventMessageType"/>
  <xs:element name="Canceled{Information_Object_Name}"
type="tns:{Information_Object_Name}EventMessageType"/>
  <xs:element name="Closed{Information_Object_Name}"
type="tns:{Information_Object_Name}EventMessageType"/>
  <xs:element name="Deleted{Information_Object_Name}"
type="tns:{Information_Object_Name}EventMessageType"/>
  <xs:element name="{Information_Object_Name}ResponseMessage"
type="tns:{Information_Object_Name}ResponseMessageType"/>
  <xs:element name="{Information_Object_Name}FaultMessage"
type="tns:{Information_Object_Name}FaultMessageType"/>
</xs:schema>
```

2) Message XSD Template for

- **Get and Reply**

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://iec.ch/TC57/2011/Get{Information_Object_Name}Message"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msg="http://iec.ch/TC57/2011/schema/message"
xmlns:obj1="http://iec.ch/TC57/2011/{Information_Object_Name}#"
xmlns:obj2="http://iec.ch/TC57/2011/Get{Information_Object_Name}#"
targetNamespace="http://iec.ch/TC57/2011/Get{Information_Object_Name}Message"
elementFormDefault="qualified" attributeFormDefault="unqualified" version="1.0.0">
  <!-- Base Message Definitions -->
  <xs:import namespace="http://iec.ch/TC57/2011/schema/message"
schemaLocation="Message.xsd"/>
  <!-- CIM Information Object Definition -->
  <xs:import namespace="http://iec.ch/TC57/2011/{Information_Object_Name}#"
schemaLocation="{Information_Object_Name}.xsd"/>
  <!-- Remove this Import if there is no "Get" Profile associated with this Object. --
>
  <xs:import namespace="http://iec.ch/TC57/2011/Get{Information_Object_Name}#"
schemaLocation="Get{Information_Object_Name}.xsd"/>
  <!-- RequestType Definition -->
  <xs:complexType name="Get{Information_Object_Name}RequestType">
    <xs:sequence>
      <xs:element name="StartTime" type="xs:dateTime" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Start time of interest</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="EndTime" type="xs:dateTime" minOccurs="0">
        <xs:annotation>
          <xs:documentation>End time of interest</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Option" type="msg:OptionType" minOccurs="0"
maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Request type specialization</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="ID" type="xs:string" minOccurs="0" maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>Object ID for request</xs:documentation>
        </xs:annotation>
      </xs:element>
      <!-- Remove this Element if there is no "Get" Profile associated with this
Object. -->
      <xs:element ref="obj2:Get{Information_Object_Name}"/>
```

```
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded">
        <xs:annotation>
          <xs:documentation>This can be a CIM profile defined as an XSD with a CIM-
specific namespace</xs:documentation>
        </xs:annotation>
      </xs:any>
    </xs:sequence>
  </xs:complexType>
  <!-- PayloadType Definition -->
  <xs:complexType name="{Information_Object_Name}PayloadType">
    <xs:sequence>
      <xs:element ref="obj1:{Information_Object_Name}" minOccurs="0"/>
      <xs:element name="OperationSet" type="msg:OperationSet" minOccurs="0"/>
      <xs:element name="Compressed" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>For compressed and/or binary, uuencoded
payloads</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Format" type="xs:string" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Hint as to format of payload, e.g. XML, RDF, SVF, BINARY,
PDF, ...</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <!-- Message Types -->
  <!-- RequestMessageType -->
  <xs:complexType name="Get{Information_Object_Name}RequestMessageType">
    <xs:sequence>
      <xs:element name="Header" type="msg:HeaderType"/>
      <xs:element name="Request" type="tns:Get{Information_Object_Name}RequestType"/>
      <xs:element name="Payload" type="tns:{Information_Object_Name}PayloadType"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <!-- ResponseMessageType -->
  <xs:complexType name="{Information_Object_Name}ResponseMessageType">
    <xs:sequence>
      <xs:element name="Header" type="msg:HeaderType"/>
      <xs:element name="Reply" type="msg:ReplyType"/>
      <xs:element name="Payload" type="tns:{Information_Object_Name}PayloadType"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <!-- FaultMessageType -->
  <xs:complexType name="{Information_Object_Name}FaultMessageType">
    <xs:sequence>
      <xs:element name="Reply" type="msg:ReplyType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Get{Information_Object_Name}"
type="tns:Get{Information_Object_Name}RequestMessageType"/>
  <xs:element name="{Information_Object_Name}ResponseMessage"
type="tns:{Information_Object_Name}ResponseMessageType"/>
  <xs:element name="{Information_Object_Name}FaultMessage"
type="tns:{Information_Object_Name}FaultMessageType"/>
</xs:schema>
```

Note:

The following strings need to be replaced for both templates

{Information_Object_Name}

- Replaced with CIM profile that is being used. For example, EndDeviceControls. We use the plural form of the information object to avoid collisons within the XSD.

C.4 WSDL Templates

This section provides WSDL templates that would be edited in order to create design artifacts for strongly typed web services.

WSDL for 'Get' requests

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
   name="Get{Information_Object_Name}"
        targetNamespace="http://iec.ch/TC57/2011/Get{Information_Object_Name}"
        xmlns:tns="http://iec.ch/TC57/2011/Get{Information_Object_Name}"
        xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
        xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"

xmlns:infoMessage="http://iec.ch/TC57/2011/Get{Information_Object_Name}Message">
        <wsdl:types>

        <xs:schema
targetNamespace="http://iec.ch/TC57/2011/Get{Information_Object_Name}"
                     elementFormDefault="qualified">

               <xs:import
namespace="http://iec.ch/TC57/2011/Get{Information_Object_Name}Message"
schemaLocation="xsd/Get{Information_Object_Name}Message.xsd"/>

        </xs:schema>

     </wsdl:types>

     <!-- Message Definitions -->
     <wsdl:message name="Get{Information_Object_Name}RequestMessage">
        <wsdl:part name="Get{Information_Object_Name}RequestMessage"
element="infoMessage:Get{Information_Object_Name}"/>
     </wsdl:message>

     <wsdl:message name="ResponseMessage">
        <wsdl:part name="ResponseMessage"
element="infoMessage:{Information_Object_Name}ResponseMessage"/>
     </wsdl:message>

     <wsdl:message name="FaultMessage">
        <wsdl:part name="FaultMessage"
element="infoMessage:{Information_Object_Name}FaultMessage"/>
     </wsdl:message>

     <!-- Port Definitions -->
     <wsdl:portType name="Get{Information_Object_Name}_Port">

        <wsdl:operation name="Get{Information_Object_Name}">
           <wsdl:input name="Get{Information_Object_Name}Request"
message="tns:Get{Information_Object_Name}RequestMessage"/>
           <wsdl:output name="Get{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Get{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

     </wsdl:portType>

     <wsdl:binding name="Get{Information_Object_Name}_Binding"
type="tns:Get{Information_Object_Name}_Port">

        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>

        <wsdl:operation name="Get{Information_Object_Name}">
           <soap:operation
soapAction="http://iec.ch/TC57/2011/Get{Information_Object_Name}/Get{Information_Objec
t_Name}" style="document"/>
```

```
            <wsdl:input name="Get{Information_Object_Name}Request">
               <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="Get{Information_Object_Name}Response">
               <soap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="Get{Information_Object_Name}Fault">
               <soap:fault name="Get{Information_Object_Name}Fault" use="literal"/>
            </wsdl:fault>
         </wsdl:operation>
      </wsdl:binding>

      <wsdl:service name="Get{Information_Object_Name}">
         <wsdl:port name="Get{Information_Object_Name}_Port"
binding="tns:Get{Information_Object_Name}_Binding">
            <soap:address
location="http://iec.ch/TC57/2011/Get{Information_Object_Name}"/>
         </wsdl:port>
      </wsdl:service>

</wsdl:definitions>
```

WSDL for Send, Receive, Reply

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
        name="{Send | Receive | Reply}{Information_Object_Name}"
        targetNamespace="http://iec.ch/TC57/2011/{Send | Receive |
Reply}{Information_Object_Name}"
        xmlns:tns="http://iec.ch/TC57/2011/{Send | Receive |
Reply}{Information_Object_Name}"
        xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
        xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
        xmlns:infoMessage="http://iec.ch/TC57/2011/{Information_Object_Name}Message">

      <wsdl:types>

         <xs:schema targetNamespace="http://iec.ch/TC57/2011/{Send | Receive |
Reply}{Information_Object_Name}"
                    elementFormDefault="qualified">

               <xs:import
namespace="http://iec.ch/TC57/2011/{Information_Object_Name}Message"
schemaLocation="xsd/{Information_Object_Name}Message.xsd"/>

         </xs:schema>

      </wsdl:types>

      <!-- Message Definitions -->

      <wsdl:message name="Created{Information_Object_Name}EventMessage">
         <wsdl:part name="Created{Information_Object_Name}EventMessage"
element="infoMessage:Created{Information_Object_Name}"/>
      </wsdl:message>

      <wsdl:message name="Changed{Information_Object_Name}EventMessage">
         <wsdl:part name="Changed{Information_Object_Name}EventMessage"
element="infoMessage:Changed{Information_Object_Name}"/>
      </wsdl:message>

      <wsdl:message name="Closed{Information_Object_Name}EventMessage">
         <wsdl:part name="Closed{Information_Object_Name}EventMessage"
element="infoMessage:Closed{Information_Object_Name}"/>
      </wsdl:message>

      <wsdl:message name="Canceled{Information_Object_Name}EventMessage">
         <wsdl:part name="Canceled{Information_Object_Name}EventMessage"
element="infoMessage:Canceled{Information_Object_Name}"/>
      </wsdl:message>
```

```
     <wsdl:message name="Deleted{Information_Object_Name}EventMessage">
        <wsdl:part name="Deleted{Information_Object_Name}EventMessage"
element="infoMessage:Deleted{Information_Object_Name}"/>
     </wsdl:message>

     <wsdl:message name="ResponseMessage">
        <wsdl:part name="ResponseMessage"
element="infoMessage:{Information_Object_Name}ResponseMessage"/>
     </wsdl:message>

     <wsdl:message name="FaultMessage">
        <wsdl:part name="FaultMessage"
element="infoMessage:{Information_Object_Name}FaultMessage"/>
     </wsdl:message>

     <!-- Port Definitions -->
     <wsdl:portType name="{Information_Object_Name}_Port">

        <wsdl:operation name="Created{Information_Object_Name}">
           <wsdl:input name="Created{Information_Object_Name}Event"
message="tns:Created{Information_Object_Name}EventMessage"/>
           <wsdl:output name="Created{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Created{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

        <wsdl:operation name="Changed{Information_Object_Name}">
           <wsdl:input name="Changed{Information_Object_Name}Event"
message="tns:Changed{Information_Object_Name}EventMessage"/>
           <wsdl:output name="Changed{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Changed{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

        <wsdl:operation name="Canceled{Information_Object_Name}">
           <wsdl:input name="Canceled{Information_Object_Name}Event"
message="tns:Canceled{Information_Object_Name}EventMessage"/>
           <wsdl:output name="Canceled{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Canceled{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

        <wsdl:operation name="Closed{Information_Object_Name}">
           <wsdl:input name="Closed{Information_Object_Name}Event"
message="tns:Closed{Information_Object_Name}EventMessage"/>
           <wsdl:output name="Closed{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Closed{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

        <wsdl:operation name="Deleted{Information_Object_Name}">
           <wsdl:input name="Deleted{Information_Object_Name}Event"
message="tns:Deleted{Information_Object_Name}EventMessage"/>
           <wsdl:output name="Deleted{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Deleted{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

     </wsdl:portType>

     <wsdl:binding name="{Information_Object_Name}_Binding"
type="tns:{Information_Object_Name}_Port">

        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>

        <wsdl:operation name="Created{Information_Object_Name}">
           <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Created{Information_Obje
ct_Name}" style="document"/>
           <wsdl:input name="Created{Information_Object_Name}Event">
              <soap:body use="literal"/>
           </wsdl:input>
```

```
           <wsdl:output name="Created{Information_Object_Name}Response">
              <soap:body use="literal"/>
           </wsdl:output>
           <wsdl:fault name="Created{Information_Object_Name}Fault">
              <soap:fault name="Created{Information_Object_Name}Fault" use="literal"/>
           </wsdl:fault>
        </wsdl:operation>
        <wsdl:operation name="Changed{Information_Object_Name}">
           <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Changed{Information_Obje
ct_Name}" style="document"/>
           <wsdl:input name="Changed{Information_Object_Name}Event">
              <soap:body use="literal"/>
           </wsdl:input>
           <wsdl:output name="Changed{Information_Object_Name}Response">
              <soap:body use="literal"/>
           </wsdl:output>
           <wsdl:fault name="Changed{Information_Object_Name}Fault">
              <soap:fault name="Changed{Information_Object_Name}Fault" use="literal"/>
           </wsdl:fault>
        </wsdl:operation>
        <wsdl:operation name="Canceled{Information_Object_Name}">
           <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Canceled{Information_Obj
ect_Name}" style="document"/>
           <wsdl:input name="Canceled{Information_Object_Name}Event">
              <soap:body use="literal"/>
           </wsdl:input>
           <wsdl:output name="Canceled{Information_Object_Name}Response">
              <soap:body use="literal"/>
           </wsdl:output>
           <wsdl:fault name="Canceled{Information_Object_Name}Fault">
              <soap:fault name="Canceled{Information_Object_Name}Fault"
use="literal"/>
           </wsdl:fault>
        </wsdl:operation>
        <wsdl:operation name="Closed{Information_Object_Name}">
           <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Closed{Information_Objec
t_Name}" style="document"/>
           <wsdl:input name="Closed{Information_Object_Name}Event">
              <soap:body use="literal"/>
           </wsdl:input>
           <wsdl:output name="Closed{Information_Object_Name}Response">
              <soap:body use="literal"/>
           </wsdl:output>
           <wsdl:fault name="Closed{Information_Object_Name}Fault">
              <soap:fault name="Closed{Information_Object_Name}Fault" use="literal"/>
           </wsdl:fault>
        </wsdl:operation>
        <wsdl:operation name="Deleted{Information_Object_Name}">
           <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Deleted{Information_Obje
ct_Name}" style="document"/>
           <wsdl:input name="Deleted{Information_Object_Name}Event">
              <soap:body use="literal"/>
           </wsdl:input>
           <wsdl:output name="Deleted{Information_Object_Name}Response">
              <soap:body use="literal"/>
           </wsdl:output>
           <wsdl:fault name="Deleted{Information_Object_Name}Fault">
              <soap:fault name="Deleted{Information_Object_Name}Fault" use="literal"/>
           </wsdl:fault>
        </wsdl:operation>
     </wsdl:binding>

     <wsdl:service name="{Send | Receive | Reply}{Information_Object_Name}">
        <wsdl:port name="{Information_Object_Name}_Port"
binding="tns:{Information_Object_Name}_Binding">
           <soap:address location="http://iec.ch/TC57/2011/{Send | Receive |
Reply}{Information_Object_Name}"/>
        </wsdl:port>
     </wsdl:service>

</wsdl:definitions>
```

Note that {Send | Receive | Reply} should be replaced with a proper service patter name such Receive.

WSDL for Request, Execute

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
      name="{Request | Execute}{Information_Object_Name}"
         targetNamespace="http://iec.ch/TC57/2011/{Request |
Execute}{Information_Object_Name}"
         xmlns:tns="http://iec.ch/TC57/2011/{Request |
Execute}{Information_Object_Name}"
         xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
         xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"
         xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
         xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
         xmlns:infoMessage="http://iec.ch/TC57/2011/{Information_Object_Name}Message">

      <wsdl:types>

         <xs:schema targetNamespace="http://iec.ch/TC57/2011/{Request |
Execute}{Information_Object_Name}"
                        elementFormDefault="qualified">

                  <xs:import
namespace="http://iec.ch/TC57/2011/{Information_Object_Name}Message"
schemaLocation="xsd/{Information_Object_Name}Message.xsd"/>

         </xs:schema>

      </wsdl:types>
      <xs:schema
targetNamespace="http://iec.ch/TC57/2011/ExecuteEndDeviceControlsMessage"
                        elementFormDefault="qualified">
                  <xs:import
namespace="http://iec.ch/TC57/2011/EndDeviceControlsMessage"
schemaLocation="xsd/EndDeviceControlsMessage.xsd"/>
                  <!--<xs:include schemaLocation="xsd/EndDeviceControlsMessage.xsd"/>-->

         </xs:schema>

      <!-- Message Definitions -->

      <wsdl:message name="Create{Information_Object_Name}RequestMessage">
         <wsdl:part name="Create{Information_Object_Name}RequestMessage"
element="infoMessage:Create{Information_Object_Name}"/>
      </wsdl:message>

      <wsdl:message name="Change{Information_Object_Name}RequestMessage">
         <wsdl:part name="Change{Information_Object_Name}RequestMessage"
element="infoMessage:Change{Information_Object_Name}"/>
      </wsdl:message>

      <wsdl:message name="Close{Information_Object_Name}RequestMessage">
         <wsdl:part name="Close{Information_Object_Name}RequestMessage"
element="infoMessage:Close{Information_Object_Name}"/>
      </wsdl:message>

      <wsdl:message name="Cancel{Information_Object_Name}RequestMessage">
         <wsdl:part name="Cancel{Information_Object_Name}RequestMessage"
element="infoMessage:Cancel{Information_Object_Name}"/>
      </wsdl:message>

      <wsdl:message name="Delete{Information_Object_Name}RequestMessage">
         <wsdl:part name="Delete{Information_Object_Name}RequestMessage"
element="infoMessage:Delete{Information_Object_Name}"/>
      </wsdl:message>

      <wsdl:message name="ResponseMessage">
         <wsdl:part name="ResponseMessage"
element="infoMessage:{Information_Object_Name}ResponseMessage"/>
      </wsdl:message>

      <wsdl:message name="FaultMessage">
         <wsdl:part name="FaultMessage"
element="infoMessage:{Information_Object_Name}FaultMessage"/>
```

```
     </wsdl:message>

     <!-- Port Definitions -->
     <wsdl:portType name="{Information_Object_Name}_Port">

        <wsdl:operation name="Create{Information_Object_Name}">
           <wsdl:input name="Create{Information_Object_Name}Request"
message="tns:Create{Information_Object_Name}RequestMessage"/>
           <wsdl:output name="Create{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Create{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

        <wsdl:operation name="Change{Information_Object_Name}">
           <wsdl:input name="Change{Information_Object_Name}Request"
message="tns:Change{Information_Object_Name}RequestMessage"/>
           <wsdl:output name="Change{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Change{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

        <wsdl:operation name="Cancel{Information_Object_Name}">
           <wsdl:input name="Cancel{Information_Object_Name}Request"
message="tns:Cancel{Information_Object_Name}RequestMessage"/>
           <wsdl:output name="Cancel{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Cancel{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

        <wsdl:operation name="Close{Information_Object_Name}">
           <wsdl:input name="Close{Information_Object_Name}Request"
message="tns:Close{Information_Object_Name}RequestMessage"/>
           <wsdl:output name="Close{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Close{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

        <wsdl:operation name="Delete{Information_Object_Name}">
           <wsdl:input name="Delete{Information_Object_Name}Request"
message="tns:Delete{Information_Object_Name}RequestMessage"/>
           <wsdl:output name="Delete{Information_Object_Name}Response"
message="tns:ResponseMessage"/>
           <wsdl:fault name="Delete{Information_Object_Name}Fault"
message="tns:FaultMessage"/>
        </wsdl:operation>

     </wsdl:portType>

     <wsdl:binding name="{Information_Object_Name}_Binding"
type="tns:{Information_Object_Name}_Port">

        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>

        <wsdl:operation name="Create{Information_Object_Name}">
           <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Create{Information_Objec
t_Name}" style="document"/>
           <wsdl:input name="Create{Information_Object_Name}Request">
              <soap:body use="literal"/>
           </wsdl:input>
           <wsdl:output name="Create{Information_Object_Name}Response">
              <soap:body use="literal"/>
           </wsdl:output>
           <wsdl:fault name="Create{Information_Object_Name}Fault">
              <soap:fault name="Create{Information_Object_Name}Fault" use="literal"/>
           </wsdl:fault>
        </wsdl:operation>
        <wsdl:operation name="Change{Information_Object_Name}">
           <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Change{Information_Objec
t_Name}" style="document"/>
           <wsdl:input name="Change{Information_Object_Name}Request">
              <soap:body use="literal"/>
```

```
            </wsdl:input>
            <wsdl:output name="Change{Information_Object_Name}Response">
               <soap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="Change{Information_Object_Name}Fault">
               <soap:fault name="Change{Information_Object_Name}Fault" use="literal"/>
            </wsdl:fault>
         </wsdl:operation>
         <wsdl:operation name="Cancel{Information_Object_Name}">
            <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Cancel{Information_Objec
t_Name}" style="document"/>
            <wsdl:input name="Cancel{Information_Object_Name}Request">
               <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="Cancel{Information_Object_Name}Response">
               <soap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="Cancel{Information_Object_Name}Fault">
               <soap:fault name="Cancel{Information_Object_Name}Fault" use="literal"/>
            </wsdl:fault>
         </wsdl:operation>
         <wsdl:operation name="Close{Information_Object_Name}">
            <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Close{Information_Object
_Name}" style="document"/>
            <wsdl:input name="Close{Information_Object_Name}Request">
               <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="Close{Information_Object_Name}Response">
               <soap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="Close{Information_Object_Name}Fault">
               <soap:fault name="Close{Information_Object_Name}Fault" use="literal"/>
            </wsdl:fault>
         </wsdl:operation>
         <wsdl:operation name="Delete{Information_Object_Name}">
            <soap:operation
soapAction="http://iec.ch/TC57/2011/{Information_Object_Name}/Delete{Information_Objec
t_Name}" style="document"/>
            <wsdl:input name="Delete{Information_Object_Name}Request">
               <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="Delete{Information_Object_Name}Response">
               <soap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="Delete{Information_Object_Name}Fault">
               <soap:fault name="Delete{Information_Object_Name}Fault" use="literal"/>
            </wsdl:fault>
         </wsdl:operation>
      </wsdl:binding>

      <wsdl:service name="{Request | Execute}{Information_Object_Name}">
         <wsdl:port name="{Information_Object_Name}_Port"
binding="tns:{Information_Object_Name}_Binding">
            <soap:address location="http://iec.ch/TC57/2011/{Request |
Execute}{Information_Object_Name}"/>
         </wsdl:port>
      </wsdl:service>

</wsdl:definitions>
```

Note that {Request | Execute} should be replaced with a proper service patter name such Execute.

Notes:

For all templates, the following string needs to be replaced.

{Information_Object_Name}
Replaced with CIM profile that is being used, for example, EndDeviceControls. The plural form of the information object is used to avoid collisions within the XSD.

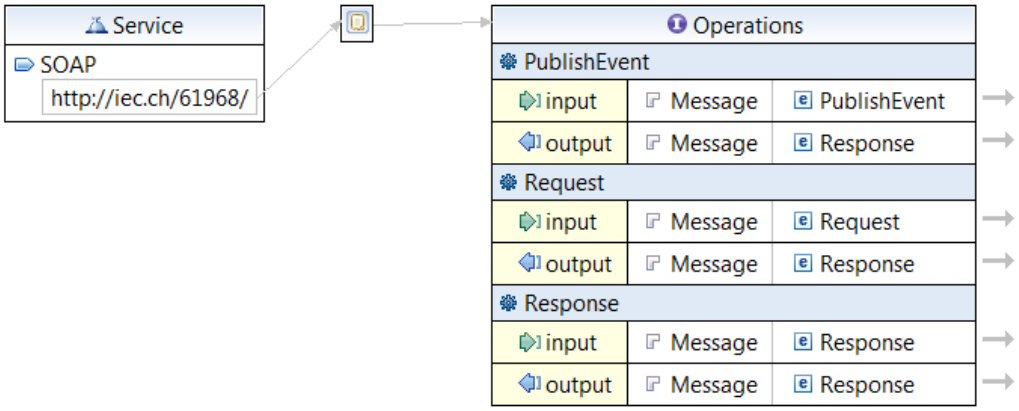
Annex D (Normative): Generic WSDL

The purpose of this annex is to describe a generic WSDL that is not strongly typed to a specific verb and noun combination . Instead, this WSDL provides the ability to convey messages that may use any valid verb/noun and noun combinations, where the common message envelope as defined by Message.xsd is used without modification. The generic WSDL provides three operations:

- Request: to issue requests, where a response may be returned
- Response: to issue asynchronous responses
- PublishEvent: to sent event messages, where the assumption is that an intermediary is responsible for publication of the event to all potentially interested 'listeners'

This approach has the benefit of avoiding the need to construct variations of Message.xsd.

The following diagram provides an overview of the operations and messages.



The following is XML that defines the abstract WSDL for the implementation of a generic IEC 61968-100 web service.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- IEC 61968 WSDL for Generic, Type-Independent Web Services -->
<!-- Uses document wrapped WSDL style -->
<wsdl:definitions xmlns:ns="http://iec.ch/TC57/2011/abstract"
xmlns:ns2="http://iec.ch/TC57/2011/schema/message"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://iec.ch/TC57/2011/abstract">
   <wsdl:types>
      <xsd:schema targetNamespace="http://iec.ch/TC57/2011/schema/message">
         <xsd:include schemaLocation="xsd/Message.xsd"/>
         <xsd:element name="PublishEvent" type="ns2:EventMessageType"/>
         <xsd:element name="Request" type="ns2:RequestMessageType"/>
         <xsd:element name="Response" type="ns2:ResponseMessageType"/>
      </xsd:schema>
   </wsdl:types>
   <wsdl:message name="EventMessage">
      <wsdl:part name="Message" element="ns2:EventMessage"/>
   </wsdl:message>
   <wsdl:message name="RequestMessage">
      <wsdl:part name="Message" element="ns2:RequestMessage"/>
   </wsdl:message>
   <wsdl:message name="ResponseMessage">
      <wsdl:part name="Message" element="ns2:ResponseMessage"/>
   </wsdl:message>
   <wsdl:portType name="Operations">
      <wsdl:operation name="PublishEvent">
         <wsdl:input message="ns:EventMessage"/>
```

```
      <wsdl:output message="ns:ResponseMessage"/>
    </wsdl:operation>
    <wsdl:operation name="Request">
      <wsdl:input message="ns:RequestMessage"/>
      <wsdl:output message="ns:ResponseMessage"/>
    </wsdl:operation>
    <wsdl:operation name="Response">
      <wsdl:input message="ns:ResponseMessage"/>
      <wsdl:output message="ns:ResponseMessage"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="SOAP" type="ns:Operations">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
            <!-- Operation for publication of events -->
    <wsdl:operation name="PublishEvent">
      <soap:operation soapAction="http://iec.ch/61968/PublishEvent"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
            <!-- Operation for request/reply interactions -->
    <wsdl:operation name="Request">
      <soap:operation soapAction="http://iec.ch/61968/Request"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
            <!-- Operation for asynchronous responses -->
    <wsdl:operation name="Response">
      <soap:operation soapAction="http://iec.ch/61968/Response"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Service">
    <wsdl:port name="SOAP" binding="ns:SOAP">
      <soap:address location="http://iec.ch/61968/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Annex E (Informative): AMQP

The Advanced Message Queueing Protocol (AMQP) defines an open ‘wire’ protocol for queue-based messaging. Products that use the AMQP protocol are becoming common in the marketplace, as well as the availability of open source offerings. This is contrasted by, but also complementary of the fact that JMS provides a standardized API but JMS implementations do not have a standardized wire protocol.

The use of AMQP with IEC 61968-100 is identical to the use of JMS using queues in cases where the clients use the JMS API. From the perspective of the client application code, the use of AMQP may be completely transparent. The application data conveyed within messages is simply conveyed using the common message envelope as defined by Message.xsd. The same general approach can be taken with other AMQP APIs.

Details on AMQP are available at http://amqp.org.

Annex F (Informative): Payload Compression Example

The purpose of this annex is to provide an example of the code required to compress and encode a payload, where the payload is then passed as the contents of the Payload/Compressed element. Payload compression can be used for any messaging technology, including JMS, generic web services and strongly typed web services.

The following is a Java class example that leverages commonly used classes for compression and base64 encoding.

```
package soap.test;

import java.io.*;
import java.util.zip.*;
import org.apache.commons.codec.binary.Base64;

public class CompressionClientCompressandEncode{
     private byte[] input = null;

     public CompressionClientCompressandEncode(String xmlInput) {
       this.input = xmlInput.getBytes();
     }

     public CompressionClientCompressandEncode(byte[] input) {
       this.input = input;
     }

     // Returns the Compressed and Encoded byte[]

     private byte[] compressAndEncode() throws Exception {

       // GZIP the contents..
         ByteArrayOutputStream outstream =
           new ByteArrayOutputStream();
         GZIPOutputStream gzipOutstream =
           new GZIPOutputStream(outstream);
         gzipOutstream.write(input);
         gzipOutstream.close();

         // Encode the compressed byte array from the stream
         byte[] b =
           Base64.encodeBase64(outstream.toByteArray());
         return b;
     }
}
```

The following program (when combined with the above class) shows example usage, where sample input XML is zipped and encoded using the above code, and then decoded and unzipped to get the original input:

```
public static void main ( String args[]) {
  String s = "<root>" +
             "<name>raju</name>" +
             "</root>";

  System.out.println("Original Xml");
  System.out.println(s);

  byte[] b = s.getBytes();
```

```
        CompressionClientCompressandEncode cc =
            new CompressionClientCompressandEncode(b);

        try {

              byte[] compressedAndEncoded = cc.compressAndEncode();

                      System.out.println("Compressed And Encoded");
                      System.out.println(
                        new String(compressedAndEncoded));

                    // Validate ..
                    byte[] unencoded =
                          Base64.decodeBase64(compressedAndEncoded);

                    ByteArrayInputStream bi1 =
                          new ByteArrayInputStream(unencoded);
                    GZIPInputStream g1 = new GZIPInputStream(bi1);


                    System.out.println("Unencoded and Uncompressed..");
                    for (int c = g1.read(); c != -1; c = g1.read()) {
                        System.out.write(c);
                    }
                    System.out.flush();

              } catch (Exception e) {
                     // TODO Auto-generated catch block
                     e.printStackTrace();
              }
    }
```

An important note is that if compression is used, either both the source and target systems must support compression, or any mismatches be addressed by intermediary processes within the ESB. The Gzip compression algoroithm should be used, as is demonstrated in this example.

Annex G (Informative): XMPP

The Extensible Messaging and Presence Protocol (XMPP) defines an application level protocol for near real-time communications using XML. XMPP is standardized by IETF RFCs 6120, 6121 and 6122. There are many functional similarities to JMS, and as a consequence it is possible to map IEC 61968-100 onto XMPP.

Using XMPP, clients connect to an XMPP server just as JMS clients would connect to a JMS server. Messages are sent using XML 'stanzas', where each stanza conveys an XML element that is logically a fragment of an XML document that is representative of a session. There are three fundamental types of stanzas:

- <message> – used for pushing messages
- <iq> – Info/Query, used for request/response patterns, where an IQ stanza may be of one of four types: get, set, result or error
- <presence> – Used to convey the status of a contact

The IEC 61968-100 message envelope can be placed within an XMPP stanza. The type of stanza that should be used is dependent upon the messaging pattern. Request/reply patterns should use the <iq> stanza type. Publish/subscribe message patterns should use the <message> stanza type. The following is an example of an IEC61968-9 event message being conveyed within an XMPP stanza.

```
<message from='meter76943@myUtility.com' to='headend@myUtility.com'>
  <body>
        <ns0:Message xmlns:ns0="http://www.iec.ch/TC57/2011/schema/message">
          <ns0:Header>
            <ns0:Verb>created</ns0:Verb>
            <ns0:Noun>EndDeviceEvents</ns0:Noun>
            <ns0:Revision>1</ns0:Revision>
            <ns0:Timestamp>2009-11-04T18:52:50.001-05:00</ns0:Timestamp>
            <ns0:Source>Metering System</ns0:Source>
          </ns0:Header>
          <ns0:Payload>
            <ns1:EndDeviceEvents xmlns:ns1="http://iec.ch/TC57/2011EndDeviceEvents#">
              <ns1:EndDeviceEvent ref='3.26.1.185'>
                <ns1:mRID>76943</ns1:mRID>
                <ns1:createdDateTime>2009-11-04T18:52:50.001-05:00</ns1:createdDateTime>
                <ns1:description>Power off alarm</ns1:description>
                <ns1:severity>1</ns1:severity>
                <ns1:Assets>
        <ns1:mRID>AC761473800C7B0417481114A11348C16111911121B46C016BF012C68121106</ns1:mRID>
                </ns1:Assets>
                  </ns1:EndDeviceEvent>
              </ns1:EndDeviceEvents>
          </ns0:Payload>
        </ns0:Message>
  </body>
</message>
```

XMPP messages are addressed using the 'from' and 'to' attributes within the <message> and <iq> elements. Where JMS topic or queue names would be used for addressing with JMS, addressing for XMPP is defined by IETF RFC 6122.

Bibliography

*EIP: Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison Wesley, October 2003; ISBN-10: 0321200683, ISBN-13: 978-0321200686 (*http://www.eaipatterns.com*).*

SOAP over Java Message Service Proposed Recommendation: http://www.w3.org/TR/soapjms/

XSL: Extensible Stylesheet Language (XSL): www.w3.org/TR/xsl/.

XPATH: XML Path Language (XPATH): www.w3.org/TR/xpath/.

IEC 61968-13: CIM RDF Model exchange format for distribution

IEC 61968-9: Interfaces for meter reading and control

IEC 61970-452: CIM Statis Transmission Network Model Profiles

IEC 61970-453[2]: Diagram Layout Profile

---

[2] under consideration