Open Geospatial Consortium, Inc.

Date: 2009-07-29

Reference number of this document: OGC 09-032

Version: 0.3.0

Category: Public Engineering Report

Editors: Thomas Everding, Johannes Echterhoff

OGC® OWS-6 SWE Event Architecture Engineering Report

Warning

This document is not an OGC Standard. This document is an OGC Public Engineering Report created as a deliverable in an OGC Interoperability Initiative and is not an official position of the OGC membership. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard. Further, any OGC Engineering Report should not be referenced as required or mandatory technology in procurements.

Document type:	Public Engineering Report
Document subtype:	NA
Document stage:	Approved for public release
Document language:	English

Preface

This public Engineering Report (ER) is a deliverable of the Open Geospatial Consortium (OGC) Interoperability Program Open Web Service (OWS) Testbed phase 6 (OWS-6).

The document describes an abstract event architecture for service oriented architectures. Furthermore various techniques for implementing an event architecture and working with events are discussed.

This work was supported by the European Commission through the GENESIS project (an Integrated Project, contract number 223996), the OSIRIS project (an Integrated Project, contract number 033475) and through the SANY project (an Integrated Project, contract number 0033564), Information Society and Media DG of the European Commission within the RTD activities of the Thematic Priority Information Society Technologies.

Suggested additions, changes, and comments on this draft report are welcome and encouraged. Such suggestions may be submitted by email message or by making suggested changes in an edited copy of this document.

The changes made in this document version, relative to the previous version, are tracked by Microsoft Word, and can be viewed if desired. If you choose to submit suggested changes by editing this document, please first accept all the current changes, and then make your suggested changes with change tracking on.

Forward

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium Inc. shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

OWS-6 Testbed

OWS testbeds are part of OGC's Interoperability Program, a global, hands-on and collaborative prototyping program designed to rapidly develop, test and deliver Engineering Reports and Change Requests into the OGC Specification Program, where they are formalized for public release. In OGC's Interoperability Initiatives, international teams of technology providers work together to solve specific geoprocessing interoperability problems posed by the Initiative's sponsoring organizations. OGC Interoperability Initiatives include test beds, pilot projects, interoperability experiments and interoperability support services - all designed to encourage rapid development, testing, validation and adoption of OGC standards.

In April 2008, the OGC issued a call for sponsors for an OGC Web Services, Phase 6 (OWS-6) Testbed activity. The activity completed in June 2009. There is a series of on-line demonstrations available here:

http://www.opengeospatial.org/pub/www/ows6/index.html

The OWS-6 sponsors are organizations seeking open standards for their interoperability requirements. After analyzing their requirements, the OGC Interoperability Team recommended to the sponsors that the content of the OWS-6 initiative be organized around the following threads:

1. Sensor Web Enablement (SWE)
2. Geo Processing Workflow (GPW)
3. Aeronautical Information Management (AIM)
4. Decision Support Services (DSS)
5. Compliance Testing (CITE)

The OWS-6 sponsoring organizations were:

- U.S. National Geospatial-Intelligence Agency (NGA)
- Joint Program Executive Office for Chemical and Biological Defense (JPEO-CBD)
- GeoConnections - Natural Resources Canada
- U.S. Federal Aviation Agency (FAA)
- EUROCONTROL
- EADS Defence and Communications Systems
- US Geological Survey
- BAE Systems

- ERDAS, Inc.
- Lockheed Martin

The OWS-6 participating organizations were:

52North; AM Consult; Carbon Project; Charles Roswell; Compusult; con terra; CubeWerx; ESRI; FedEx; Galdos; Geomatys; GIS.FCU, Taiwan; GMU CSISS; Hitachi Ltd.; Hitachi Advanced Systems Corp; Hitachi Software Engineering Co., Ltd.; iGSI, GmbH; interactive instruments; lat/lon, GmbH; LISAsoft; Luciad; Lufthansa; NOAA MDL; Northrop Grumman TASC; OSS Nokalva; PCAvionics; Snowflake; Spot Image/ESA/Spacebel; STFC, UK; UAB CREAF; Univ Bonn Karto; Univ Bonn IGG; Univ Bunderswehr; Univ Muenster IfGI; Vightel; Yumetech.

Contents

Page

Figures

Page

Tables

Page

Listings

Page

OGC® OWS-6 SWE Event Architecture Engineering Report

1 Introduction

1.1 Scope

This OGC Engineering Report describes an abstract event architecture that can be applied to OGC web services. This architecture consists of different roles and interfaces defining general functionality in the event architecture.

This document defines the terms event, alert and their distinction. Furthermore the various flavors of event time and timestamps are discussed.

This document defines an application schema for events and gives an introduction to Event Processing, Complex Event Processing (CEP), Event Stream Processing (ESP) and the use of causal vectors used in complex events.

This document describes how the event architecture can be implemented using WS-Notification from OASIS. A REST based approach to the event architecture is not discussed in this document.

1.2 Document contributor contact points

All questions regarding this document should be directed to the editor or the contributors:

Name	Organization
Thomas Everding	Institute for Geoinformatics (IfGI), University of Münster
Johannes Echterhoff	International Geospatial Services Institute (iGSI)
Ingo Simonis	International Geospatial Services Institute (iGSI)

1.3 Revision history

Date	Release	Editor	Primary clauses modified	Description
07.12.2008	0.0.1	TE, JE	all	initial draft
17.03.2009	0.0.2	TE, JE	all	second draft
10.04.2009	0.3.0	TE, JE	all	pre-release version
17.04.2009	0.3.0	TE, JE	all	release version
2009-07-09	0.3.0	Carl Reed	Various	Get ready for publication

1.4 Future work

This report establishes the first version of an OGC Event Architecture. It does so by defining an abstract architecture and by providing guidance how this architecture can be implemented using existing standards. Several existing OGC specifications deal with aspects of an event architecture to a certain extent. These are, for example, the Sensor Alert Service (SAS), Sensor Event Service (SES) and Web Notification Service (WNS). While the former define a Publish/Subscribe approach for the Sensor Web domain in their specific ways, the latter provides functionality for relaying messages via various protocols.

Development of an Event Service Specification

We believe that a common Event Service specification can be developed, which implements Publish/Subscribe functionality in a generic way. This work will need to take into account existing standards and cope with the various flavors of architectural styles that are important to the OGC. The development of this service will need to carefully investigate the specific requirements of the geospatial domain with respect to Publish/Subscribe message exchanges.

The Event Service specification should ideally define interfaces or resources which can easily be added to OGC service specifications. Profiles or extensions could be defined to handle specific usage scenarios.

Investigation and Improvement of Subscription Models and Filter Languages

Various options of subscription functionality need to be investigated in more detail. This report provides an overview of possible subscription models. One of them is using filters to define the events of interest. The OGC Filter Encoding is one candidate for per-message content filters, while the Event Pattern Markup Language is an example of an OGC specification for enabling complex event processing. The applicability of these and other filter and processing languages (e.g. XPath 1.0 / 2.0, XQuery etc.) needs to be investigated in more detail and enhanced or adapted if required.

To improve support of event processing functionality in an OGC Event Architecture, the EML will need to be revised and extended. New versions of the EML should take into account the newest version of the OGC Filter Encoding and include spatial views and better support invocations of other services like a WPS. This will enable a higher flexibility when composing event processing models and lead to more reactivity of service oriented architectures.

Testing and Implementing the Event Architecture

This report describes possible ways to event-enable existing OGC services (see chapter 7). An upcoming OGC testbed should have a dedicated thread with the goal to prototypically realize the event architecture.

This would require the development of a larger use case that ideally integrates as many of the existing OGC web service specifications as possible. Data processing workflows, as

have been demonstrated before, could be the basis of such a use case. The implementation of the event architecture could benefit by leveraging Enterprise Service Bus (ESB) technology. The scenario could also be used to demonstrate on-the-fly (complex event) processing and transformations between different encodings and protocols.

The definition of an event taxonomy or hierarchy (see chapter 6.5.5) in support of the implementation of the OGC Event Architecture would be another aspect of the envisioned testbed thread. For this, the taxonomy examples presented in chapter 7 can be used as a starting point.

As this report mainly focuses on an abstract architecture and remote procedure call based interaction patterns it has to be investigated how the event architecture can be implemented according to REST and / or RESTful interaction patterns. First suggestions are described in chapter 8.2.3.

The applicability of WS-Notification for the well-known POX (Plain Old XML via HTTP) architectural style of OGC web services needs to be proved. A common approach applicable to all architectural styles should be the goal. This work should be done in a testbed thread as mentioned before.

An investigation of the relationship of resource management WS-* standards with REST should be performed, especially under the aspect of highest compatibility between the two approaches when HTTP is the application protocol / transport binding used.

SOAP 1.2 outlines how a web friendly use of SOAP would look like, using HTTP GET for information retrieval only, while POST should be used to invoke real operations. By constraining the resource identification information into the URI part of a WS-Addressing endpoint address and not into its reference parameters, higher web friendliness could be achieved. This should be experimented with in an OGC testbed, to improve the understanding of the different architectural styles, their commonalities and the options for harmonization when HTTP is used.

In addition, a comparison of OASIS WS-Notification and the emergent W3C WS-Eventing standard should be performed. An investigation of the OGC requirements for Publish/Subscribe support in OWS is required, with which the applicability of the two different WS-* solutions could be studied in more detail (once WS-Eventing has reached final standard status at W3C). Right now WS-Notification can be used to implement the Event Architecture. However, with respect to the discussions around WS-* and REST, a detailed comparison of the two approaches needs to be performed, taking into account the OGC requirements of a Pub/Sub solution as well as the web friendliness of the standards they depend upon and the available or required extensions for the two Pub/Sub specifications.

Develop and Implement Policies for OGC Web Services

Policies for web services are a means to specify the behavior of a service instance. This may for example enable clients to define whether notifications that match their subscriptions should be transmitted reliably or not. Policies allow services to indicate the

executed behavior but also which options clients have to modify the behavior. Clients may use policies to request specific behavior. It has to be investigated which behavior definitions (i.e., policies) are needed by OGC services today and how they can be integrated into the architecture. Policies to define subscription, caching, filter precision etc. can be imagined.

Enhancement of Bounding Information in Feature Encodings

The feature modeling in ISO 19136 will need to be corrected to support pure temporal types in the boundedBy property of features in addition to spatial and spatio-temporal types. This is necessary for an encoding of events as features.

Enhancement of Gazetteer to Handle Temporal Locations

The gazetteer specification will need to be extended so that also temporal locations can be identified (like "Backup 28278", "Version 2.0 Release Date"), just like spatial locations.

1.5 Changes to the OGC Abstract Specification

Topic 5 defines what the term feature means with respect to the OGC. When the modeling of geospatial features is discussed, the focus lies upon "Features with Geometry", "Features as Coverages" and "Features as Observations". It should be emphasized that - according to ISO 19101 - features can also be events as defined in this report. An event is also intimately related to the other feature types but represents a different concept.

The OGC community should discuss whether the concepts of the abstract event architecture covered in this report are generic enough to incorporate them in a new topic of the Abstract Specification.

2 References

The following documents are referenced in this document. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. For undated references, the latest edition of the normative document referred to applies.

ISO 8601, *Representation of dates and times*.

ISO 19101, *Reference model*.

ISO 19108, *Temporal schema*.

ISO 19109, *Rules for application schema*.

ISO 19112, *Spatial referencing by geographic identifiers*.

ISO 19115, *Metadata*.

ISO 19125-2, *Simple feature access – Part 2: SQL option*.

ISO 19136, *Geography Markup Language (GML)*.

OGC 06-009r6, *OpenGIS® Sensor Observation Service Implementation Specification*

OGC 06-028r5, *OpenGIS® Sensor Alert Service Candidate Implementation Specification*

OGC 07-014r3, *OpenGIS® Sensor Planning Service Implementation Specification*

OGC 07-022r1, *OpenGIS® Observations and Measurements – Part 1 – Observation schema*

OGC 07-036, *OpenGIS® Geography Markup Language (GML) Encoding Standard*

OGC 08-126, *OpenGIS® Abstract Specification Topic 5: Features*

OGC 08-133, *OpenGIS® Sensor Event Service Interface Specification (proposed)*

3 Terms and Definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Specification [OGC 06-121r3] and in OpenGIS® Abstract Specification Topic 5: Features shall apply. In addition, the following terms and definitions apply.

3.1
abstraction
An event is an abstraction of a set of events if it summarizes, represents, or denotes that set of events. [Luckham, Schulte 2008]

3.2
action
An action is a special type of event that is actively performed by a subject.

3.3
architecture
The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. [Luckham, Schulte 2008]

3.4
architecture style
A coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style. [Luckham, Schulte 2008]

3.5
asynchronous communication
Communication pattern where the initiator of the communication does not block its program flow in order to wait for an answer.

3.6
broker
A role in the event architecture. A broker is a specialization of a router and offers an additional subscription interface.

3.7
causal vector
causality
Data structure of a complex event that stores all member events that are related from the complex event via the caused-by relationship.

3.8
causality model
event causality model
A causality model is a classification scheme for events using caused-by relationships.

3.9
cause
An event A is a cause of another event B, if A had to happen in order for B to happen. [Luckham, Schulte 2008]

3.10
communication pattern
A communication pattern is an abstract description of a communication sequence.

3.11
complex event
An event that is an abstraction of other events called its members. [Luckham, Schulte 2008]

3.12
complex event processing
Computing that performs operations on complex events, including reading, creating, transforming or abstracting them. [Luckham, Schulte 2008]

3.13
composite event
A derived, complex event that is created by combining base events using a specific set of event constructors such as disjunction, conjunction, sequence, etc. [Luckham, Schulte 2008]

3.14
consumer
event sink
A role in the event architecture. A consumer receives notifications from other components.

3.15
derived event
synthesized event
An event that is generated as a result of applying a method or process to one or more other events. It is a specialization of a complex event. [Luckham, Schulte 2008]

3.16
event
Anything that happens or is contemplated as happening at an instant or over an interval of time.

NOTE: The term event may also be used for event objects. The current meaning depends on the context.

3.17
event attribute
event property
A component of the structure of an event. [Luckham, Schulte 2008]

3.18
event channel
event topic
A conduit in which events are transmitted from event sources (emitters) to event sinks (consumers). [Luckham, Schulte 2008]

3.19
event cloud
A partially ordered set of events (poset), either bounded or unbounded. [Luckham, Schulte 2008]

3.20
event-driven
The behavior of a device, software module or other entity whose execution is in response to the arrival of events from external or internal sources. [Luckham, Schulte 2008]

3.21
event-driven architecture
An architectural style in which some of the components are event driven and communicate by means of events. [Luckham, Schulte 2008]

3.22
event hierarchy
An event hierarchy is an event taxonomy with event property definitions.

3.23
event object
An object that represents, encodes, or records an event, generally for the purpose of computer processing. [Luckham, Schulte 2008]

3.24
event pattern
A template containing event templates, relational operators and variables. An event pattern can match sets of related events by replacing variables with values. [Luckham, Schulte 2008]

3.25
event pattern language
event processing language
A high level computer language for defining the behavior of event processing agents. [Luckham, Schulte 2008]

3.26
event processing
Computing that performs operations on events, including reading, creating, transforming and deleting events. [Luckham, Schulte 2008]

3.27
event processor
A component that performs event processing.

3.28
event setting
The temporal or spatio-temporal characteristics of an event.

3.29
event stream
A linearly ordered sequence of events. [Luckham, Schulte 2008]

3.30
event stream processing
Computing on inputs that are event streams. [Luckham, Schulte 2008]

3.31
event taxonomy
A classification scheme for events using parent-child relationships.

3.32
event type
A class of event objects. [Luckham, Schulte 2008]

3.33
feature
An abstraction of real world phenomena [ISO 19101]

3.34
genuine event properties
Event attributes that are not changeable after creation.

3.35
geographic feature
Representation of a real world phenomenon associated with a location relative to the Earth. [ISO 19125-2]

3.36
notification
message
A container for event objects.

3.37
occurrence
A special type of event that is happening to a subject.

3.38
poset
A partially ordered set.

3.39
producer
A role in the event architecture. A producer is a specialization of a publisher that offers an additional subscription interface.

3.40
publisher
A role in the event architecture. A publisher sends notifications to other components.

3.41
router
A role in the event architecture. A router receives notifications from and sends notifications to other components.

3.42
simple event
An event that is not an abstraction or composition of other events. [Luckham, Schulte 2008]

3.43
service oriented architecture
An architecture where the components are deployed as services.

3.44
synchronous communication
Communication pattern where the initiator of the communication does block its program flow in order to wait for an answer.

3.45
window
(data) view
A bounded portion of an event stream. [Luckham, Schulte 2008]

4 Conventions

4.1 Abbreviated terms

AI	Artificial Intelligence
AIM	Aeronautical Information Management
AIXM	Aeronautical Information Exchange Model
API	Application Programming Interface
BAM	Business Activity Monitoring
CEP	Complex Event Processing
ED-SOA	Event Driven SOA
EDA	Event Driven Architecture
EML	Event Pattern Markup Language
ep-ts	Event Processing Technical Society
EPL	Event Pattern Language
ER	Engineering Report
ESP	Event Stream Processing
GFM	General Feature Model
GML	Geography Markup Language
GMT	Greenwich Mean Time
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ISO	International Organization for Standardization
Java AWT	Java Abstract Window Toolkit
KVP	Key Value Pair
MOM	Message Oriented Middleware

O&M	Observations and Measurements
OASIS	Organization for the Advancement of Structured Information Standards
OGC	Open Geospatial Consortium
OGC AS	OGC Abstract Specification
OGC RM	OGC Reference Model
OMG	Object Management Group
OS	Operating System
OSI	Open Systems Interconnection
OWS	OGC Web Services
POX	Plain Old XML
Pub/Sub	Publish / Subscribe
QoS	Quality of Service
RDF	Resource Description Framework
REST	Representational State Transfer
RFID	Radio Frequency Identification
RTP	Real-Time Transportation Protocol
SAS	Sensor Alert Service
SDI	Spatial Data Infrastructure
SES	Sensor Event Service
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SOS	Sensor Observation Service
SPS	Sensor Planning Service
SQL	Structured Query Language
SWE	Sensor Web Enablement
SWES	SWE Common Service

TC	Technical Committee
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
VoIP	Voice over IP
W3C	World Wide Web Consortium
WCS	Web Coverage Service
WFS	Web Feature Service
WMS	Web Map Service
WNS	Web Notification Service
WPS	Web Processing Service
WS-*	Web Services -*
WS-I	Web Services Interoperability Organization
WS-N	WS-Notification
WSA	WS-Addressing
WSDL	Web Service Description Language
WSN-B	WS-BaseNotification
WSN-BR	WS-BrokeredNotification
WSN-T	WS-Topics
WSRF	WS-Resource Framework
WWW	World Wide Web
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XSLT	Extensible Stylesheet Language Transformation

4.2 UML notation

Some diagrams that appear in this standard are presented using the Unified Modeling Language (UML) static structure diagram, as described in Subclause 5.2 of [OGC 06-121r3].

5 Overview

This engineering report begins with the specification of an abstract event architecture for the OGC service suite (chapter 6). The chapter includes the development of an OGC event definition and application schema for events as well as the actual abstract event architecture specification in terms of roles and interfaces.

In chapter 7 the mapping of the abstract architecture to existing OGC services is demonstrated. The chapter also includes the presentation of different capabilities of the architecture using various use cases.

Chapter 8 contains introductions and discussions of various techniques that are related to the event architecture and event driven systems in general, like important communication principles and event processing.

The conclusion of this report is provided in chapter 9.

Annex A provides a tutorial of WS-Notification, a standard developed by OASIS which can be used to implement the OGC Event Architecture for the SOAP / WS-* architectural style.

6 Abstract Event Architecture

6.1 Introduction

In this chapter the abstract event architecture is developed and presented. It is abstract because it does not specify service interfaces that can directly be implemented. It rather defines the capabilities and behavior of the architecture components. It is not called a pure Event Driven Architecture (EDA) since it does not solely make use of push-based (also called notification based) communication patterns as pure EDAs do, instead it extends the existing OGC architecture (see chapter 7).

This chapter begins with an overview of different definitions for the term event followed by the development of an OGC event definition which is used throughout this report. In addition, related terms like notification and alert are specified and distinguished. An application schema for events is also developed, which may serve as the foundation for encoding events in OGC services. Finally the abstract event architecture is defined using different roles and interfaces for services and other components.

6.2 Event Definitions Overview

There are many definitions for the term *event*. For the definition of the OGC wide *event architecture* it helps to understand these definitions and their (sometimes subtle) differences. This report therefore presents and discusses exemplary definitions of the term event, which will lead to the definition that will be used in this report.

NOTE: In the following, we will explain various event definitions and their specifics. If you, the reader of this report, are not interested in this background information, feel free to skip this chapter and go directly to the OGC event definition chapter (6.3).

6.2.1 Allen and Ferguson

The first definition introduced in this report is from James F. Allen and George Ferguson. They say that "[...] events are primarily linguistic or cognitive in nature. That is, the world does not really contain events. Rather, events are the way by which agents classify certain useful and relevant patterns of change" (Allen, Ferguson, 1994).This means that for example "rainfall" may be an event in some application, another uses lots of "raindrop falling" events. So it is up to the domain expert to define and classify the events - with the required properties - that occur within a given domain. The paper - being concerned primarily with questions of AI and an appropriate event model for time-aware agents - also provides some more interesting information about events and related terms.

– Actions and events take time. Some events are instantaneous, but most occur over an interval of time.

– Actions and events may have effects. For example, driving a car results in the consumption of gas and the car altering its location in space-time.

– Actions and events may interact in complex ways when they overlap or occur simultaneously. For example pressing only shift-up has usually no effect when editing text, pressing only a letter like 'h' will add an 'h' at the current cursor position, while if both shift-up and 'h' are pressed simultaneously, a capital 'H' will be added to the text. Other examples are for instance rogue waves[1]. This shows that a causality between different events can exist.

– An event might depend on the happening of external events and pre-conditions. For example, sailing across a lake is only possible if there is enough wind - and if the lake has enough water in it for the boat to swim, there are sails on the boat, the sails have been raised etc. External events may also be further distinguished into triggered, definite and spontaneous events. The difference is in what is causing the event: a triggered event obviously requires a triggering action (willingly performed by some entity, e.g. "the operator triggered the alarm by pushing a button") while a definite event does not require such an action, i.e. it is assured to happen in time (e.g. "sun rising in the morning"). A spontaneous event is not guaranteed to happen, it may occur at any time.

– There is a distinction between states and events. While the former is *temporally homogeneous* over an interval of time (i.e. if a state holds in one time interval, it also holds during that interval) the latter is *temporally anti-homogeneous* (i.e. an event that occurred over an interval of time did not occur during that interval, because the event would not have been completed yet).

We see that an event may be a complex entity with many specific properties. It will depend upon the given use case which of the specific properties like 'effect' or 'causing event' need to be incorporated in the event model.

6.2.2 The Internet Encyclopedia of Philosophy

The Internet Encyclopedia of Philosophy contains an article about events that discusses multiple event theories. It starts with a common definition: "Events are particular happenings, occurrences or changes, ..." The event theories in that article mainly deal with identity conditions on events like "if Brutus kills Caesar by stabbing him, are there two events, the stabbing and the killing, or only one event?" The answer to this question is not the goal of this report, as the decision on what is an event and what is not has to be made by the designers of an event architecture implementation and depend upon the perspective of domain experts. Thus, it is only to be assured that both approaches (that Brutus killing Caesar involved only one event or at least two) are valid in our event architecture. Some aspects of the theories are presented in the following clauses because they help to improve the understanding of the term "event" and events themselves. Note that these aspects are parts of a larger philosophical discussion about events. The complete article is available at http://www.iep.utm.edu/e/events.htm.

1 see http://en.wikipedia.org/wiki/Rogue_wave

6.2.2.1 Jaegwon Kim's theory

Kim defines events as follows: "Events are structured: they are constituted by an object (or number of objects), a property or relation, and a time (or an interval of time)". This leads to a notation for events "[x, P, t]", read the unique event x is having P at t. Of further interest is a condition about the existence of events: "[x, P, t] exists iff object x exemplifies the n-adic property P at time t". This means that an event always has an associated object. These objects may be virtual objects for instance in a simulation environment or again an event for instance when sending a notification of a detected event. The theory furthermore contains some statements of events which are listed partly below:

1) Events are non-repeatable, concrete particulars, including not only changes but also states and conditions.

2) Each event has a spatiotemporal location.

3) Although events may exemplify any number of properties, only one property, the constitutive property, individuates the event.

4) Two events are identical if they have the same constituting object (or number of objects), property or relation, and time (or interval of time).

The third statement means that events have some kind of primary key (like used in relational databases) and also that there may be multiple events associated to a single object at the same time but with different properties. For example stabbing and killing Caesar may be modeled as two events. Note the difference in usage of the terms event and object here and in IT (the latter meaning the event representation while the former denotes the real-world phenomenon).

6.2.2.2 Donald Davidson's theory

Davidson also developed conditions for sameness and difference of events. He also understands events as "particular, non-repeatable occurrences" (compare to Kim's first statement of events). After discarding this definition of sameness, Davidson came up with a new proposition which states that "no two events can occur in exactly the same space-time zone". In the Caesar example this would result in just one event and the use of the spatiotemporal location as the only primary key property.

6.2.2.3 David Lewis' theory

In his theory of events Lewis defines events the following way: "(An entity) e is an event *only if* it is a class of spatiotemporal regions, both thisworldy (assuming it occurs in the actual world) and otherworldy". This means that spatiotemporal regions have events as properties, because he uses property synonymously with class and "to have a property" is used synonymously with "to be member of a class". As an example the entity "rainfall" happens at multiple spatiotemporal regions or the other way around: multiple spatiotemporal regions have the property (are member of the class) "rainfall". Therefore the entity "rainfall" is an event. This definition leads to the idea of event hierarchies

which are presented in chapters 6.5.4, 6.5.5 and 8.5. This definition also allows that spatiotemporal regions have multiple events as properties which is contrary to Davidson's second proposal. Lewis' definition of events also contains the definition of non-identity: two events x and y are not identical "if and only if there is at least one member of x that is not a member of y (or vice versa)". He also elaborates on causality of events, stating that events may be related to each other, *causing* being a strong relationship.

6.2.2.4 Summary

Philosophy has much to give when it comes to the definition of a specific term like event, but as we can see it is hard for philosophers (and not only for them) to agree upon a common definition of this term. Sometimes philosophers are very specific in their definitions which then are often argued to be too fine-grained. What can be said is that even the philosophy recognizes a lot of aspects that will constitute our definition of an event. We also see that it depends on the view of the domain expert on what constitutes an event and how fine-grained it has to be.

6.2.3 Worboys, Galton and Hornsby

Worboys defines events as occurrences or happenings. Besides an event, an occurrence may also be a process or an action. He states that "one person’s process is another’s event, and vice versa" and therefore the differences are philosophical and linguistical (Worboys 2005). From the perspective of an information system events are occurrences while objects are continuants (not to confuse with event objects as an information system's representation of events) (Worboys, Hornsby 2005).

Galton and Worboys further differentiate between objects, processes and events. They state that objects and processes may change. Examples are the color that is shown by a traffic light (change of an object) or a change in traffic flow from fast to slow (change in a process). Events however are not subject to change. They represent "a completed episode of history" with fixed properties. For instance an earthquake event has properties like severity, region, time etc. These "genuine" properties are not changed later on.

There may be changes to events but they belong to relational properties. For example, an earthquake event changes from the most recent earthquake to the second most recent but the severity does not change. Another reason for change may be the correction of errors in an event representation, where "genuine" properties may be changed. The properties of the event as it happened in the real-world does not change, one can only correct the representation of that event (Galton, Worboys 2005). For example if the earthquake was reported to have magnitude 5.4 but later on an error in the sensor measuring that magnitude has been corrected, a notification correcting the information of the first report can be issued.

6.2.4 Design Patterns

In computer programming, design patterns play an important role as they represent reusable solutions to common software design problems. One of them is the *observer pattern* (Gamma et al. 1995). Here, an object called the *observer* can express interest in

receiving notifications about events that occur at another object, called the *subject*. Usually the observer is just notified that something happened at the subject and will therefore have to access the subject to retrieve its current state. Depending on the implementation the current state or state change can also be contained in the notification itself so that the observer does not need to access the subject. Note that usually the observer is not able to fully specify which state changes, i.e. situations, it is interested to be notified about. The general approach is to let the observer decide whether a state change is of interest to him by accessing the current state of the subject after a state-change notification has been received.

Subjects can have zero to many observers. This shows that events may occur in many parts of a program but if either the implementation itself does not foresee the publication of such events or there are no interested observers then the events pass unnoticed. The observer pattern is realized in many Graphical User Interfaces.

6.2.5 Graphical User Interfaces

Graphical User Interface (GUI) operating systems make use of events to interact with applications. These applications need to be implemented according to an *event-driven* model, meaning that they do nothing unless an event occurs. The operating system (OS) generates an event if it recognizes that a *situation of interest* to an application has occurred - like a mouse movement or click - and sends a *notification* to the application via method or procedure calls. The application is free to react to the event *notification* in whichever way it is programmed to do - even by doing nothing. (Faison 2006) The application can be modeled as an observer and the operating system as the subject.

6.2.6 Unified Modeling Language

In UML "*an event describes a set of possible occurrences; an occurrence is something that happens that has some consequence within the system*" (OMG 2007). State machines are good examples of where events are used in UML. Here, events can cause transitions from one state to another: "*A transition is a directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type.*" (OMG 2007)

Guards can be applied to only enable transitions between two states if the triggering event satisfies the guard expression. In addition, each transition may also have an effect assigned to it. If the transition fires (i.e. an event occurs that satisfies the guard) then the behavior for the transition is performed, meaning that one or more actions are performed, e.g. sending signals or invoking operations.

6.2.7 Common Event Expression

Heinbockel et al. define an event as "observable situations or modifications within an environment that occur over a time interval." (Heinbockel et al. 2007). The mentioned time interval may have zero length (hence be a time instant) as they explicitly allow state changes as events. They also state that the origin of an event may be a single component

or the interaction of multiple components in a system. Furthermore they introduce "differing levels of abstractions" (see chapter 6.5.5) and simple as well as aggregated, correlated events (see chapter 8.5, Heinbockel et al. 2007).

6.2.8 ISO 19100

In ISO there are several definitions for the term *event*. In ISO 19136 (which is the same as GML 3.2.1, OGC 07-036), an event is "*an action that occurs at an instant or over an interval of time*". Another - different - definition is given in ISO 19108, where an event is "*an action that occurs in an instant*", an instant being "*a zero-dimensional geometric primitive representing position in time*". More specifically, in ISO 19108 an instant is an interval whose duration is less than the resolution of the time scale. The main difference of these definitions is the time aspect. The previously used example "rainfall" would be a valid event in ISO 19136 but not in ISO 19108, where a valid event would be "rainfall starts" and "rainfall ends" (also valid for ISO 19136).

ISO 19136 also states the following:

> *"As time is a one dimensional topological space, temporal topology primitives shall be a time node corresponding to an instant, and a time edge corresponding to a period. A time node is an abstraction of an event that happened at a certain instant as a start or an end of one or more states. A state is a condition — a characteristic of a feature or data set that persists for a period. A 'static feature' [...] means a feature that holds a consistent identifier during its life span. Time edge is an abstraction of a state, and associates with time nodes representing its start and end. However, temporal topology primitives do not directly indicate "when" or "how long". A time node need not be a start or an end of a time edge in the case of describing the event not associating with states. Such a node is called an isolated node."*

Applying this definition to the rainfall example, begin and end of the rainfall would be modeled as time nodes with a time edge connecting them. The complete rainfall, encoded as a time edge, would then represent a state of the environment, while the time nodes would represent the events marking the beginning and end of that state.

6.2.9 Event Processing Technical Society

The Event Processing Technical Society (ep-ts) defines an event as "*anything that happens or is contemplated as happening*" (Luckham, Schulte 2008). In this definition the term event is overloaded with two meanings. Firstly "event" denotes the real or simulated phenomenon or activity and secondly its representation by an object in a computer (also called event object). The current meaning of the term “event” is indicated by the context (Luckham, Schulte 2008).

6.2.10 Summary

As we can see there are a lot of different definitions of what an event is. Sometimes an event is defined as an occurrence or situation of interest while sometimes it is defined as

an action that occurs. Often, time is explicitly stated as an important property of the event while sometimes it is implicitly provided and only relative timing plays a role. There are many aspects when talking about events that are important in different use cases. With the goal of defining an OGC Event Architecture, this report has to provide a good definition of the term event and also which information an event may contain. In the next chapter, we will develop this common OGC event definition.

6.3 OGC Event Definition

As we have seen in the previous chapter, a consistent definition of the term event does not exist. For the discussion about and the creation of an OGC wide event architecture, we thus need to develop our own definition. This definition should be compatible with the OGC baseline, i.e. the OGC Reference Model (OGC RM) and Abstract Specification (OGC AS). In addition, it should be consistent with ISO TC 211 nomenclature. In this report, the term *event* is defined as follows:

> *An event is anything that happens or is contemplated as happening at an instant or over an interval of time*.

The definition is a mix of those given by Luckham and Schulte (2008) and ISO 19136. It emphasizes the fact that an event has a strong temporal aspect and may represent anything that happens in the real world but can as well be simulated or happen in software. The term *happening* encompasses *action*, *occurrence* and *situation of interest*, *state change* etc. which all represent something that happens.

6.3.1 Modeling Events

Which happenings need to be represented by an application depends upon the use case and the application domain. Of course, not all happenings in the world will be relevant in a given domain. Thus, when modeling events the domain expert needs to specify all relevant event types together with their inheritance relationships, properties and associations.

This process is related closely to modeling features, as described in OGC AS topic 5. In fact, treating events as features is a valid modeling perspective because an event is a type of feature in accordance with ISO 19101 and the General Feature Model (GFM) defined in ISO 19109. The description of a feature given by the OGC AS topic 5 also allows events to be modeled as features, although this is not explicitly stated there.

Feature types usually have a set of properties which define the characteristics of the given feature in a domain. Because two domains will probably not share the same view upon a certain feature like "river", the types defined by each domain for that feature will probably have a different set of properties. This is the same for modeling events except that events have one property in common: a property that provides the time of the happening represented by the event (see clauses 6.3.4.1 and 6.3.4.2.2). To emphasize different aspects of a happening, multiple event types can be defined. Thus, an event (a

happening) may be represented by multiple *event objects*, being instances of multiple event types.[2] For example, a disaster like a tsunami may encompass event types like *WaterDisplacement* and *Landfall*, each with their specific properties. In this example, there might be only one WaterDisplacement event but multiple Landfall events.

All of these events together may constitute the *Tsunami* event (depending, of course, on how the domain models a tsunami). This shows that an event may be associated to other events and of course to other objects like features.

When modeling an event type, it is essential to keep in mind that an event is *temporally anti-homogeneous* (Allen & Ferguson 1994). This invariant defines that an event which happened over an interval of time did not happen during that interval, because the event would not have been completed yet. Designers have to make sure that the definition of an event type is precise so that this invariant is met for each instance of that type.[3] If necessary, an iterative redesign of the event type definition has to be performed.

In the following paragraphs, we will discuss event inheritance, constraints, properties and associations in more detail.

6.3.2 Event Inheritance

Creating event hierarchies through specialization of generic types is a robust approach to model the events relevant for a given domain. This resembles the definition of feature type hierarchies. The concept is especially useful when subscribing to events that are of a given type which we will explain in chapter 8.1.3.1.

Luckham and Schulte (2008) provide a useful classification of events which we adopt in this report to some extent. They say that an event which is neither an abstraction nor a composition of other events is a *SimpleEvent*. Whenever an event is an abstraction of other events, it is called a *ComplexEvent*. A ComplexEvent can but does not have to list the member events of which it provides an abstraction. For example, there is no accepted agreement as to which events are members of the 1929 *StockMarketCrash* (Luckham, Schulte, 2008). The member events of a ComplexEvent are determined by some kind of assignment. Whenever an event is generated as a result of applying a well-defined procedure to one or more other events, we are talking about a *DerivedEvent*. An example would be a *ThresholdCrossed* event that is caused by two observations, one with a value below the threshold, the other with a value above. When the procedure for combining base events is disjunction, conjunction, sequence, etc. then Luckham and Schulte (2008) call it a composite event. We do not model this as a concrete specialization of DerivedEvent. Rather, we treat it as a simple classification that is implied by the procedure used in a DerivedEvent.

---

[2] The term event object is used to distinguish the (digital) representation of an event from the happening itself where this distinction is necessary and not obviously given by the context.

[3] Note that uncertainty is likely involved when assigning time stamps to an event, due to computational and observational uncertainty. This inaccuracy cannot be avoided. It depends upon the application domain which amount of uncertainty and inaccuracy is allowed

6.3.3 Event Constraints

Constraints may be defined for feature types and thus also for event types. Examples of possible constraints are pre-conditions that need to be met for an event of a given type to be created. To exemplify this, think of the following use cases. In order for a *PlaneLanding* to happen, at some point in time before that event a *PlaneTakeoff* must have happened[4], so the constraint could require an association to such an event. Another example is the *WaterFreezes* event - in order for water to freeze the water temperature needs to be below or equal to zero degree Celsius. Such pre-conditions can be modeled when defining an event type.

6.3.4 Event Properties

The General Feature Model (GFM) defined in ISO 19109 lists operations, attributes and association roles as possible property types of a feature type. An event type will have a well-defined set of these properties which we will discuss in the following.

Note that like ISO 19136, we will follow RDF (W3C, 2004) terminology and use the term property rather than attribute or association role.

6.3.4.1 Operations

Only one property is mandatory for all event types: the operation to retrieve the *event time*. This time defines when an event / happening took place. It is of type time instant or time interval. Each event type needs to define how the event time is computed. In simple events, it may be encoded as a property (see clause 6.3.4.2.2) while in complex events (see clause 8.5.4) the event time may need to be computed on-the-fly from the members that are associated to the event.

Thus, an event type definition does not necessarily need to incorporate a temporal property which represents the event time, but it needs to have the *getEventTime* operation. This operation will either simply access the value of the temporal property representing the event time or compute the event time as defined for the given event type.

A specific event type may also have more operations. Which additional operations are available is up to the domain expert modeling the event type.

6.3.4.2 Attributes and Association Roles

An event may contain a set of properties which correspond to the feature attribute and feature association role in ISO 19109. Like ISO 19136, we treat both feature attributes and association roles as feature properties and will only distinguish when necessary.

[4] Of course, the events must refer to the same plane; for consistency, the landing should not just reference any takeoff of that plane.

6.3.4.2.1 Fixed vs. Dynamic Properties

An event object has genuine properties which are fixed. These include the *event setting* as well as any other properties that describe the happening represented by the event object. The event setting provides the temporal or spatio-temporal characteristics of an event. With fixed, we mean that the value of an event object's property cannot be modified once that object has been created.
This does not mean that an event property may not be time-dependent (which differs to the event definition given by Galton and Worboys [2005]). An event may very well have a property with a coverage value. For example, a flood will have different extents at different times. Given a time instant, the coverage will provide the flood extent which was valid for that time.

Note that we are explicitly talking about event objects here, not events in general. A happening has unique properties. We may observe or measure the values of these properties using some procedure (compare OGC 07-022r1). The procedure with which we assign these values might not be exact (for various reasons, including measuring imprecision and inaccuracy, inexact parameter values etc). These observation errors in event property values may of course be corrected when more exact values become available. Such a correction does not modify the property values of a previously generated event object, though. In case that updated property values are available, a new event object shall be created, which can be viewed as an updated representation of the event (and it may have a relationship to the previous event object). Thus, the genuine properties of event objects are fixed – however, they can be updated as described.

Dynamic event properties are those which may change at some point in the future. The *Earthquake* event provided in clause 6.2.3 is an example for inclusion of a dynamic property. The event may be the most recent earthquake when it happened, but it may change this category when a more severe earthquake happened later on.

6.3.4.2.2 Temporal Properties

According to our event definition, an event is a happening which takes place at a specific time. This time is called the event time. In clause 6.3.4.1, we said that the event time is mandatory for all events. However, it only needs to be accessible via the mandatory *getEventTime* operation. An event type does not need to incorporate a property type that characterizes the event time. Thus, this temporal property may be optional in event types.[5] However, a property for the event time shall always either be of type time instant or time interval.

Note that we are using time interval instead of time period for the event time to point out that an event object is *non-repeatable*. This is obvious if you think about the fact that a happening is finished at a certain point in time and that the exact same happening will never occur again. An example is the *Sunset* event which is perceived as happening at a

[5] An optional property that characterizes the event time supports the lazy loading pattern. (see http://en.wikipedia.org/wiki/Lazy_loading).

certain location. Each event object of type *Sunset* is different in that it has a different event time. Event identity will be discussed in clause 6.3.6.

Our event definition also uses the present form ("anything that happens or is contemplated as happening ...") so it may be read to include events that are ongoing. An example of an ongoing event is a *Thunderstorm* in a given area. The beginning of such a storm might be given with the first lightning strike and while the thunderstorm is ongoing you might increase the counter for lightning strikes. The end of the thunderstorm is not known exactly, but as you know that it will end eventually, you could create a temporary event object. Depending upon your domain, you may publish the event to other actors in your system. However, the encoding of your event object should clearly indicate that what you sent is only temporary (in this case, because the end time is indeterminate). The strongest representation of an event is an event object which represents an event that has already happened (i.e., is finished) and of which the complete set of genuine properties can therefore be provided with the highest degree of quality. A weaker representation is that of an event which is ongoing (like the thunderstorm in our example) - here you are able to measure some properties with a high degree of quality but as the event is not complete, the set of genuine event properties can also not be complete. The weakest representation is an event object which describes an event that will happen in the future. If an event object does not describe an event that has happened (in the past), it should indicate that it is either temporary or simulated.

Other temporal properties can be added to an event as well (see chapter 8.6). Examples of additional temporal properties are the time when an event object was created (*creation time*) and the time when an event object was received by a software component (*arrival time*) (Luckham, Schulte, 2008). All these temporal properties may be needed for subsequent computations (like Event Processing, see clause 8.5).

Note that in a distributed environment, systems that generate events may use different temporal reference frames and clocks. Temporal event properties should always indicate which reference frame and clock they are using so that other systems may take this information into account when processing events.

6.3.4.2.3 Spatial Properties

Events may have spatial properties with different semantics. If an event happened in the real world, then usually it has a spatial attribute which provides information on where the event took place. This property would be part of the spatio-temporal setting of the event (see clause 6.3.4.2.1).

The spatial setting may be expressed with a static or dynamic geometry. The former will be used if the event happened at a time instant or may be used if the event happened in a region of space which does not change during the event's time interval. A dynamic geometry is possible for events with time interval, where the spatial extent of the happening varies over time (think of the flooding example). A dynamic spatial property of a spatio-temporal event setting can be modeled as a coverage. In this case, the coverage should provide a geometry for each instant contained in the time interval that represents the event time. Because this is not always possible – think of discrete

coverages – some kind of estimation (like interpolation, etc.) may become necessary to get the geometry for each instant in the event time interval.

As for temporal properties, the required accuracy of geometries contained in spatial properties of an event object is determined by the domain that created it. This may lead to a generalization of a dynamic geometry (like the extent of a flooding) to a static one (could be the complex hull or bounding box of the dynamic property).

6.3.4.2.4 Location Properties

Instead of an actual geometry, an event might as well just provide a unique identifier or place name to represent a spatial event property. This identifier or place name can then be sent to a gazetteer (see ISO 19112) to resolve the associated spatial location.

Note that this concept is currently not supported by ISO 19112 for identifiers that represent a temporal location. For example, the identifiers "Backup 28278" and "French Revolution" could be resolved to their temporal location just like "New York" and "General Sherman Tree" can be resolved to their spatial location[6].

6.3.4.2.5 Thematic Properties

All event properties are identified by the domain expert who designs the event types relevant to his domain. In the previous paragraphs we discussed aspects on how to model temporal and spatial event properties, especially how the event setting should be modeled. An event usually has but does not need to have thematic properties which describe a specific happening. For example, in some cases the type of the event itself may already be sufficient to give meaning to an event, like a *Robbery* with a spatio-temporal setting of *19th of May 1995*, *Federal Reserve Bank of New York*. In general, a domain expert is free to add any thematic properties to an event type, e.g. the conditions that led to the event or effects an event had.

6.3.4.2.6 Metadata Properties

Metadata may also be added to an event type and object, for example to indicate which organization created the event and where the original event is maintained. The latter can be useful in situations where only a copy of an event object is distributed to other systems (which therefore only have a snapshot of that event representation - keep in mind that event properties and also the associations of an event can be dynamic). In this case, the most recent event representation could always be retrieved by other systems. Also, if an event object contains spatial identifiers instead of the actual location, a reference to a gazetteer service with which to resolve that identifier could be added to the metadata. Possible types of metadata are described for example in ISO 19115.

---

[6] The exact definition of the temporal or spatial location depends upon the domain which assigned the identifier.

6.3.5 Event Associations

Events can have associations to other events and of course to other objects, like specific features, observations etc. For example, a *Fire* event may aggregate the observations (like temperature, smoke and IR readings) that were used for detecting the fire. Events may be used to compute new information, probably resulting in a new event that references the base events (a common use case in Event Processing - see clause 8.5). In the fire example, the *Fire* event may be the base for an *Evacuation* event.

Associations with other objects may be added to an event object during its lifecycle. For example, one can establish the relationships between a *Flood* event and all features that were affected by that event. Thus, associations are typical candidates of dynamic event properties.

The examples in this paragraph already provide a glimpse upon the many relationship roles between an event and associated object. Numerous other roles are possible. Which relationship roles exist and which need to be modeled depends, of course, upon the given domain. Nevertheless, Worboys and Hornsby (2004) as well as Galton and Worboys (2005) identified a list of roles that can be used across domains (see Table 1).

Table 1 – (Incomplete) List of Possible Event Relationships (Worboys & Hornsby 2004, Galton & Worboys 2005)

Target -> Source	Relationship
Event -> Event	cause, initiation, perpetuation (or facilitation), hindrance (or blocking), termination, supersedes
Event -> Object	creation, sustaining in being, reinforcement, degradation, destruction, splitting and merging
Event -> State	initiates, terminates
State -> Event	allows, prevents

6.3.6 Event Identity

Because an event is a feature, it has identity. To assign an identifier to an event is the responsibility of the organization that detects the event. This is an easy task. However, to determine if two event objects represent the same event can become difficult. So let us look at this in more detail.

First of all, we can discuss whether two event type definitions are actually identical in that they describe the same type of happening. Think of the example that causes so much trouble to philosophers: is the stabbing (of Caesar) the same as the killing (of Caesar)? This is a semantic problem which will likely be resolvable in one domain but can become quite difficult across domains.

Because the solution of such a semantic problem is out of scope for this report, we will follow a pragmatic approach. Two event objects represent the same event if they:

– have the same (scoped) identifier

– have the same scoped name

– have the same set of properties with the same values

If the same scoped name is found in two event objects with differing properties and / or property values, then the two event objects may be considered as representations of the same event - with different event types or different level of detail. For example, event object B may have more properties than event object A, because it was generated after A, when more information about the event was available.

6.3.7 Relationship of Events to Features and Observations

We introduced events to be specialized feature types. So how are feature types like the O&M observation related to events? We said that to be an event, only the getEventTime operation needs to be implemented (see clause 6.3.4.1). Thus, only this operation needs to be implemented by classes which are considered to be events. This can be done explicitly in a given domain model (like an application schema) or implicitly be added by specifying how the operation would be implemented. The latter is a weak approach but it allows treating specific classes in already existing application schema and domain models as events.

For example, an O&M observation represents the happening of sampling values for a property of a given feature of interest via a given procedure; so an observation is an event where the event time is provided by the sampling time. Thus, we can specify that the getEventTime operation is implemented by an observation and simply returns the value of the sampling time property. We can thus say that all observations are events. The opposite is not true.

6.4 Related Terms

In this chapter further definitions for terms associated with events are given. These terms are in particular "Alert", "Notification", "Action" and "Occurrence". As a reminder, an event is defined in this report as "anything that happens or is contemplated as happening at an instant or over an interval of time" (see chapter 6.3). The term "event" may also occur with the overloaded meaning of the term "event object" which is the representation of an event in a computer system and the actual meaning of the term is determined by the context.

6.4.1 Alert

Multiple (inconsistent) definitions of the term “event” exist (see chapter 6.2). The same is true for the term “alert”. We define an alert to be a notification that is of special interest for someone. From the perspective of a computer system there is no difference between an alert and a notification. The only difference is how a notification is interpreted and treated by the receiver. This depends upon the user receiving the notification. Sometimes, notifications of the same event type may even be an alert or not for the same receiver – it depends upon the context. An example for the former is a fire notification. This is an alert for firefighters but only a notification for a newspaper reader. An example

for the latter is a measurement notification which may be treated as an alert depending on the measured value. In the following only the more general term "notification" is used.

6.4.2 Notification

According to Mühl et al. (2006) a notification contains data describing an event. More precisely a notification is a container for event objects representing events. In this report a notification is defined more general and not restricted to contain events as the only payload. For instance, an event may describe a change of the state of an object. The according notification could contain a complete event object with all available information about the object or the change or just the message that something at the specific object has changed. The message in the latter case is not the event object representing the change. This event object would have to be pulled from a data source when receiving the message[7]. Note that a notification itself can be represented by / represents an event because it is something that happens.

6.4.3 Action vs. Occurrence

Actions and occurrences are both an event and can therefore be represented by event objects. The main difference is that the view on these events is further defined. In this sense an action is active and an occurrence is passive. Like an event can be an alert for some people, an action from one view may be an occurrence from another. It depends upon the perspective of the entity that detects the event. A bank robbery, for example, is an action from the perspective of the robber but an occurrence from the perspective of a bank employee.

6.5 Application Schema for Events

6.5.1 Introduction

Clause 6.3 defines the term *event* and discusses the specifics of an event, especially the relationship between an event and a feature. With respect to the OGC baseline, an event is a feature (see clause 6.3.1). If an event is a feature, then can we not design an application schema that could be used in various application domains? Ideally, this schema should be able to incorporate feature types from other application schema that represent event types. This chapter tries to define such a cross-domain application schema for events, called *event model*.

6.5.2 Event Model

The event model is organized in a package stereotyped <<ApplicationSchema>>. It has dependencies on a number of packages from the ISO 19100 Harmonized Model, as shown in Figure 1.

[7] This is an example for the common usage of the Observer pattern, see chapter 6.2.4.

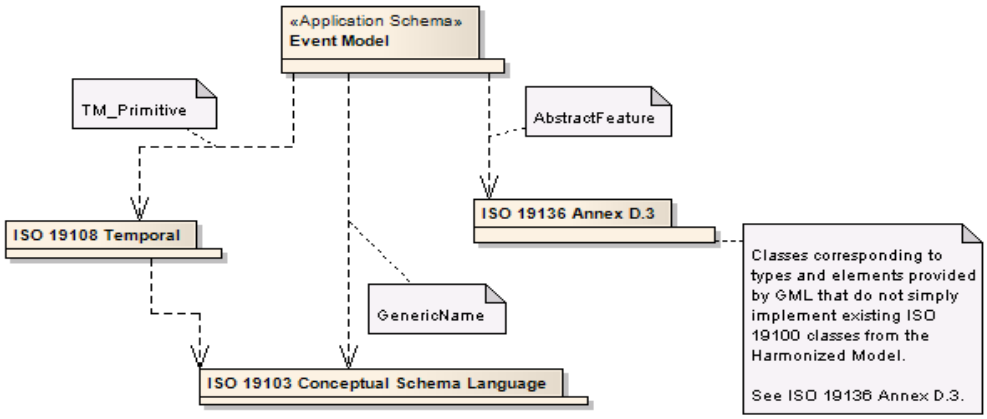

Figure 1 - Event model dependencies on packages from the ISO 19100 Harmonized model

Only one sub-package currently exists in the event model (see Figure 2).

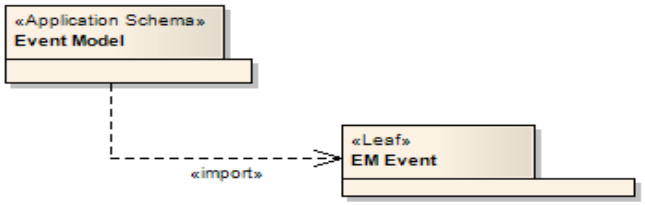

Figure 2 - Event model package structure

This package will be explained in the following.

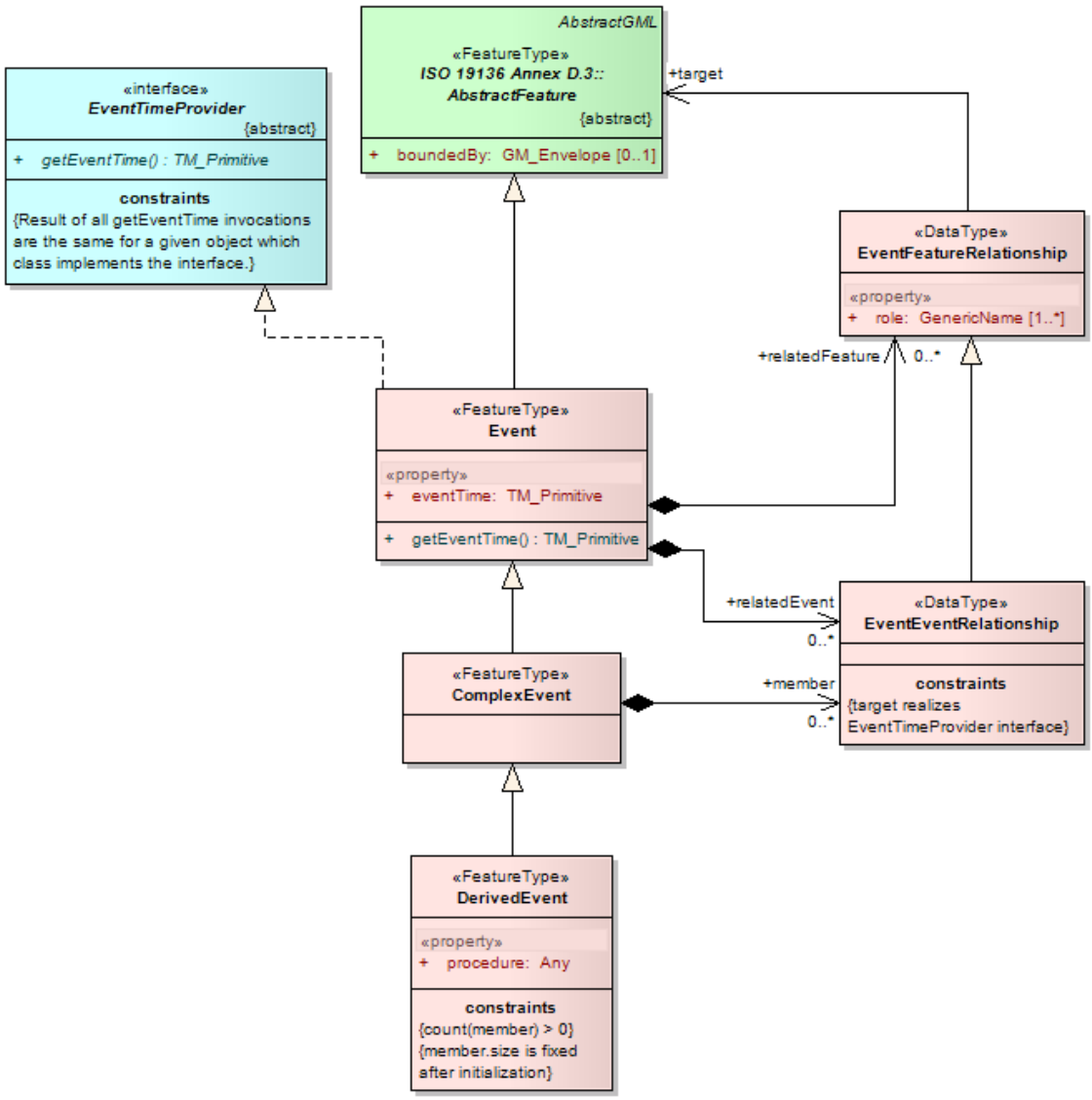

Figure 3 - The Event type and its specializations

NOTE: The class named "AbstractFeature" represents the set of all classes with stereotype <<FeatureType>>. In an implementation this abstract class will be substituted by a concrete class representing a feature type from a domain of discourse. This class is implemented in GML by the element gml:AbstractFeature.

An **Event** is a feature. It has a required property containing the time when the event happened. The **eventTime** is modeled as a TM_Primitive from ISO 19108. As such, the value may be a temporal geometry primitive (instant or interval) or a temporal topology primitive (node or edge). The Event class realizes the **getEventTime** operation from the **EventTimeProvider** interface which provides access to the value of the eventTime property.

NOTE: in ISO 19136, a feature is modeled as a gml:AbstractFeature which contains a boundedBy property that can only describe a spatial or spatio-temporal boundary, but not a pure temporal one. In the case of events, the temporal aspect is of primary interest. This also applies to non-geospatial features which contain complex temporal properties. Thus, the boundedBy property of gml:AbstractFeature should be modified to support both spatial, temporal and spatio-temporal bounding / extent information.

As described in clause 6.3.5, an event may be related to other features. Each **EventFeatureRelationship** is characterized by one to many **role** properties. This primarily supports use cases in which an event object is exchanged between multiple domains where the relationship target may incorporate a different association role and thus may have a different semantic as property of the event.

Example: an earthquake event may have a relationship to a street feature, indicating that the earthquake *affected* the street and that it actually *destroyed* the street. Similarly, a hurricane event may have a relationship to a butterfly, indicating that the hurricane was *causedBy* the butterfly and that the hurricane *blewAway* the butterfly.

An event may also be related to other events via an **EventEventRelationship**. This type inherits from EventFeatureRelationship and thus has AbstractFeature as target property. However, a constraint is added to the EventEventRelationship which requires the target to realize the EventTimeProvider interface. This allows us to establish relationships to features that can be considered as events, regardless of whether they derive from the Event type defined in this application schema or not.

Relationships between an Event and another feature / event are discussed in more detail in clause 6.3.7.

As discussed in clause 6.3.2, specializations of the **Event** type – which itself is neither an abstraction nor a composition of other events (it may nevertheless be related to other events) and thus represents a simple event – can be a **ComplexEvent** or a **DerivedEvent**.

An abstraction or aggregation of multiple events - its **members** - is called a ComplexEvent. The relationship to a member event is modeled through the EventEventRelationship. A ComplexEvent does not necessarily have member events. This situation will likely be the case when it is known that an event happened and that it was caused by other events but that these 'causing' events cannot be determined at the moment. The ComplexEvent will then be an abstraction of the happening caused by these unknown events.

This also implies that the event time of a ComplexEvent is - like for a simple Event - assigned by the entity that creates the event object. It can but does not have to be related to the event times of member events.

NOTE: Because the event time is a genuine property of an event, it may not be modified in an event object. This also applies to complex events. Therefore, if the event time of a ComplexEvent object was computed (at creation time) based upon - for example - the temporal bounding box of the member events that were available and if later the set of members is modified then the event time of the original event object may not be modified (compare 6.3.4.2.1). Rather, a new (complex) event object should be created which encompasses the modified set of member events and the new event time. This new event may for example have a relationship to the old event with a role of *supersedes* (see clause 6.3.7). Otherwise, if a modification of the member set has no implications upon the event time, then a new event object is not necessary.

If a well-defined procedure was used to detect an event based upon (the existence or absence of) one or more other events, the resulting event is called **DerivedEvent**. The procedure may be any process or algorithm that is capable of detecting an event based upon the (existence or absence of) information in member events. A DerivedEvent

therefore has at least one member. The event time of a derived event is determined by its procedure.

6.5.3 Details of Key Event Properties

6.5.3.1 Event Time – Property vs. Interface

In the event model, the base Event class and its specializations have an explicit attribute property that represents the time of the happening represented by the event. The value of this property is assigned when the event is created and is returned whenever the getEventTime operation is invoked. This operation is the means to conceptually get access to the event time of an event. With conceptually, we mean that it may very well be the case that an event type does not foresee an attribute property which represents the event time - instead the event time can be computed on-the-fly whenever the getEventTime operation is invoked.

We see that there is a difference between the conceptual model of an event type and its encoding. While the conceptual model may foresee the implementation of the EventTimeProvider interface, the encoding of that model according to the rules defined in ISO 19136 will simply omit the interface operation.

Can a feature type from a different application schema therefore be an event according to our definition (see clause 6.3)? The answer is yes. If a feature represents a happening (which is determined by the semantics of its feature type) and contains properties (attributes and associated types) which can be used to compute the time of that happening, then it is an event according to our definition. In that case, the feature type may be defined to implement the EventTimeProvider interface.

An O&M observation is a good example. It is a feature that represents an event (the observation represents the happening of sampling values for a property of a given feature of interest via a given procedure - see clause 6.3.7) the event time of which is given by the observation's sampling time.

Can an Event from the event model then use these feature types as related or member events? Again, the answer is yes. The constraint of the EventEventRelationship type explicitly says that any feature may be the target of such a relationship as long as it implements the EventTimeProvider interface.

Is it sufficient to be able to compute the event time for any given feature to be an event? Not exactly. Keep in mind that the event time is a genuine property of an event. It may not be modified during the event's lifetime. A feature which calculated the event time upon invocation of the getEventTime operation from the properties of associated types is not an event if these associations are modifiable - resulting in the event time to possibly differ per invocation of getEventTime.

6.5.3.2 Roles in Event Relationships

Events may have relationships to other features and thus also to other events. These relationships are characterized via role properties. Relationship roles are defined by

authorities and apply to a given domain. It depends upon the domain which roles need to be implemented. However, some roles serve a general purpose and are thus defined in this report. We can identify relationship roles for related events in general (see Table 2) and for events that are members of a complex event (see Table 3).

Table 2 - Roles implemented by a related event

Role Identifier	Meaning
supersedes	The target event is superseded by the source event. This means that the target event is deprecated.
revokes	The target feature is revoked which means that the target event object should be considered as not having been issued.
caused	The source event caused the instantiation of the target event. More specifically, the source event is (one of) the reason(s) why the target event was instantiated.

NOTE: In ISO 19136, a GenericName is implemented as a gml:codeType. A role property of an EventFeatureRelationship may thus have a codeSpace attribute in addition to the identifier value. The codeSpace defines the authority of the code value. For the codes listed in this clause the codeSpace value shall be the namespace of this application schema.

Superseding an event object with another is the preferred way to implement changes applied to an event object. Let us explain this in more detail. Any modification of an event object can always be critical for other applications. When transmitting an encoded representation of an event object to other systems, what they get is only a snapshot if the event object has modifiable / dynamic properties. Computations that are based upon this snapshot will need to be repeated when a change to an event object is made later on. In the worst case this would lead to a rollback of the whole computation, possibly involving a rollback on other systems as well. This situation is unavoidable and applies to all systems that base their computations upon given information. At least we can make it easier for event processing systems to detect a change of a previously received event object by implementing such a change in a new event object that has a relationship to the original event with the role *supersedes*.

If it is recognized that an event was wrongly detected, initialized and distributed, then this "happening of recognizing the failure" can be represented by a new event and an event object be distributed which relates to the wrong event, which shall be *revoked*.

How an application reacts when receiving an event that either supersedes or revokes is up to the application and not defined here.

Whenever an event is detected based upon the information contained in other events (a common use case in Event Processing - see clause 8.5), then there exists a causal relationship between the detected event and the base events. A base event *caused* the detected event (maybe multiple other events). From the perspective of the detected event, it was *causedBy* one or more other events. The set of member events that caused a ComplexEvent is sometimes referred to as the *causal vector* of that event.

Table 3 - Member Event - defined values of the role property in an EventEventRelationship

Role Identifier	Meaning
causedBy	The source event was causedBy the target event. More specifically, the target event is (one of) the reason(s) why the source event was instantiated.

6.5.4 Creating an Event Type Hierachy

Each of the three event types defined in this application schema can be used as is, if no additional properties need to be modeled. However, this is a special case. Usually you would derive your own type from one of the available event types. This is a robust approach to model an event hierarchy that is applicable in one or even several domains. Figure 4 illustrates how specific event types could be derived. Specific properties are omitted for brevity.

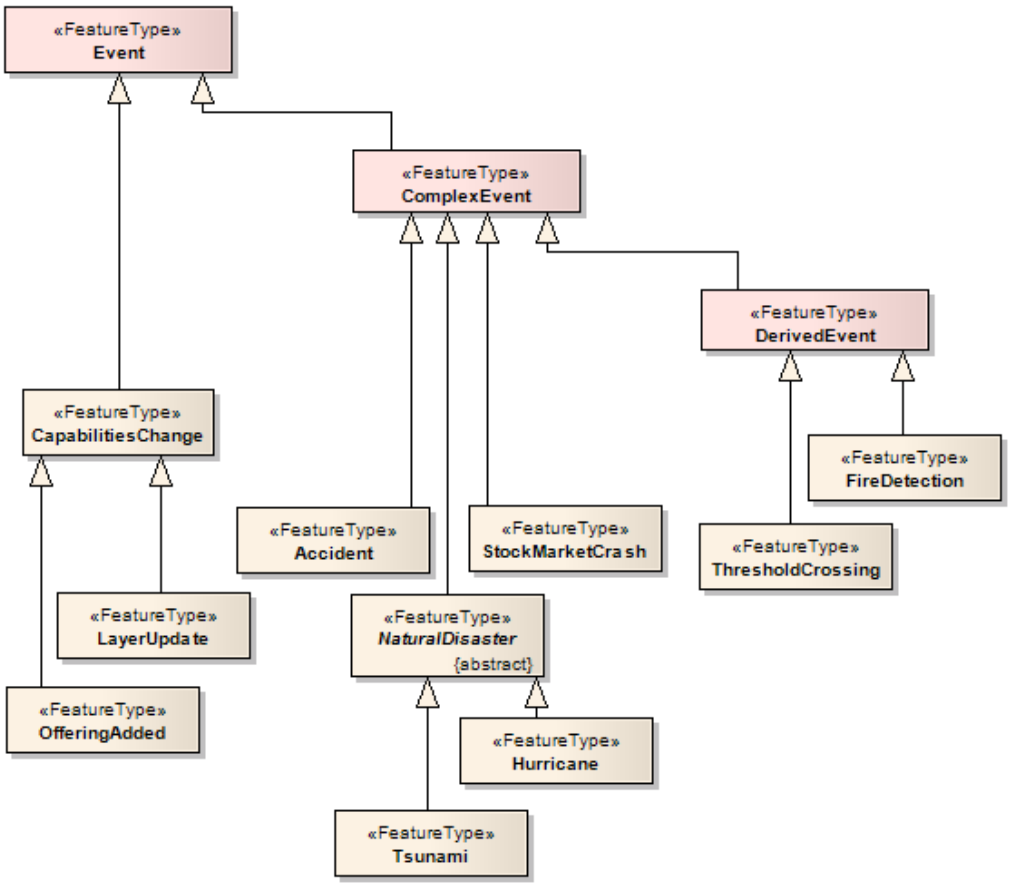

Figure 4 - Examples of specialized event types (non-normative)

Defining an OGC wide event type hierarchy is not in the scope of this report. Thus we will not go into further details at this point.

6.5.5 Event Taxonomy

A taxonomy is a classification scheme of things. It has a hierarchical structure with subtype-supertype or parent-child relationships. One example for a taxonomy is the inheritance structure of object oriented programs and APIs such as Java. Figure 5 shows a snippet of the Java object taxonomy. The arrows in the diagram describe a generalization or a subtype-supertype relationship.

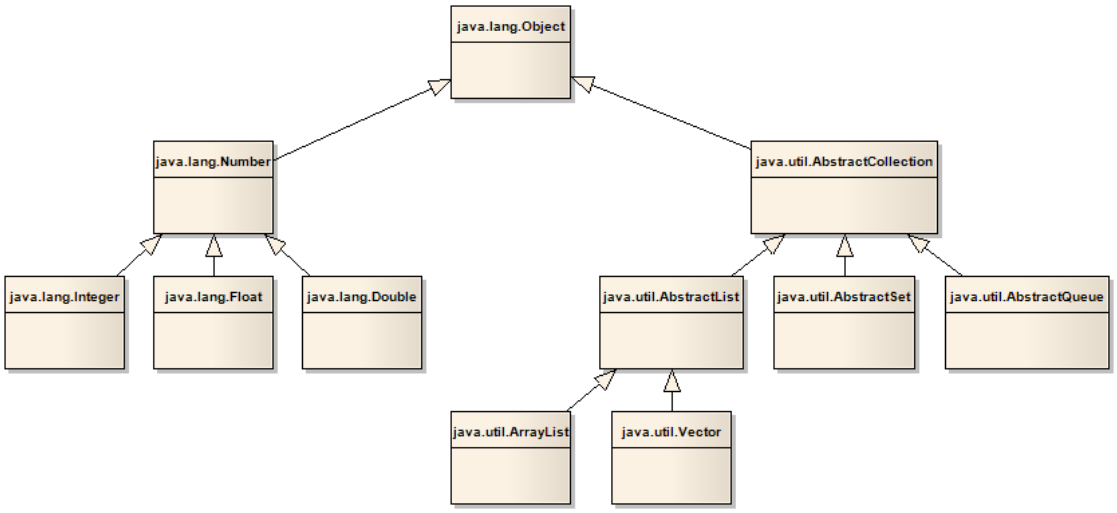

Figure 5 - Excerpt of the Java object taxonomy

In the context of taxonomies the term "ontology" also needs to be discussed. In computer science an ontology is a way of representing a set of concepts and relationships between them. Usually ontologies make use of taxonomies to represent inheritance relationships. In addition, other relationships can be used in an ontology that do not need to build a tree-style hierarchy. The graph of a relation may for instance be complete or have circles (e.g. "A knows B", "B knows C and "C knows A") which is not allowed in trees. Furthermore ontologies may contain attributes, functions and restrictions, amongst others. Like taxonomies they are usually designed for a specific knowledge domain. Further information about ontologies can be found in Gruber (2007).

In this context an *event taxonomy* is a taxonomy in the domain of an event architecture. It describes and classifies the different types and subtypes of events that are found in the event architecture. Therefore an event taxonomy may also be called an *event type taxonomy*. An event taxonomy that includes the definition of the properties of the event types is called an *event hierarchy*. An event hierarchy can be seen as an intermediate between an event taxonomy and an event ontology. An event hierarchy can be designed in UML, like shown in clause 6.5.4.

Note that an event taxonomy is not the same as a causality model as used in the Complex Event Processing (see 8.5). They both arrange different (types of) events in a tree-style

structure[8] but with different relations between these events: an event taxonomy uses "generalizes" and "specializes" whereas a causality model uses "is caused by". A causality model may be a part of an event ontology.

A common event taxonomy, hierarchy or even ontology is important and useful in various areas. First it defines names and terms for event types and their properties which are understood by multiple applications. It helps to develop and record a common understanding of the designed event types and thereby of the information flow represented by events in a system. It can also play a significant role when designing a causality model for a system using CEP because the event types from the taxonomy can - or better: should - be reused. Furthermore the definitions of event names and types improve the interoperability between different event driven systems.

The definition of an event taxonomy or hierarchy for the OGC is out of scope of this report. We can only give examples of some useful event types. The definition of a complete OGC event hierarchy would require involvement of all stakeholders with an interest in existing OGC Web Services (OWS). The resulting consensus process could fill a complete report and thus has to be done in the future. When this work is going to happen, one has to be aware of the fact that a complete event hierarchy cannot be defined; the reason being that every action in a system can be represented as an event. So it depends upon a given system which events can happen. We can solve this problem by just defining those event type hierarchies that apply to OWS in general. These hierarchies should be extensible and composable so that specific systems may combine their event types in one set which is discoverable. Clients can then make use of the events from a given system.

When defining a common event taxonomy or hierarchy, care should be taken upon how it is modeled. Have a look at Figure 6 and Figure 7. They illustrate different event taxonomies for a set of quite similar event types.

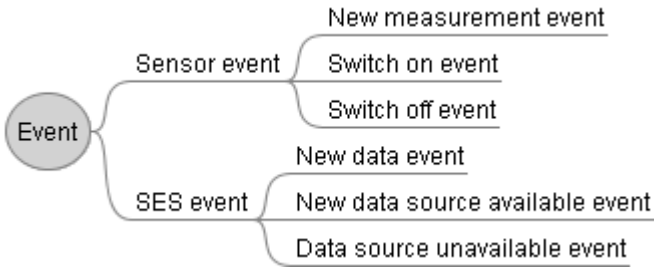

Figure 6 - Event taxonomy example 1

[8] The causality model is not necessarily a tree but rather a graph.

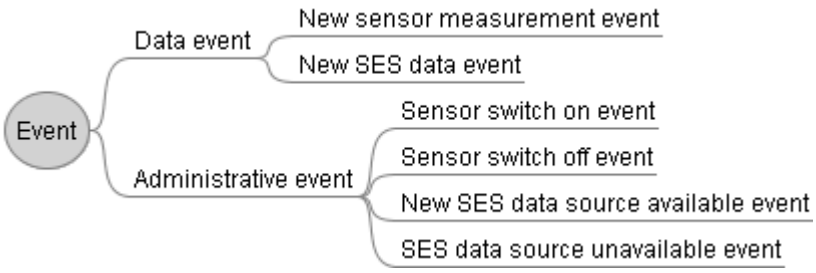

Figure 7 - Event taxonomy example 2 - a different approach with similar event types

Figure 6 shows a small event taxonomy from a service or component viewpoint where the event types are grouped on the first level by their origin. Figure 7 shows another event taxonomy with quite similar event types but from a content based point of view. Here the event types are grouped on the first level by their content (data or metadata). Note that the way in which the taxonomy or hierarchy is structured has an influence upon how clients may subscribe at a producer service (see chapter 8.1.3.1). Chapter 7 defines exemplary event taxonomies that are applicable in different OGC service scenarios.

6.6 Roles and Interfaces in the Event Architecture

6.6.1 Roles

In this chapter five different roles in an event architecture are introduced. This set of roles is defined on the basis of the roles used in the OASIS WS-Notification specifications. They give a coarse overview of the basic functionality in an event architecture and are used to identify which functionality is or shall be provided by which part of an architecture; when designing a new architecture as well as when working with an existing one. The roles do not depend on any algorithms performed by an implementing component but rather on the communication pattern being used. In the following seven roles are introduced. Have a look at Figure 8 which provides an overview of the different roles.

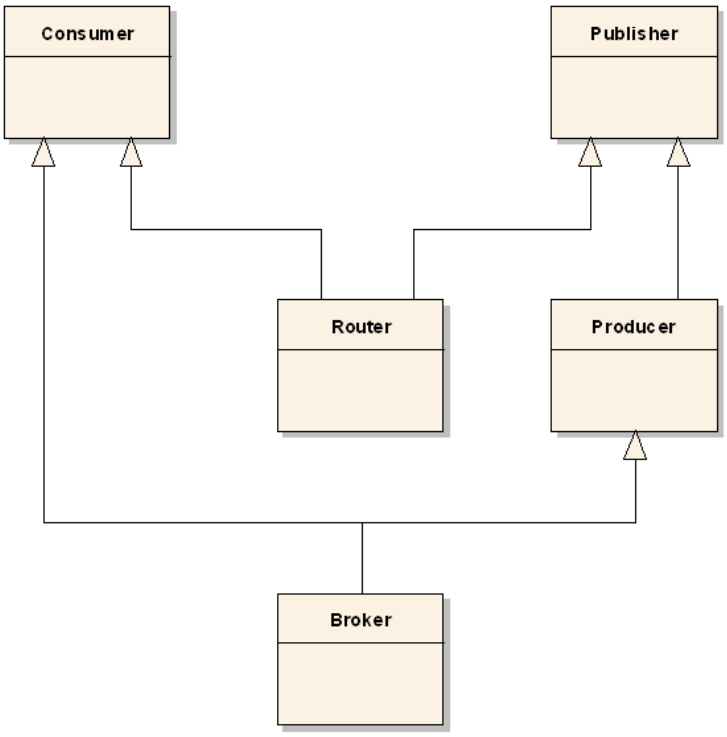

Figure 8 - Overview of the Roles in the Event Architecture

6.6.1.1 Consumer

Consumers receive notifications and therefore act as event sinks. They receive notifications from their counterparts, the Publishers (and thus also from Routers, Producers and Brokers). An example for a Consumer is a client that is receiving and displaying notifications from an alerting service. Consumers only receive. Consumers that support sending notifications are called Routers or Brokers (see below). A Consumer may also accept new Producers as event sources at runtime (see Registrar interface, chapter 6.6.2.2).

6.6.1.2 Publisher

Components that are event sources have the role of a Publisher. They are capable of sending notifications to Consumers. These Consumers have to be known or identified somehow, for instance by pre-configuration or implicitly. An example for a Publisher is a simple sensor sending notifications for each new measurement via its physical link to which a Consumer (e.g. a controller) is connected to.

6.6.1.3 Producer

A Producer is a specialization of a Publisher. Producers offer subscription interfaces in addition to Publisher capabilities which enables Consumers to subscribe at this kind of

Publisher. The Publish/Subscribe messaging pattern as well as subscription models is described in detail in chapter 8.1.3.

For example, an SPS could provide status notifications for the running tasks. In order to receive these notifications a Consumer would have to be subscribed at the SPS to receive the notifications of interest.

A Producer may still be capable of publishing notifications without prior subscriptions. This optional behavior can be advertised using policies for every Producer instance. Furthermore a Producer may offer methods to pause, resume and cancel subscriptions.

6.6.1.4 Router

Routers are derived from Publishers and Consumers. Routers receive events and forward them according to their configuration. They don't offer interfaces to subscribe to the notifications they send. Examples are components that receive notifications from Publishers and forward them to pre-configured Consumers. It is also possible that the desired Consumers can be derived from the notifications, for instance if they are listed explicitly (direct) or by using the spatial information about the notifications and possible Consumers (indirect). Routers are also often used as gateways to thin-wired sensor networks. Here, they build bridges between a network of often miniaturized sensors and the (fat wired) Internet. Another examples are relay services that distribute notifications from (registered) Publishers. Intelligent routing functionality is also used in Enterprise Service Busses (see chapter 8.4).

6.6.1.5 Broker

Brokers are derived from both Consumers and Publishers, i.e. they combine Consumer and Producer functionality. Brokers are capable of receiving and sending notifications. Consumers have to subscribe for notifications. Examples are news channels where everyone is allowed to post news (notifications) that are sent to subscribed Consumers, or an SAS where notifications from registered sensors are forwarded to Consumers with matching subscriptions.

Note that the combination of a Consumer and a Producer is only called Broker if one has access to the notifications received at the brokers Consumer-side via subscriptions. However, a broker does not need to grant access to every received notification. Some notifications might be invisible because of security constraints or they are just for service internal use.

6.6.2 Interfaces

In this chapter functionality of the roles described in the previous chapter are mapped to specific interfaces. These interfaces serve as a means to provide methods and method descriptions for the actions that are performed by the several roles. These interfaces are meant to be abstract and do not intend to define names and parameters for the given methods. For instance the name of the parameter "RegistrationParameters" is just a

placeholder for the information necessary for a registration (Registrar interface). Chapter 7 contains further examples for the described roles and interfaces.

The four interfaces shown in Figure 9 provide the methods that are used in the event architecture. Figure 10 illustrates which interfaces are implemented by which roles. Note that all roles capable of receiving notifications are also distinguished from each other by allowing the registration of new Publishers or not.

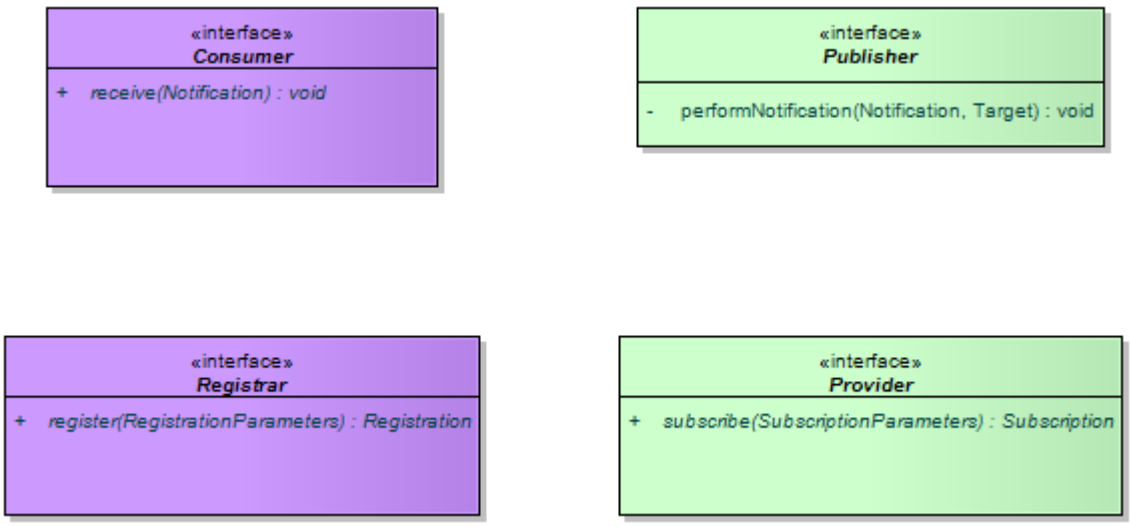

Figure 9 - Interfaces in the Event Architecture

6.6.2.1 Consumer

The interface for the Consumer role, a basic Consumer just has to implement this interface (see Figure 10). The method "receive" is invoked by a Publisher to send a notification to the Consumer. This interface is not using the registration offered by a Registrar (see 6.6.2.2 and 6.6.3.1).

The only parameter of the method "receive" is the notification that the Consumer shall receive. A response is not expected, therefore we have the datagram messaging pattern, see chapter 8.1.1.

6.6.2.2 Registrar

This interface is optional to implement by Consumers. It does not represent an autonomous role. The method "register" is invoked to register a Publisher at a Consumer. Whether the registration of a new publisher is required or not depends upon the Consumer. Policies can be used to indicate that registrations are required. They can also be used to define when a registration is required in greater detail. The register method cannot be used to subscribe a Consumer at a Producer.

The RegistrationParameters serve as a container for all the information a Consumer needs to know from a new Publisher. What kind of information is needed is application

specific. It may be a description of the publisher (e.g. a SensorML file as the description of a sensor system), a reference to the publisher or a specification if certain behavior like on demand publishing or pausability is supported.

The response payload may range from a simple acknowledgment message (e.g. HTTP 200 'OK') over the information needed to alter the registration (e.g. to renew or cancel it) to a full representation of the registration.

In an extension a Registrar could offer additional methods for the management of registrations. These methods are "cancelRegistration" to unregister a publisher and "updateRegistration" to change an existing registration for instance to set a new expiring time.

6.6.2.3 Publisher

The interface for the Publisher role; a basic Publisher only needs to implement this interface (see Figure 10). The method "performNotification" is invoked by the Publisher itself and therefore private. It is mentioned in order to amplify that the Publisher is the active part when sending a notification. When the method is invoked the given notification is sent to the designated target (a Consumer).

The parameters of the method "performNotification" are the notification that shall be sent and the endpoint (target) where this notification shall be sent to. The method has no response.

6.6.2.4 Provider

This interface also does not reflect an autonomous role in the event architecture. Like the Registrar provides an optional method for Consumers, the Provider provides an optional method for Publishers. The method "subscribe" is invoked in order to subscribe a Consumer at a Publisher for notifications of interest.

The SubscriptionParameters serve as a container for all the information a Publisher needs from a new Consumer. What kind of information is needed is application specific. As a minimum it must contain the reference to the target where notifications shall be sent to. Other possible information include topic restrictions, filter statements, policies and so forth.

The response payload may range from a simple acknowledgment message (e.g. HTTP 200 'OK') over the information needed to alter the subscription (e.g. to renew or cancel it) to a full representation of the subscription.

In an extension a Provider could offer additional methods for the management of subscriptions. These methods are "cancelSubscription" to unsubscribe a consumer and "updateSubscription" to change an existing subscription, for instance to set a new expiration time.

6.6.3 Combination of Roles and Interfaces

In this chapter the roles and interfaces introduced in the previous chapters are shown in combination (see Figure 10). Note that there are three additional roles which implement the Registrar interface. They are described in the following.

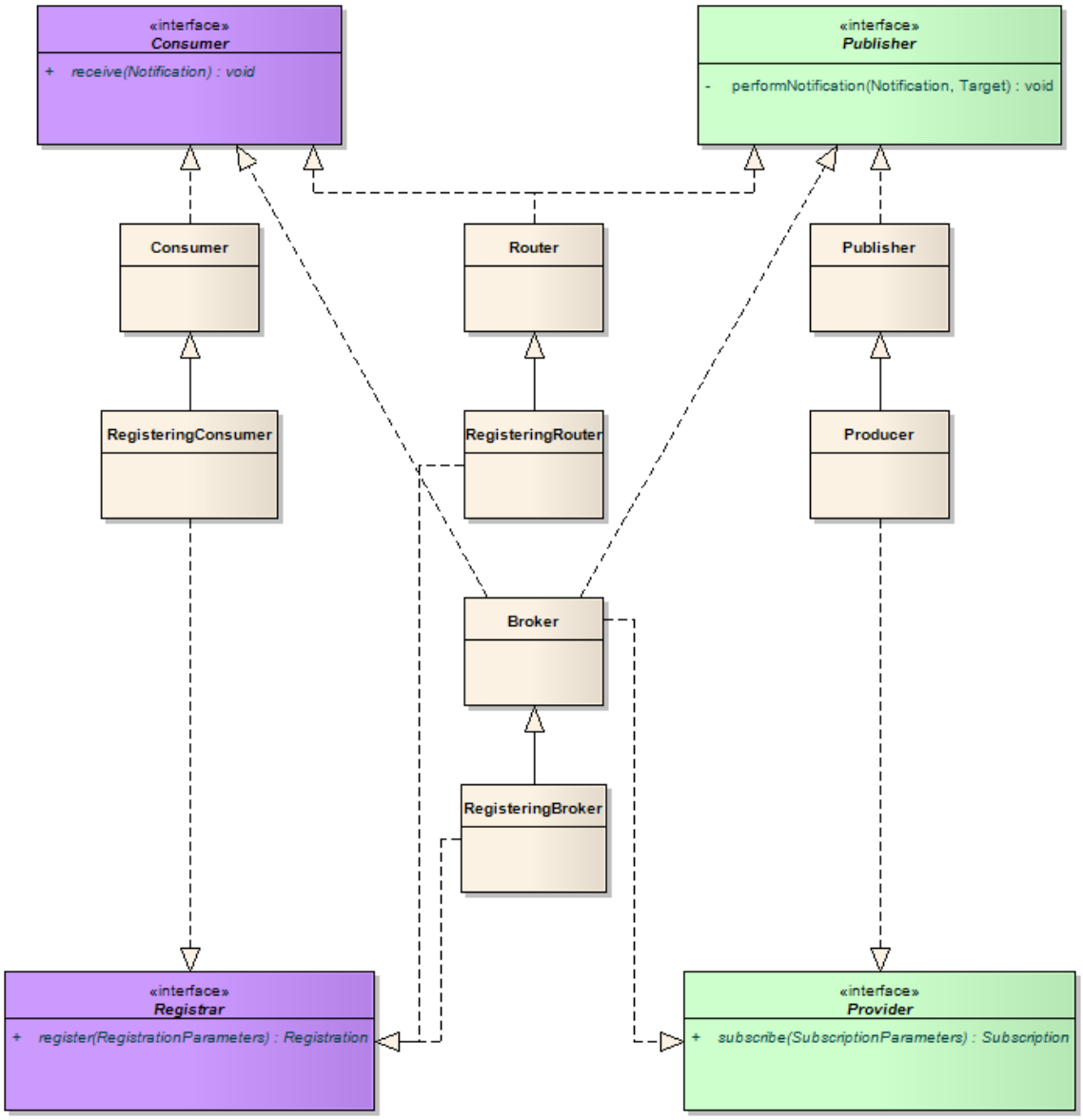

Figure 10 - Overview of the Roles and Interfaces in the Event Architecture

6.6.3.1 RegisteringConsumer

The RegisteringConsumer is an extension of the Consumer described above. It also implements the Registrar interface. It enables registration of Publishers at the Consumer. Again policies can be used to define whether a Publisher needs to be subscribed before publishing or not.

If a Producer shall be accepted as the notification source of a RegisteringConsumer and the connection between the two entities shall be made via an automatic subscription of the Consumer at the Producer, the Consumer has to be capable of additional features. First it has to accept registrations of Producers and not only Publishers. These registrations have to contain the information that an additional subscription at the Producer is necessary. In this case it is not necessary for the RegisteringConsumer to provide an endpoint for the notifications. An endpoint can also be communicated with the subscription at the Producer. This gives the RegisteringConsumer the opportunity to only subscribe for notifications that are necessary at a specific point in time and - if supported - pause or cancel them later on. This capability is further demonstrated in a scenario in chapter 7.4.

6.6.3.2 RegisteringRouter

The RegisteringRouter is an extension of the Router described above. It implements the Registrar interface. It enables registration of Publishers at the Router.

6.6.3.3 RegisteringBroker

The RegisteringBroker is an extension of the Broker described above. It implements the Registrar interface. It enables registration of Publishers at the Broker.

7 Mapping the Abstract Architecture to the OGC World

In this chapter we apply the abstract event architecture described in chapter 6 to various OGC services as well as clients and sensors. The mapping of the architecture to OGC services can be achieved in different ways, depending upon the given use case. In this chapter we provide an exemplary mapping scenario for each role and interface of the abstract architecture to components such as sensors, SOS and WNS but also common OGC services like WMS, and WFS.

Each scenario contains a short description of the scenario and a diagram showing the role (see Figure 10) that is instantiated by the central component of the scenario. Furthermore the sequence of interactions that constitute the scenario is shown by a sequence diagram and additional description. Finally, possible event types that may apply to each scenario are introduced. We do not intend to provide a complete event taxonomy or hierarchy. The event types in each scenario merely serve as examples of which events could occur in each scenario.

7.1 Event-Enabling a Client

7.1.1 Introduction

Clients are an essential part of a client-server architecture. Usually we have a user or application in mind when talking about clients, but of course services can also be clients to other services. When designing an event architecture it is important to have clients that fit into it, otherwise one would not be able to tap the full potential of the underlying architecture. The following scenario demonstrates how a client component fits into the event architecture.

7.1.2 Scenario

A client application shall receive notifications from one or more publishers. There are many ways in which publishers can be configured to send notifications to a specific client application, which will be exemplified in the scenario.

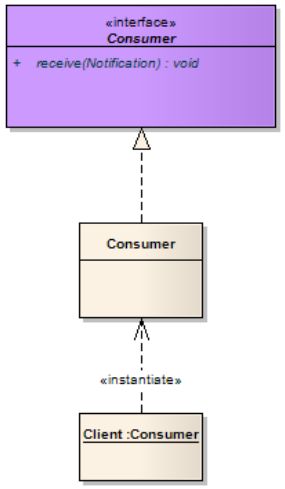

Figure 11 - Mapping the Consumer role to a client

In this scenario the client is an instantiation of the Consumer role (see clause 6.6.1.1).

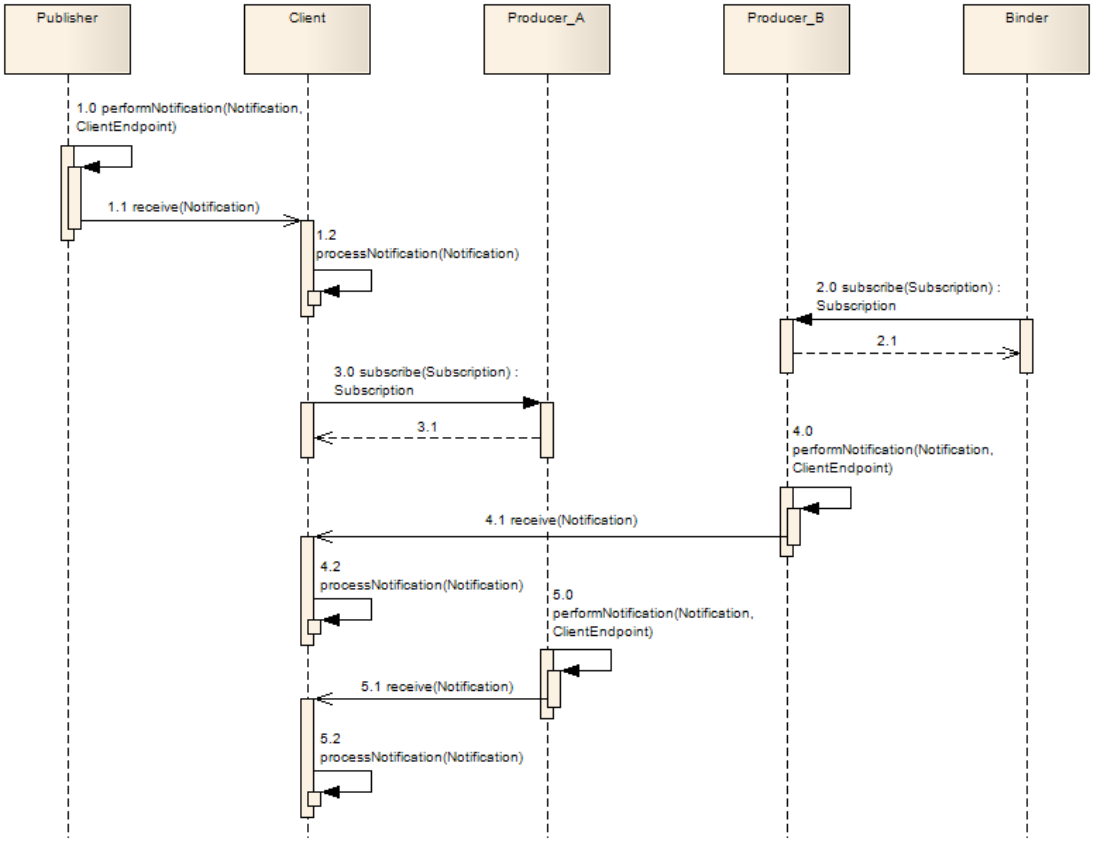

Figure 12 - Exemplary sequence of interactions between a client and several publishers

The sequence of interactions shown in Figure 12 will be explained in the following.

1.0: A Publisher wants to perform a notification (of the client). It therefore invokes the according method on itself with the actual notification and the client's endpoint as parameters. The knowledge of the endpoint has to be pre-configured in some way that is of no interest here.

1.1: The Publisher invokes the "receive" method - inherited from the Consumer role - at the client to deliver the notification.

1.2: The client processes (e.g. renders) the notification.

2.0: In order to receive notifications from Producer B[9] the client first needs to be subscribed at this Producer. Here, another component called Binder invokes the "Subscribe" method at the Producer. The subscription parameters contain the client's endpoint and optional information to limit the notifications of interest to a specific subset for instance by using a filter statement.

2.1: The Producer responds to the subscribe request with a confirmation including (a reference to) the new subscription.

3.0 and 3.1: Instead of using a binder the client may also subscribe itself at a Producer (called A in this example). This is done in the same way as when using a Binder.

4.0 to 4.2 and 5.0 to 5.2: The delivery of notifications from a Producer is performed in the same way like shown in 1.0 to 1.2. Note that only notifications that satisfy the subscription criteria are sent to the client. The processing steps of matching data against the subscription criteria of the client are not shown in the diagram.

7.1.3 Event types

The events that can be received by the client are defined by the application domain and of course the publisher (thus Figure 13 shows a generic Publisher_Event and Producer_A_Event). This can be data events like sensor measurements or derived events based on measurements or even administrative events like "new sensor available" or "new service available". Another type of administrative events that may occur in this scenario are subscription events. These events may be of interest for an administrator.

Figure 13 - Exemplary event taxonomy for the client scenario

[9] Note: A Producer is a specialization of a Publisher (see 6.6).

7.2 Event-Enabling a Sensor

7.2.1 Introduction

In addition to client components, data sources form an essential part of a distributed system. Usually they are encapsulated or hidden behind the services inside a system. However, with the introduction of transactional web services, these data sources but also other services now become top-level system components. The following scenario shows how a sensor, being a data source of the Sensor Web, could be integrated into the event architecture.

7.2.2 Scenario

A sensor shall provide measurements and status messages via notifications to a sensor controller implementing a Consumer (e.g. a more complex sensor system connected to the internet, or an SOS - see clause 7.3).

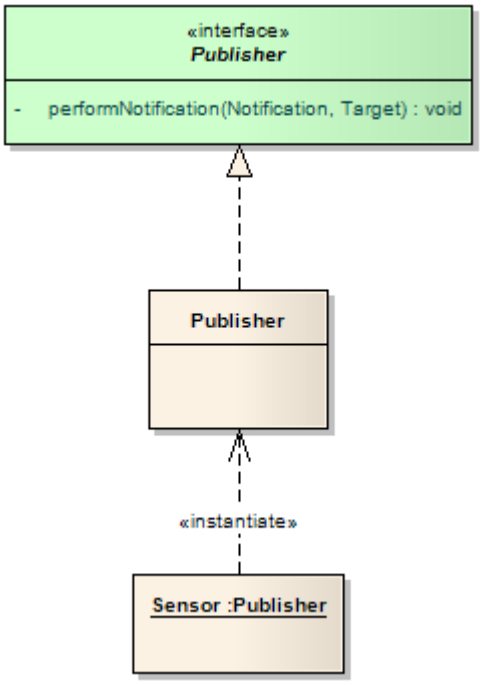

Figure 14 - Mapping the Publisher role to a sensor

In this scenario the sensor is an instantiation of the Publisher (see clause 6.6.1.2).

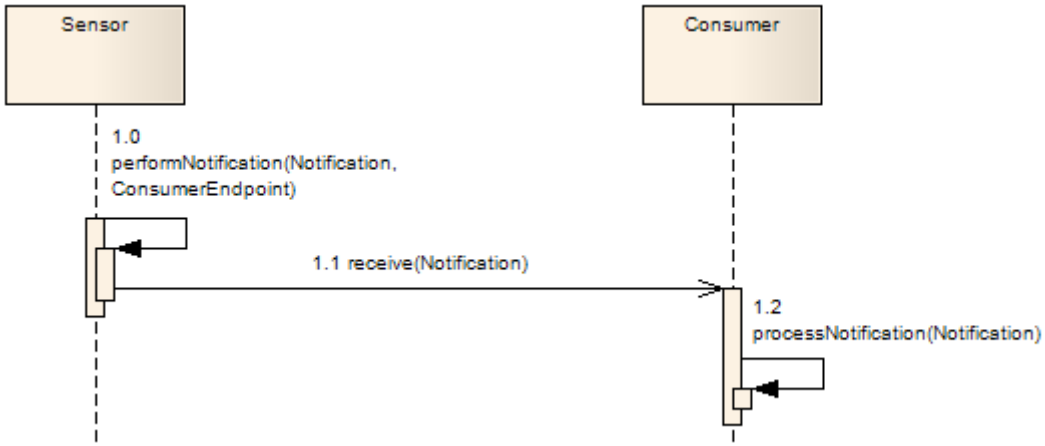

Figure 15 - Exemplary sequence of interactions between a sensor and a consumer

The sequence of interactions shown in Figure 15 will be explained in the following.

1.0: The sensor wants to perform a notification (of the consumer). It therefore invokes the according method on itself with the actual notification and the client's endpoint as parameters. The knowledge of the endpoint has to be pre-configured in some way that is of no interest here.

1.1: The sensor invokes the "receive" method at the Consumer to deliver the notification.

1.2: The Consumer processes the notification (e.g. a SOS might store contained measurement).

7.2.3 Event types

The events that a sensor sends via notifications depend upon the capabilities of the sensor itself (for example, simple sensors might not support status messages) and the application domain. The possible event types in Figure 16 contain only one event to publish measured data (Data event). This could be different if the sensor in this scenario would be replaced by a complex sensor system where multiple other sensors are mounted upon and where each of these sensors sends specialized types of the generic data event. The status events (a specialized type of administrative events) are examples of the many event types that could possibly represent happenings related to status changes of a sensor.

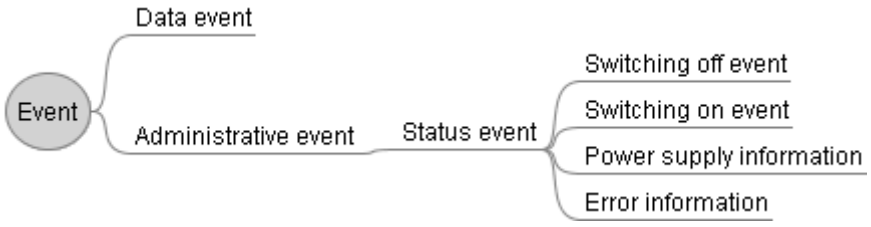

Figure 16 - Exemplary event taxonomy for the sensor scenario

7.3 Event-Enabling a Sensor Observation Service

7.3.1 Introduction

The Sensor Observation Service (SOS, OGC 06-009r6) is defined to provide pull based access to observations. For the integration of a SOS in the event architecture one could redesign it to allow a push based access, for instance by allowing GetObservation requests with a sampling time filter that contains a time interval that ends in the future. Push behavior is currently more generally solved by the Sensor Event Service (SES, OGC 08-133) and Sensor Alert Service (SAS, 06-028r5). Note that all of these specifications are intended to define service interfaces. This means that one web service may implement a SES interface in addition to a SOS interface for providing push and pull based access to its sensor data.

However the same does not apply on the consumer side of the SOS where new observations are inserted. In its current specification state, the SOS uses a custom request / response model which is not part of the event architecture. The following scenarios describe how a SOS can be integrated into the event architecture. Scenario 1 discusses the Consumer side of the SOS, scenario 2 discusses a possible combination with the SES interface.

7.3.2 Scenarios

7.3.2.1 Scenario 1

A SOS shall archive data that is generated by some other component in the event architecture. Data shall be pushed to the SOS, but only by a previously registered component.

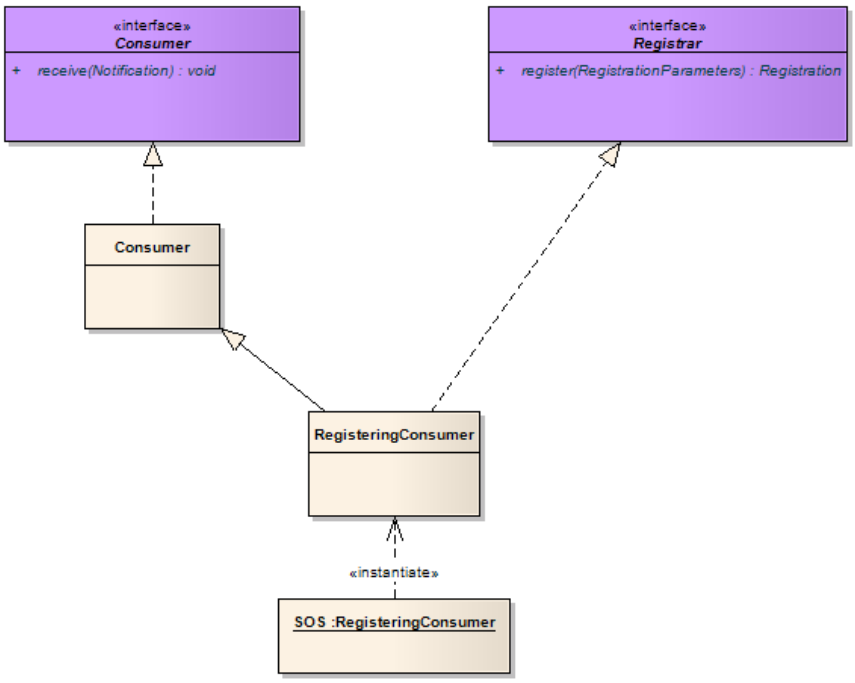

Figure 17 - Mapping the RegisteringConsumer role to a SOS

In this scenario the SOS is an instantiation of the RegisteringConsumer (see clause 6.6.3.1).

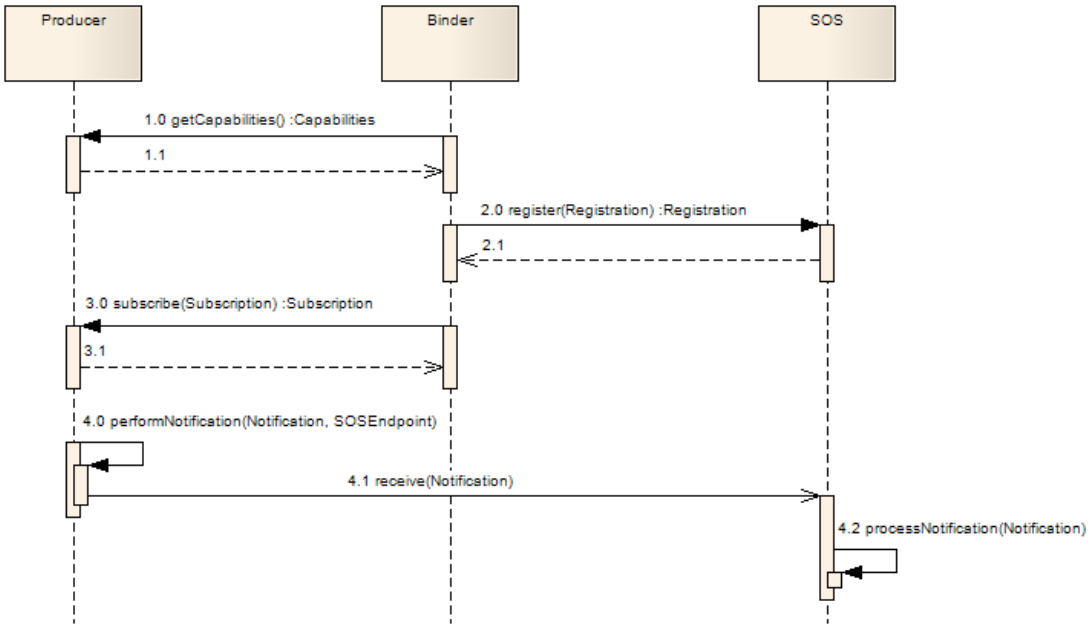

Figure 18 - Exemplary sequence of interactions between a SOS, binder and producer

The sequence of interactions shown in Figure 18 will be explained in the following.

1.0 and 1.1: A binder application retrieves the capabilities of a Producer.

2.0: The binder registers the Producer at the SOS. The registration parameters include the information required by the SOS, like the descriptions of the sensors from which it will receive new measurements. This information is retrieved by the binder from the producer's capabilities.

2.1: The SOS responds to the register request with a representation of the registration (or a reference to it). This registration contains the endpoint at the SOS for this registration.

3.0: The binder application invokes the "subscribe" method at the Producer. The communication endpoint of the SOS for the previous registration is included in the parameters.

3.1: The Producer responds to the subscribe request with a confirmation including (a reference to) the new subscription.

4.0: The Producer starts to perform a notification of the SOS (adds new data to the SOS). It therefore invokes its "performNotification" method with the actual notification and the SOS registration's endpoint as parameters.

4.1: The Producer invokes the "receive" method at the SOS to deliver the notification.

4.2: The SOS processes the notification (i.e. inserts the new data in the database).

If a Publisher shall be used instead of a Producer the steps 1.0, 1.1, 3.0 and 3.1 are not applicable. In this case the capabilities of the Publisher have to be known at the binder application to register it at the SOS. Also the SOS registration's endpoint has to be known at the Publisher in advance and therefore either has to be pre-configured or be made known by some other means, e.g. through a custom operation.

7.3.2.2 Scenario 2

A SES that instantiates the Producer role shall be an event source of a Consumer. The SES interface could in fact be realized by a web service in addition to the SOS interface. This would result in a push based access to the data of a SOS.

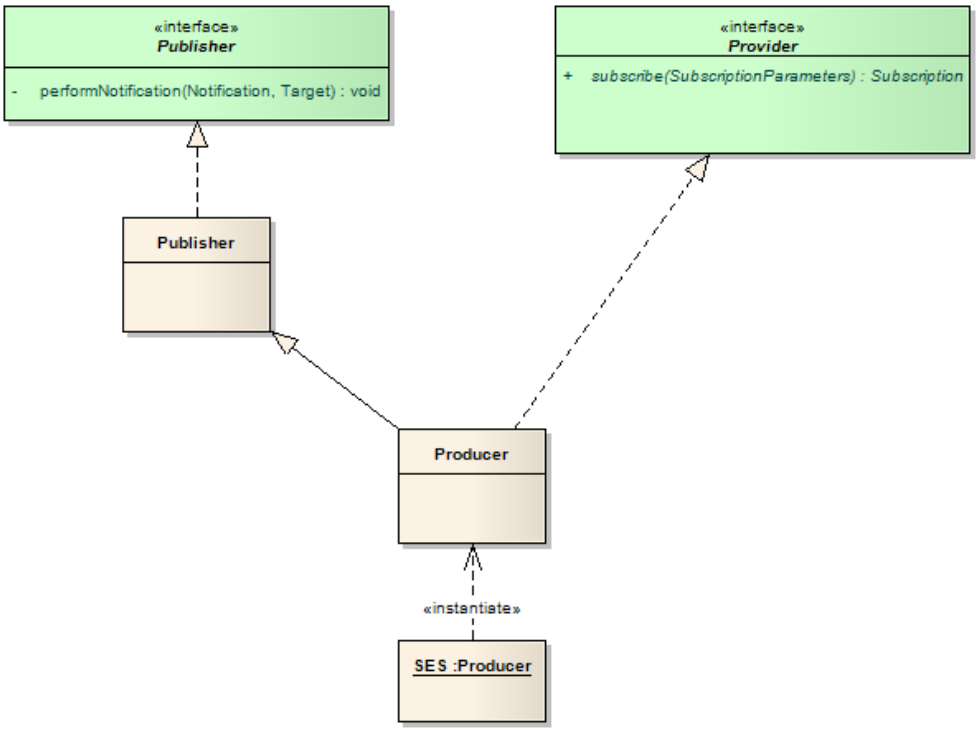

Figure 19 - Mapping the Producer role to a SES

In this scenario the SES is an instantiation of the Producer role (see clause 6.6.1.3).

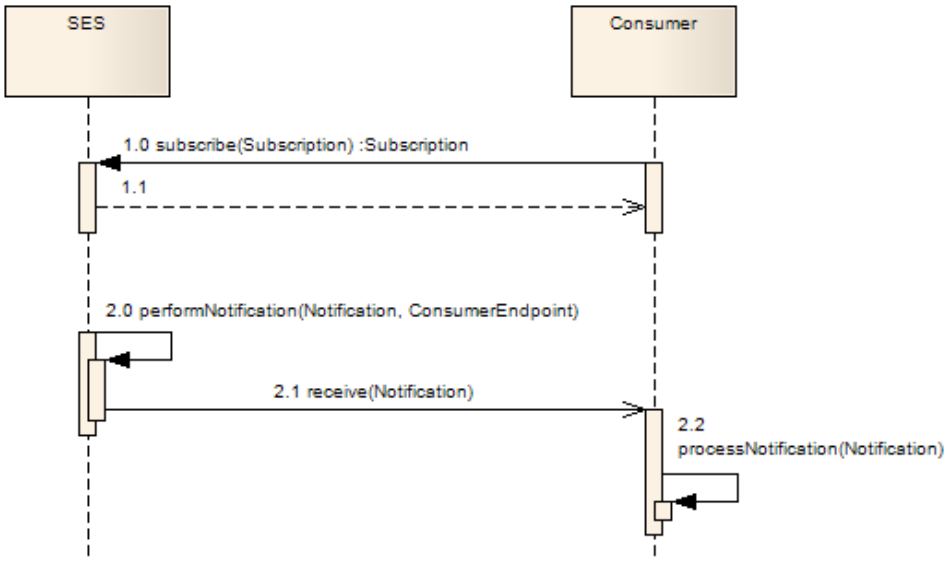

Figure 20 - Exemplary sequence of interactions between a SES and consumer

The sequence of interactions shown in Figure 20 will be explained in the following.

1.0: The Consumer invokes the "subscribe" method at the SES. It provides its endpoint in the subscription parameters.

1.1: The SES responds to the subscribe request with a confirmation including (a reference to) the new subscription.

2.0: If an event matches the subscription criteria the SES invokes its "performNotification" method with a notification containing the event and the consumer's endpoint as parameters.

2.1: The SES invokes the "receive" method at the Consumer to deliver the notification.

2.2: The Consumer processes the notification.

7.3.2.3 Conclusion

A web service can implement multiple interfaces, for instance the extended SOS and the SES interfaces as described in scenario 1 and 2. It would then be capable of receiving notifications from publisher components contained in an event driven system and also to publish events to subscribers. This combination would then be an implementation of a broker with the additional capability to provide pull based access to archived data.

7.3.3 Event types

As in many services one can separate the occurring events in two major types: the data event and the administrative events. For the service used in the two scenarios there are two types of administrative events. The registration events describe changes regarding registrations like creation, canceling, expiring and renewing of registrations. The subscription events describe these changes for subscriptions.

On the other hand the service has to deal with data events. These event types can be defined via a registration (i.e. a Publisher registers its data event at the service) or via a subscription (i.e. a Consumer subscribes for events of a special type). The events of the latter type may be a subset of the former type (e.g. a filtered subset of events) but do not have to (e.g. events created via CEP, see clause 8.5). In scenario 1 only events defined by registrations are used while in scenario 2 both types may appear.

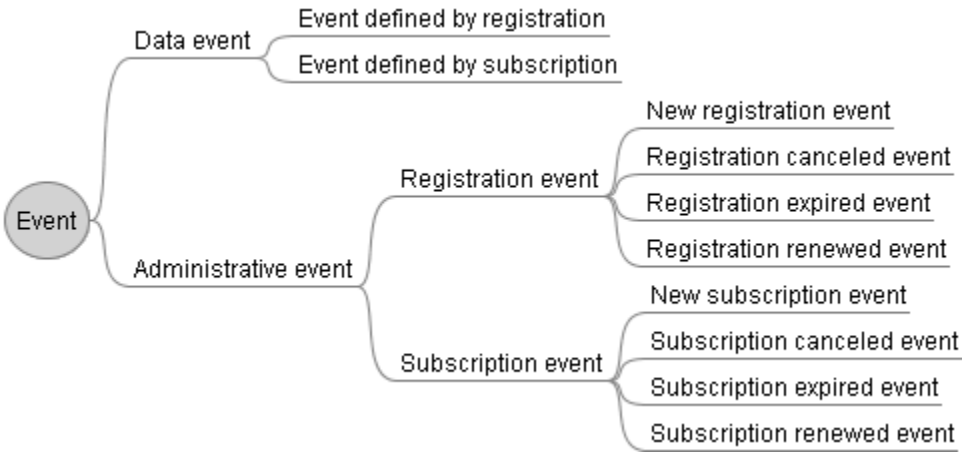

Figure 21 - Exemplary event taxonomy for the SOS scenario

7.4 Using Gridded Data in an Event-Enabled Environment

7.4.1 Introduction

In this paragraph we discuss a scenario where gridded data is used. With gridded data we mean data that is provided in n dimensional arrays, for instance satellite images or scan lines. Clients are interested to access or receive only subparts of the whole data grid. The examples in this paragraph focus on two dimensional arrays.

7.4.2 Scenario

A sensor provides its measurements as gridded data. This data shall be brokered and distributed by a SES. Because of bandwidth limitations only a requested subset of the overall data contained in one grid published by the sensor, i.e. specific array entries, shall be published by the sensor. This could be achieved by combining the SES and sensor in one entity or by registering every single array entry at the SES and make these registrations pausable. The former does not solve the problem with bandwidth limitations because of possibly many connections by clients to this SES-sensor combination. The latter becomes very confusing at least for clients because the SES would offer 100 different events for a 10x10 sized array.

Another solution is that the sensor provides a subscription interface for the SES, so that the SES can subscribe for the array entries that are needed to solve the queries requested by clients. The SES has to handle its subscriptions at the sensor in an intelligent manner to avoid multiple subscriptions for one entry but also to group different necessary entries in one subscription, for instance by subscribing for a complete column of a two dimensional array.

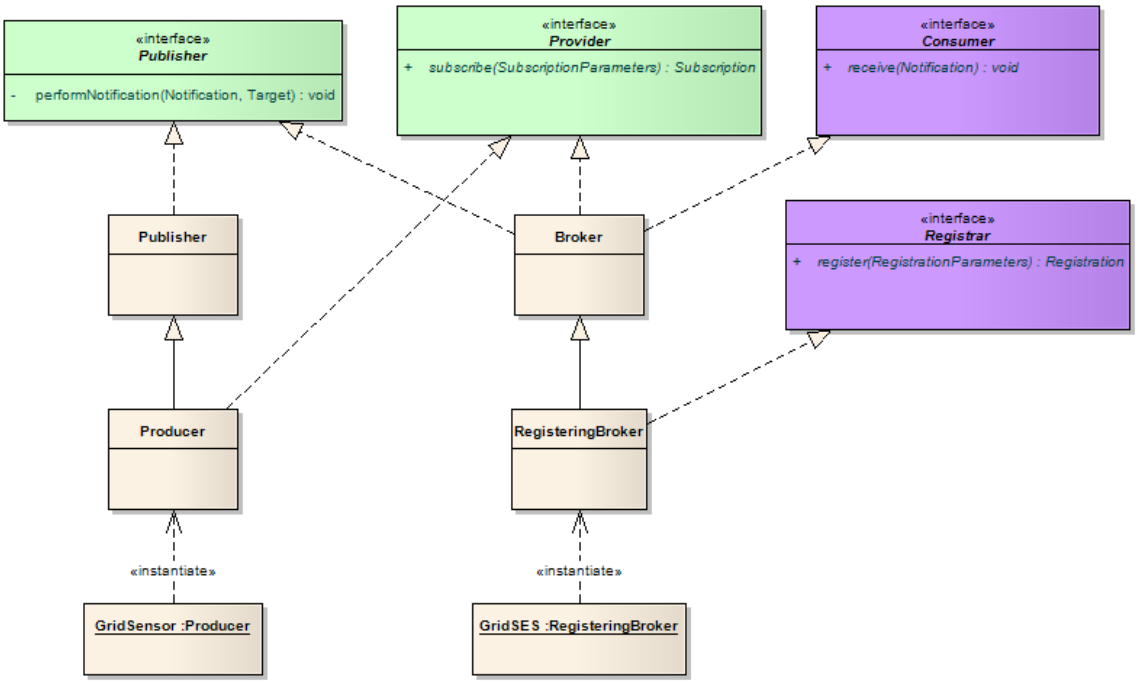

Figure 22 - Mapping the Producer and RegisteringBroker roles to components providing gridded data

In this scenario the sensor is an instantiation of the Producer role (see clause 6.6.1.3) while the SES is an instantiation of the RegisteringBroker role (see clause 6.6.3.3).

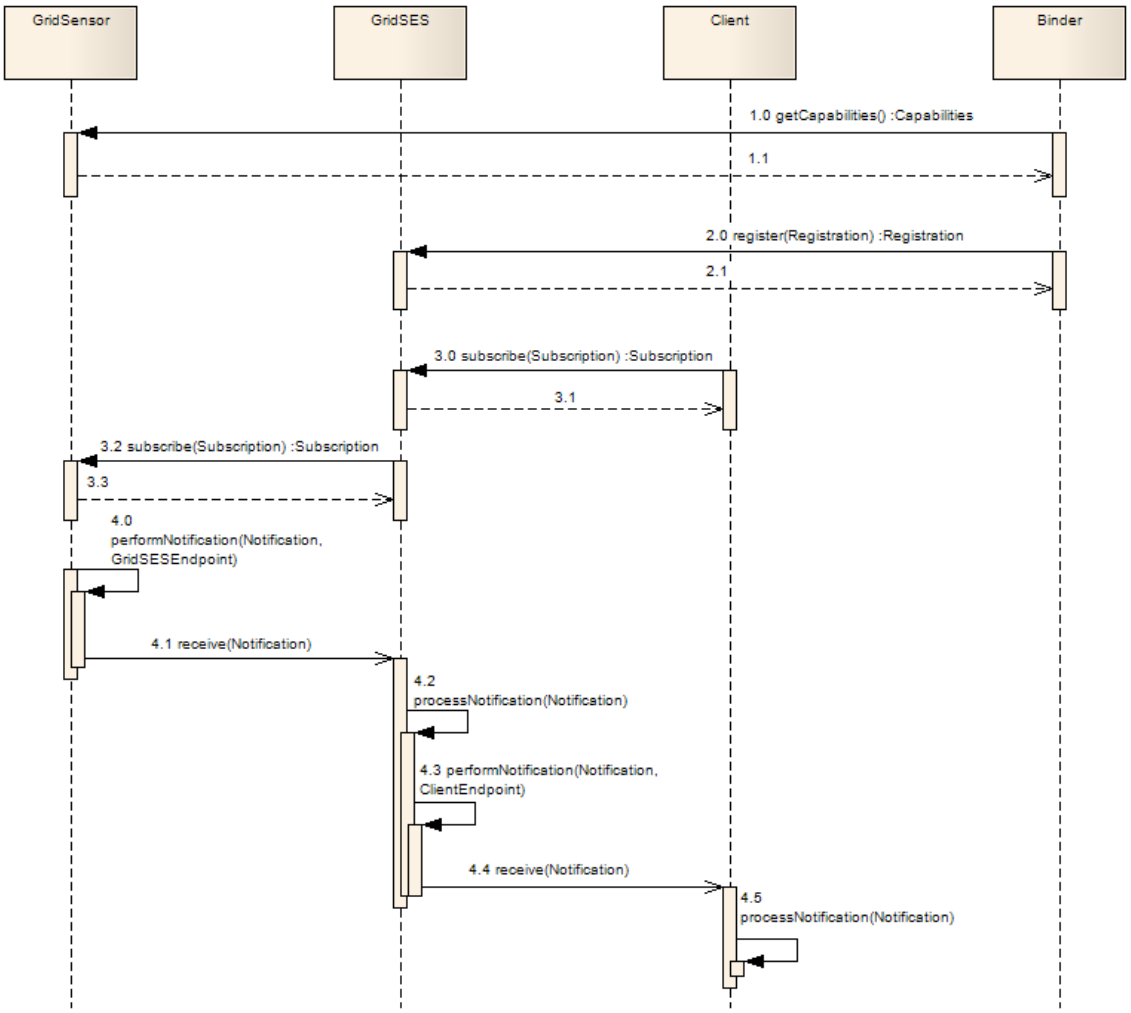

Figure 23 - Exemplary sequence of interactions between a SES and sensor providing gridded data, a client and a binder

The sequence of interactions shown in Figure 23 will be explained in the following.

1.0 and 1.1: A binder application retrieves the capabilities of the GridSensor.

2.0: The binder registers the GridSensor at the GridSES. The registration parameters include the GridSensor's capabilities. This is necessary for the GridSES to be able to complete its own capabilities, which requires all publishing sensors to be listed

2.1: The GridSES responds to the register request with a representation of the registration (or a reference to it). Communication endpoints for data transmission are exchanged in step 3.2.

3.0: A client subscribes for notifications offered by the GridSES based on the data offered by the GridSensor. The subscription parameters contain a filter reducing the necessary data to a subset of the whole available grid.

3.1: The GridSES responds to the subscribe request with a confirmation including (a reference to) the new subscription.

3.2: The GridSES subscribes for notifications at the GridSensor. The subscription parameters contain a filter describing which subset of the offered grid shall be published. The intelligent subscription behavior as described in the introduction has to be applied here. This may result in additional communication for instance to cancel or pause unnecessary subscriptions.

3.3: The GridSensor responds to the subscribe request with a confirmation including (a reference to) the new subscription.

4.0: If an update occurs at an array entry the GridSES subscribed for, the GridSensor invokes its "performNotification" method with a notification containing the required part of the grid and the GridSES's endpoint as parameters.

4.1: The GridSensor invokes the "receive" method at the GridSES to deliver the notification.

4.2: The GridSES processes the notification.

4.3: The GridSES invokes its "performNotification" method with a notification containing the process results based on the GridSensor's data and the client's endpoint as parameters.

4.4: The GridSES invokes the client's "receive" method to deliver the notification.

4.5: The client processes the received notification.

7.4.3 Event types

Again there are two major types of events involved in this scenario: data events and administrative events. The administrative events are the same as in the previous chapter but the importance of the subscription events is a bit higher here because subscriptions to the GridSES have more complex consequences (see steps 3.x). The data events can be separated into grid events and processed events. The former is used by the GridSensor to publish updates of the data grid. They can contain data of a single field or of multiple fields at once. The processed events are used as the output of the GridSES. They contain data derived from the grid events. If a client subscribes for unprocessed grid data the GridSES can of course forward the requested data as grid events.

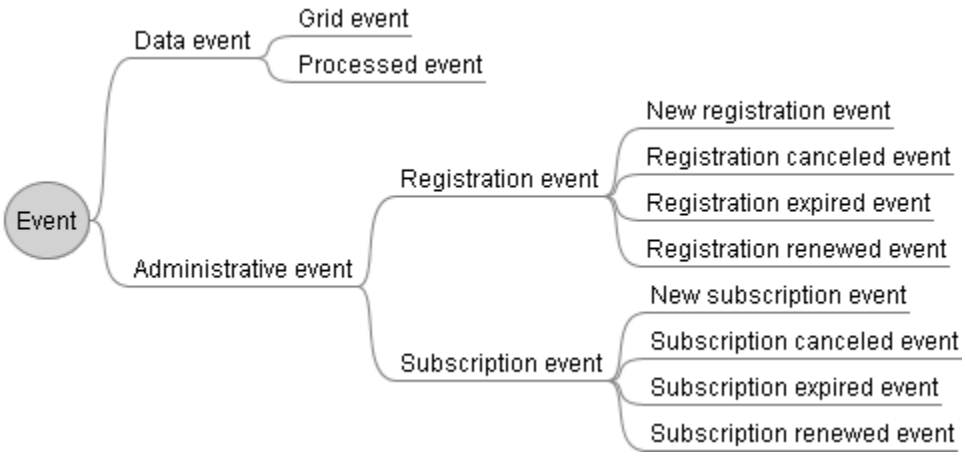

Figure 24 - Exemplary event taxonomy for the gridded data scenario

7.5 Event-Enabling a Sensor Planning Service

7.5.1 Introduction

The Sensor Planning Service (SPS) is used to task sensors. This may range from changing the sampling rate of a simple detector to the complete control of earth observation satellites. The SPS is currently available in version 1.0 but version 2.0 is in development. It is intended that this new version already supports publish / subscribe mechanisms for status notifications.

7.5.2 Scenario

In this scenario a client subscribes for status update notifications at an SPS. This client could for instance be embedded in a satellite control center monitoring the satellites. The actual SPS task may be submitted by other components or users.

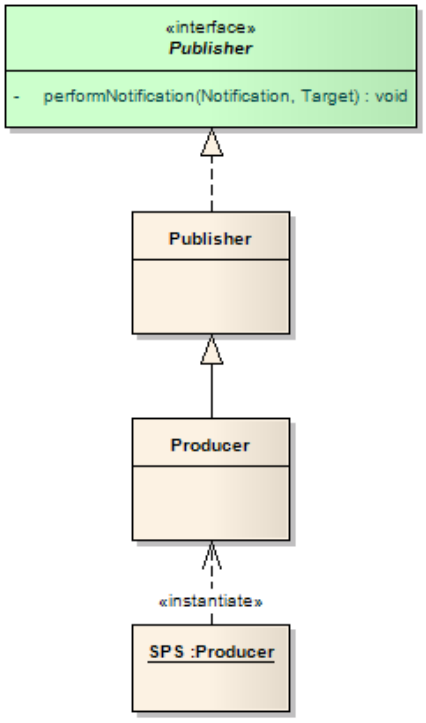

Figure 25 - Mapping the Producer role to an SPS

In this scenario the SPS is an instantiation of the Producer role (see clause 6.6.1.3).

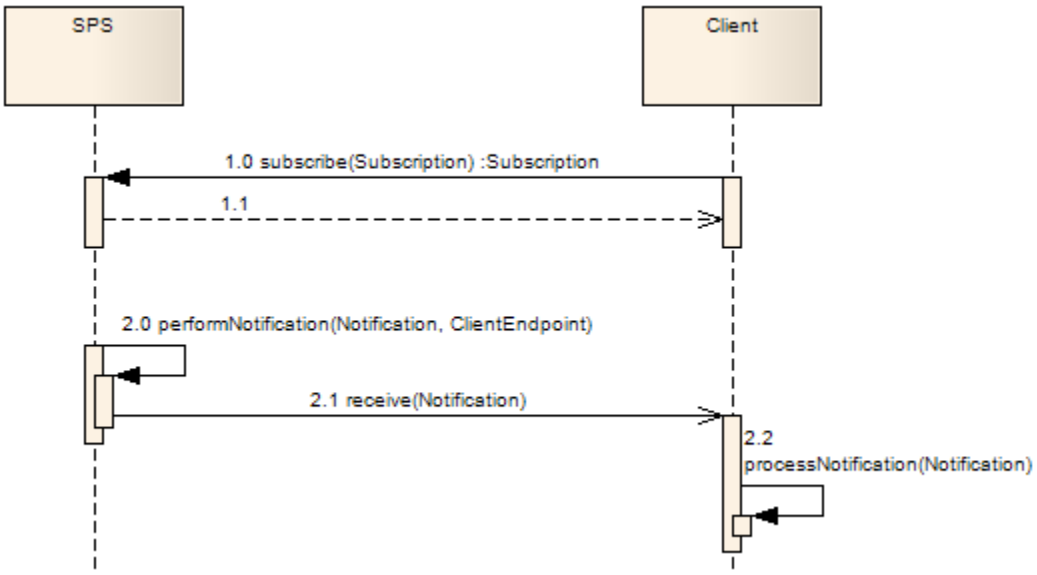

Figure 26 - Exemplary sequence of interactions between an SPS and a Client

The sequence of interactions shown in Figure 26 will be explained in the following.

1.0: The Client subscribes at the SPS for status updates. It provides its endpoint for notifications in the subscription parameters.

1.1: The SPS responds to the subscribe request with a confirmation including (a reference to) the new subscription.

2.0: When a status update occurs the SPS starts to perform a notification of the Client. It therefore invokes its "performNotification" method with the actual notification and the client's endpoint as parameters.

2.1: The SPS invokes the "receive" method at the client to deliver the notification.

2.2: The client processes the notification.

7.5.3 Event types

The events used in this scenario can be divided into three groups. The service metadata changed events are used to communicate changes of the service's capabilities, in this example changes to offerings. Sensor metadata changed events contain information regarding sensors like recalibration or movement. The task status update events contain information regarding a single task like reserving, accepting and canceling. Also, if new data generated by this task is available a task status update event is sent.

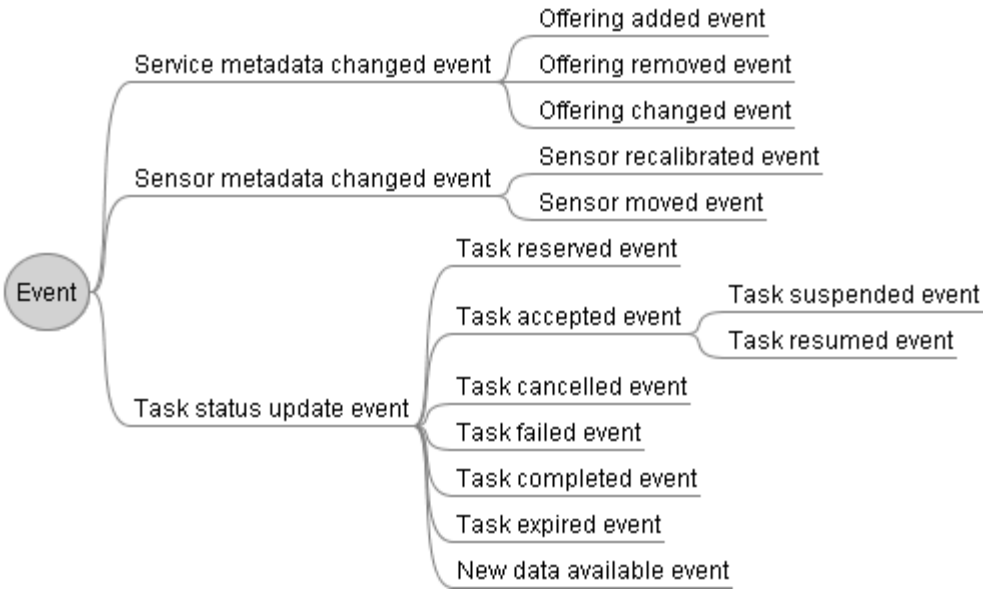

Figure 27 - Exemplary event taxonomy for the SPS scenario

The events as shown in Figure 27 are also used as an example in Annex A.

7.6 Event-Enabling a Web Notification Service

7.6.1 Introduction

The Web Notification Service (WNS) is part of the SWE services suite. It is used as a notification service for a "last mile" notification of users. It is for instance used in the SPS 1.0 specification to notify a user about completed tasks. More generally the WNS could (in a next version) be seen as a protocol transformator receiving notifications in one protocol (e.g. O&M) and publishing them in another protocol (e.g. a textual description via email). This service is already based on a push based communication pattern. The following scenario shows how the WNS fits into the event architecture.

7.6.2 Scenario

A WNS shall be used to receive notifications from a Producer and forward them after transformation to a Consumer. The communication endpoint of the Consumer can be derived from the notification (e.g. by including an email address into the notification).

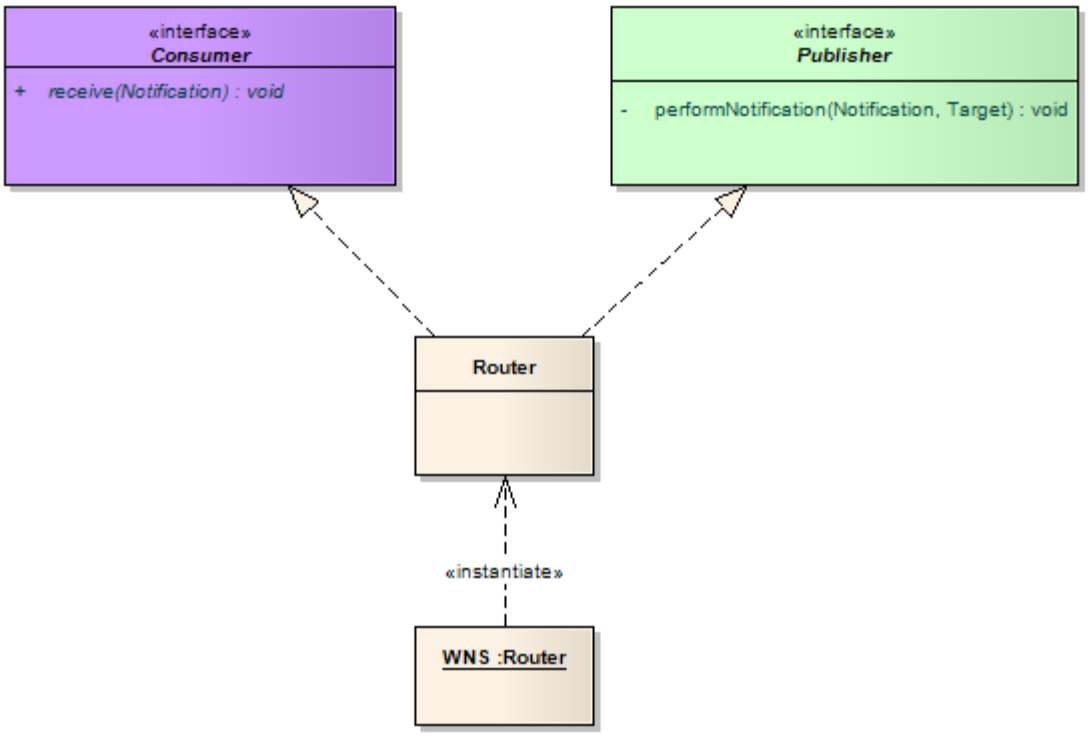

Figure 28 - Mapping the Router role to a WNS

In this scenario the WNS is an instantiation of the Router role (see clause 6.6.1.4).

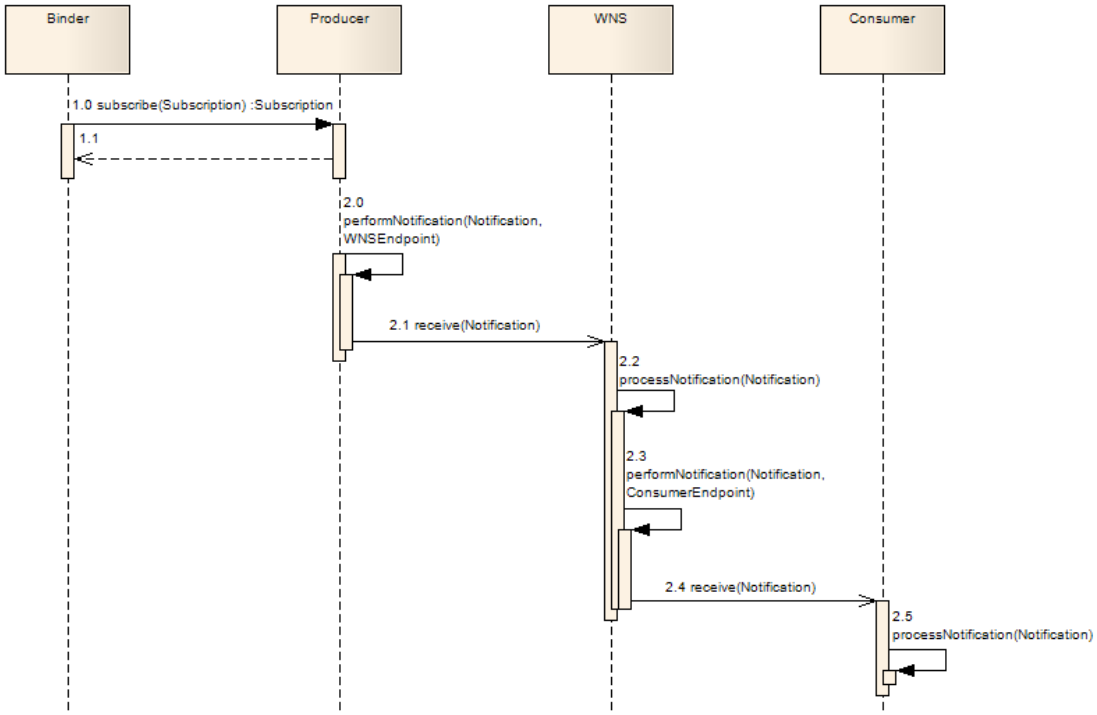

Figure 29 - Exemplary sequence of interactions between a WNS, client, binder and producer

The sequence of interactions shown in Figure 29 will be explained in the following.

1.0: A binder application invokes the "subscribe" method at the Producer. It provides the communication endpoint of the WNS in the subscription parameters.

1.1: The Producer responds to the subscribe request with a confirmation including (a reference to) the new subscription.

2.0: The Producer starts to perform a notification of the WNS. It therefore invokes its "performNotification" method with the actual notification containing the client's endpoint and the WNS' endpoint as parameters.

2.1: The Producer invokes the "receive" method at the WNS to deliver the notification.

2.2: The WNS processes the notification (i.e. transforms the notification and extracts the consumer endpoint).

2.3: The WNS invokes its "performNotification" method with the transformed notification and the consumer endpoint as parameters.

2.4: The WNS invokes the "receive" method at the Consumer to deliver the transformed notification.

2.5: The consumer processes the received notification.

In this scenario the notification provides the Consumer endpoint. This enables the WNS to determine the target for its notifications. If this cannot be achieved the WNS would have to implement the Provider interface in addition, resulting in the implementation of a Broker instead of a Router. Every Consumer would then have to be subscribed for notifications at the WNS. The drawback of this solution is that it is not as flexible as the Router solution.

7.6.3 Event types

In this scenario two types of data events are in use. The event to transform is the type of event the WNS receives. For this scenario it has to be machine readable and has to contain the communication endpoint of the desired Consumer. The other event type is the transformed event. It does not have to contain the Consumer endpoint anymore. It is furthermore not necessary that this event is machine readable, for instance if such an event is an automatic phone call. The only administrative events in this scenario are subscription events since no registration is performed.

Figure 30 - Exemplary event taxonomy for the WNS scenario

7.7 Event-Enabling a Web Feature Service

7.7.1 Introduction

The Web Feature Service (WFS) is used to provide access to vector data (features). Like with other data storage services (WCS, SOS, ...) a client can send requests to the WFS in order to get data. The WFS allows the usage of the OGC filter encoding to access only a subset of the provided data. The following scenario demonstrates how a WFS can be integrated into the event architecture. Like in the scenario for the SOS the techniques of the event architecture are used to publish data updates to the services.

7.7.2 Scenario

In this scenario the WFS shall be capable of receiving new features. Unlike the scenario of the SOS no registration of Publishers is used.

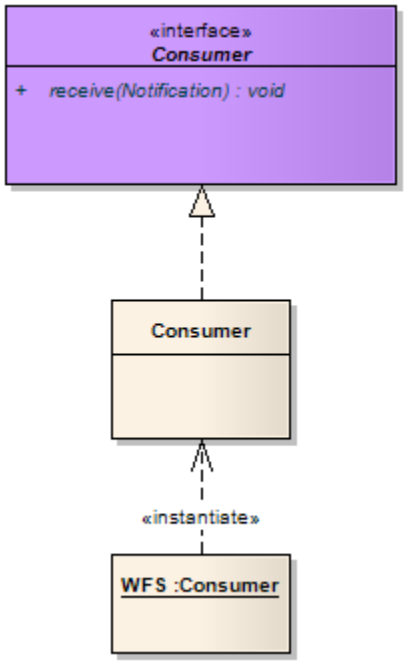

Figure 31 - Mapping the Consumer role to a WFS

In this scenario the WFS is an instantiation of the Consumer role (see clause 6.6.1.1).

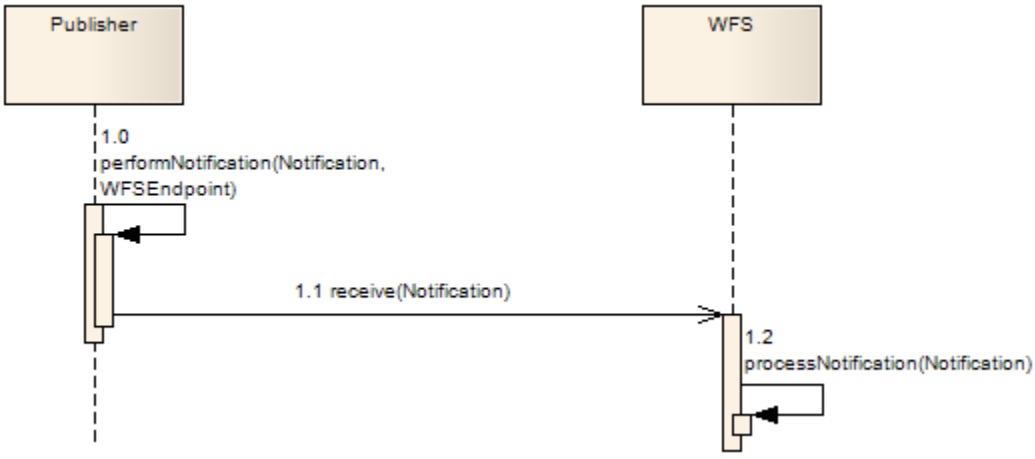

Figure 32 - Exemplary sequence of interactions between a WFS and a publisher

The sequence of interactions shown in Figure 32 will be explained in the following.

1.0: A Publisher invokes its "performNotification" method with a notification containing the new feature and the WFS' endpoint as parameters.

1.1: The Publisher invokes the "receive" method at the WFS to deliver the notification.

1.2: The WFS processes the notification. This includes the update of its database as well as its capabilities.

7.7.3 Event types

There is one data event type used in this scenario. The complete feature upload event contains complete features which are newly created or used to overwrite existing ones. The administrative events are constrained to ‘change notifications events’. They are used to publish changes to the feature set like newly available features or updates. There are no subscriptions or registrations used in this scenario.

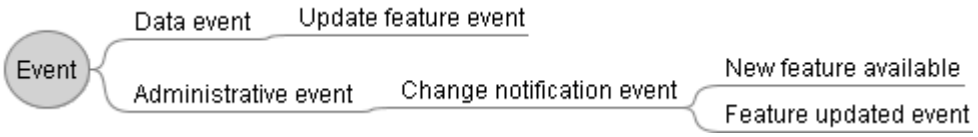

Figure 33 - Exemplary event taxonomy for the WFS scenario

7.8 Event-Enabling a Web Processing Service

7.8.1 Introduction

The Web Processing Service is usually used to process complex methods and algorithms on fixed data sets. In that cases event or push based communication is used. The following scenario shows how a WPS can support simple event based communication patterns.

7.8.2 Scenario

The processing result of a WPS process shall be published to a receiver when it is available. This is especially useful if the processing takes a long time.

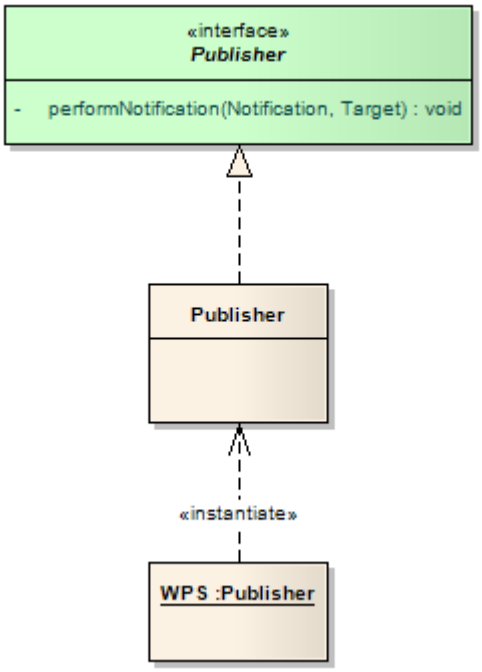

Figure 34 - Mapping the Publisher role to a WPS

In this scenario the WPS is an instantiation of the Publisher role (see clause 6.6.1.2).

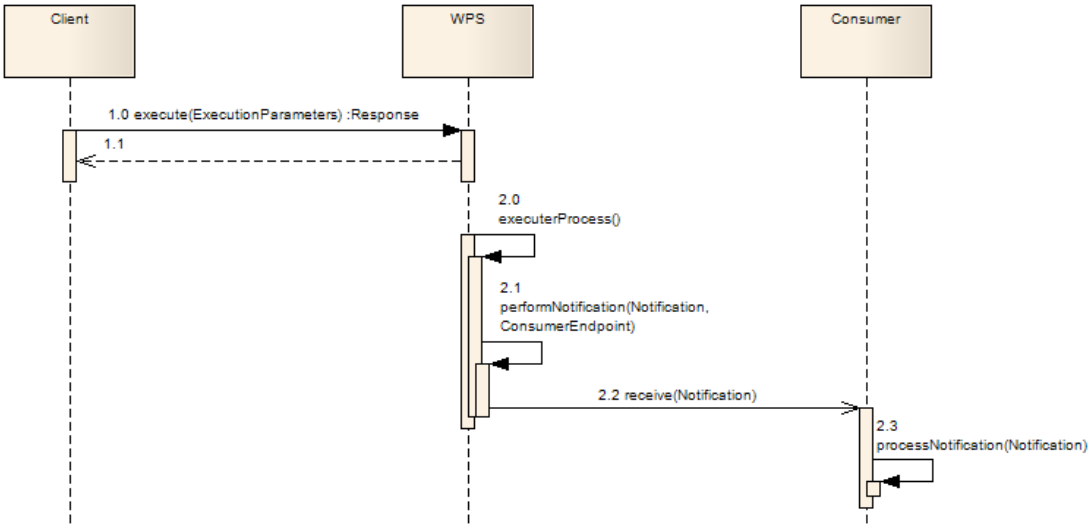

Figure 35 - Exemplary sequence of interactions between a WPS, client and a consumer

The sequence of interactions shown in Figure 35 will be explained in the following.

1.0: A WPS client accesses the WPS and invokes the "execute" method. The execution parameters contain beside the standard WPS parameters an endpoint reference to a consumer where the processing results are sent to when available. This consumer may of course be the client itself. Multiple consumers may also be allowed.

1.1: The WPS immediately responses to the request. This response does not contain the processing results.

2.0: The WPS executes the requested process.

2.1: When finished, the WPS invokes its "performNotification" method with the notification containing the processing results and the consumer endpoint as parameters.

2.2: The WPS invokes the client's "receive" method to deliver the notification.

2.3: The consumer processes the received notification.

7.8.3 Further integration

The WPS in this scenario can of course be extended to be more integrated into the event architecture. The WPS could for instance implement a Producer to allow subscription for status updates or intermediate results. This could replace the functionality described in the scenario above but does not have to. This is because it is not necessary in any case to subscribe at a producer to receive notifications. This depends upon the service and / or service policies.

Also the input data for the processes could be delivered as notifications resulting in a WPS implementing a Router or Broker.

7.8.4 Event types

In the scenario described above only one event type is used: the processing result event. The other event types shown in Figure 36 are used if one extends the WPS as described in the further integration section. The events for intermediate results and process inputs are of the type data event whereas status update events (like the process started event and the process state changed event) are administrative events.

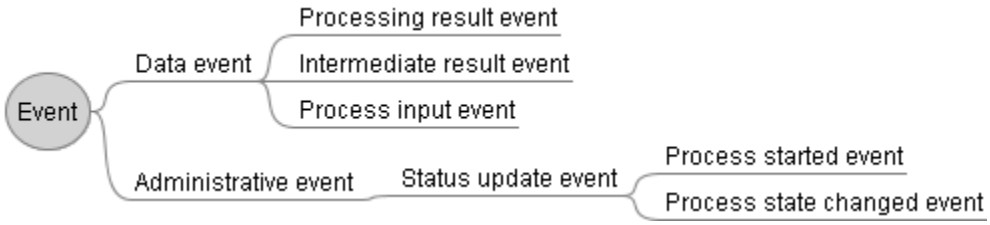

Figure 36 - Exemplary event taxonomy for the WPS scenario

7.9 Event-Enabling a Web Map Service

7.9.1 Introduction

The Web Map Service (WMS) is used to publish maps. These maps are not intended to provide data for further processing like WFS and WCS but to present results. These maps are served as image data sometimes using lossy compression methods like JPEG.

7.9.2 Scenario

A client is interested in receiving notifications that tell him whenever the list of map layers offered by a WMS has been updated. There are multiple WMS involved in this scenario. There is one group of services capable of publishing layerlist update notifications (WMS_A) and one service that is capable of brokering these notifications from other services (WMS_B). The access to the maps of the services of type A in this scenario is provided by the brokering WMS_B. A GetMap request directed to WMS_B may be forwarded to a WMS of type A if the requested map is served there.

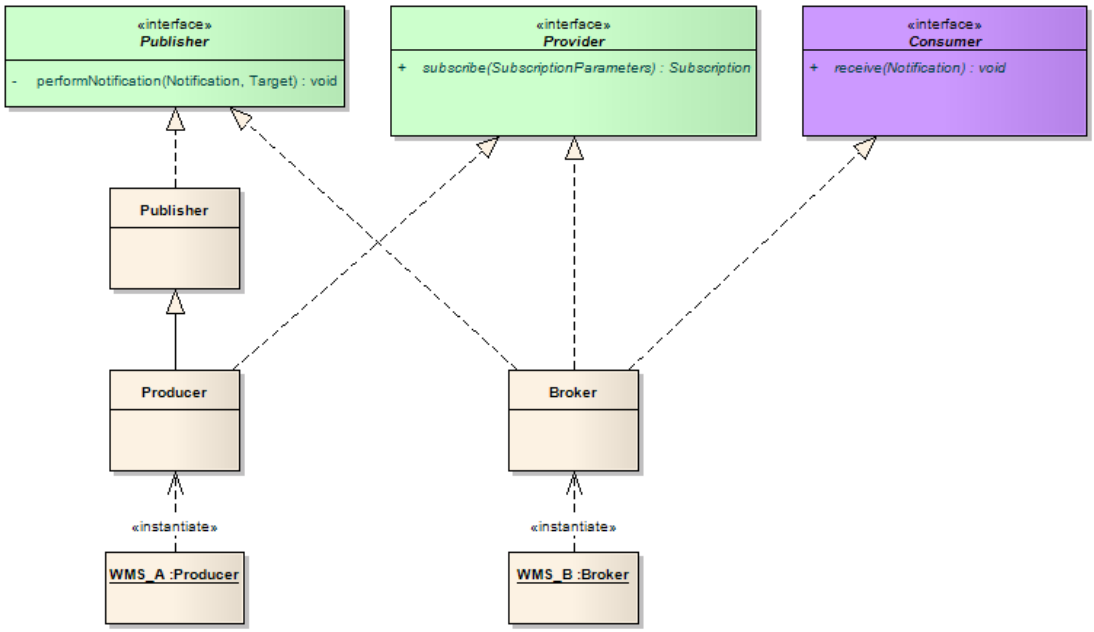

Figure 37 - Mapping the Producer and Broker role to WMSs

In this scenario the WMS_A is an instantiation of the Producer role (see clause 6.6.1.3) while WMS_B instantiates a Broker (see clause 6.6.1.5).

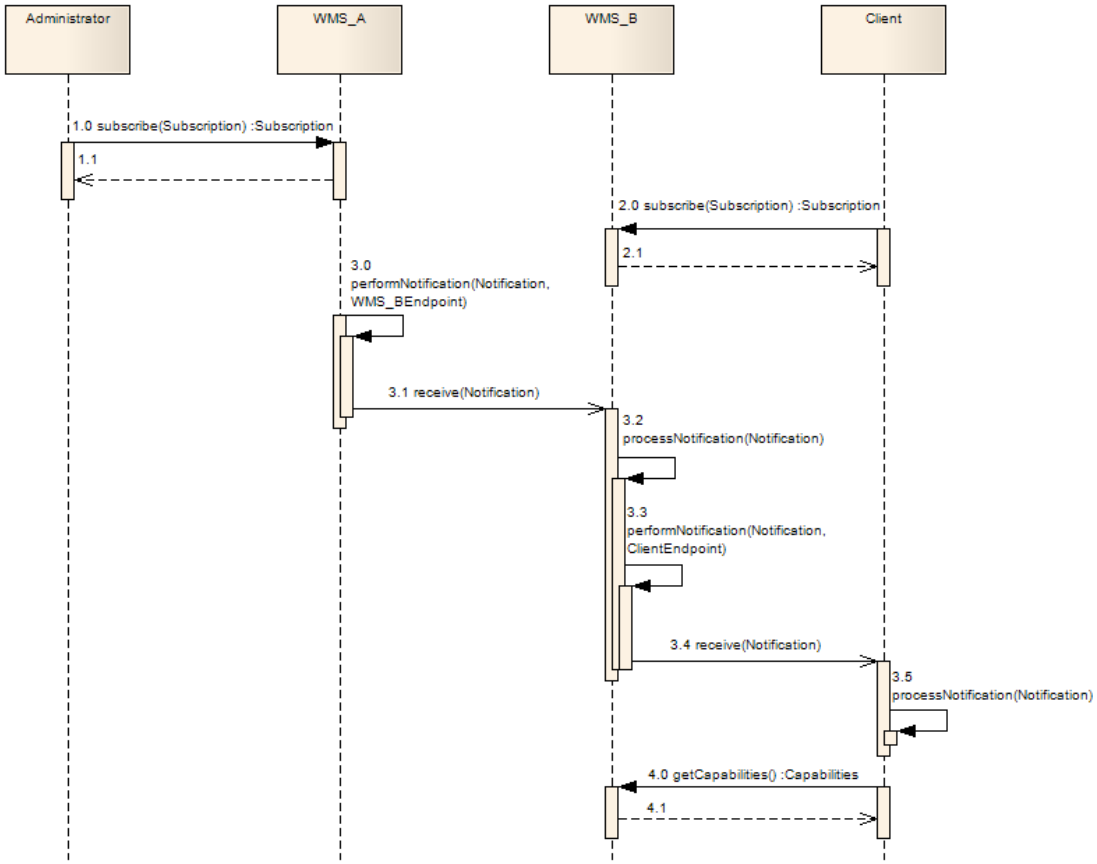

Figure 38 - Exemplary sequence of interactions between two WMS, a client and an administrator

The sequence of interactions shown in Figure 38 will be explained in the following.

1.0: An administrator subscribes the update broker (WMS_B) for updates on the layerlists at the WMS of type A. The administrator has to know the endpoint of WMS_B to include it in the subscription parameters.

1.1: The WMS_A responds to the subscribe request with a confirmation including (a reference to) the new subscription. These steps have to be performed for every WMS of type A in the system.

2.0: Now a client can subscribe at WMS_B for layerlist updates of this service (including updates of all WMSs of type A where WMS_B is subscribed to).

2.1: The WMS_B responds to the subscribe request with a confirmation including (a reference to) the new subscription.

3.0: If the layerlist of a WMS of type A is updated (e.g. layer added or removed by an administrator) it invokes its "performNotification" method with a notification containing an event for the update and the WMS_B's endpoint as parameters. This update event should contain information about the updated layer.

3.1: The WMS_A invokes the "receive" method at the WMS_B to deliver the notification.

3.2: The WMS_B processes the notification.

3.3: If the notification matches for a subscription at the WMS_B it invokes its "performNotification" method to forward the notification to the subscribed client.

3.4: The WMS_B invokes the "receive" method at the client to deliver the notification.

3.5: The client processes the notification. If the change in the layerlist may be of further interest to the client it can connect to the WMS_B to get more information (step 4.0 and 4.1).

4.0 and 4.1 (optional): If the update event promised for instance a new layer of interest the client could request the updated capabilities of WMS_B for further information. Alternatively it could directly access the layer data via a GetMap request or ignore this update.

7.9.3 Event types

In this scenario only administrative events are used. The data (i.e. maps) is transferred using WMS specific communication patterns. The events in this scenario can be separated in subscription events and layerlist updated events which are used to publish the information a client or brokering WMS (here WMS_B) subscribed for.

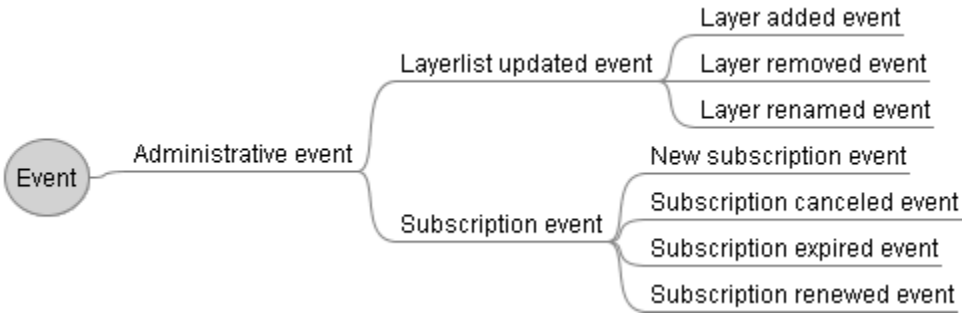

Figure 39 - Exemplary event taxonomy for the WMS scenario

7.10 Summary

The scenarios discussed above give a broad overview how different OGC services and components can be event-enabled. They cover most of the roles that are defined in chapter 6.6. They also cover most of the existing OGC services as well as clients and sensors. The only major services that are not discussed in the scenarios are the Web Catalog Service and the Web Coverage Service. Both can be used in the place of other services in at least one scenario (e.g. WCS instead of WFS, Catalogs instead of WMSs).

There are also a lot of different event types described along the scenarios. These types are partially service specific (e.g. the Layer added event of a WMS) and partially scenario specific (e.g. subscription events). The various event type taxonomies provided in the previous clauses are neither complete nor necessarily compatible. They give an idea what kinds of event types may appear in an event-enabled OGC architecture and may provide input for a future OGC event taxonomy. One has to be aware that such an event taxonomy will never be complete because every action that occurs in a system or an application may trigger an event thus resulting in a possible infinite amount of event types (at least one for every action human beings can think of). An OGC event taxonomy should rather focus on general super-types of events for which sub-events can be defined by service specifications and service instances.

In each of these scenarios multiple of the roles described earlier in this report are used whereas the component that is discussed has to implement only one. This helps if one tries to combine the functionality described in different scenarios. For instance if one wants to combine a SOS with a SES (SES would be the data source for SOS updates) one can replace the Producer in the SOS scenario with a SES which is implementing a Producer in its scenario. Some of the scenarios are also applicable to other services, for instance the scenario of the WFS could be translated to the WCS nearly one-to-one. Furthermore most of these scenarios can be extended simply by implementing additional interfaces. This could for instance extend a Producer to a Broker.

8 Related Technologies

8.1 Messaging Patterns

Communication between two or more systems in a distributed environment will involve message exchanges of various sorts. Many messaging patterns can be observed. In this report we will only cover the very basic patterns that are essential in an event-driven architecture. There are numerous books and web pages on SOA patterns which also describe messaging patterns; the interested reader is referred to these resources for more information on other patterns.

8.1.1 Datagram

Whenever no response is expected upon a message sent to a recipient, the datagram pattern is in effect (see Figure 40). This pattern is often used in an event architecture.

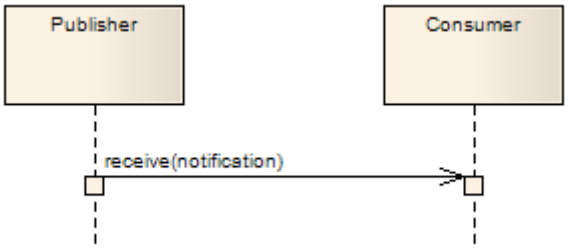

Figure 40 - message sequence in the datagram messaging pattern

The underlying transport protocol might indicate on a basic level whether the notification reached its destination. For example, when using HTTP to convey a notification, a simple HTTP response code of 2xx would indicate that the operation was successful. Whenever the publisher requires an acknowledgement (of the reception of the notification) from the consumer, some form of reliable messaging can be applied (see chapter 8.8.1).

WSDL 1.1 (W3C 2001) refers to this pattern as a *one-way* or *notification* message exchange, depending on whether the client or service sends the message. In WSDL 2.0 (W3C 2007a, W3C 2007b), the pattern is referred to as *in-only* or *out-only*, possibly also as *robust in-only*, *robust out-only*, *in-optional-out* and *out-optional-in* which are a mix of the datagram and request-response patterns (depending upon the message semantics and situation).

8.1.2 Request-Response

The most common case in client-server communication today is the request-response pattern (see Figure 41).

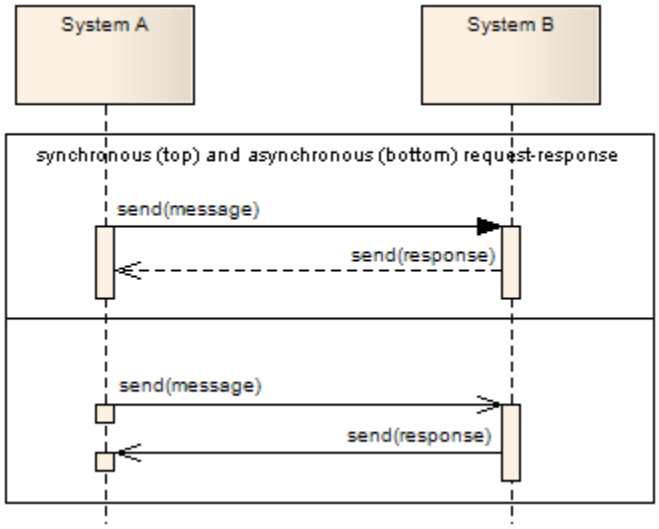

Figure 41 - message sequence in the request-response messaging pattern

In an event-driven architecture, this pattern will most likely be in use. We see that the communication involves a pair of request and associated response that are exchanged between two systems. The semantics of the request can be quite diverse. For example, it may invoke some process or retrieve data. Request-response communication may be performed synchronously as well as asynchronously. The difference between the two types and how to implement synchronous or asynchronous communication is explained in clause 8.2.

WSDL 1.1 (W3C 2001) refers to this pattern as a *request-response* or *solicit-response* message exchange, depending on whether the client or service sends the request. In WSDL 2.0 (W3C 2007a, W3C 2007b), the pattern is referred to as *in-out* or *out-in*, possibly also as *in-optional-out* and *out-optional-in* which are a mix of the datagram and request-response patterns (depending upon the message semantics and situation).

8.1.3 Publish / Subscribe

In this messaging pattern, one system - the producer - is capable of generating notifications and sending them to interested systems – the consumers. Systems express their interest in receiving notifications by subscribing at the producer. The subscription can also be performed by a distinct binder component. In any case, the subscription may indicate in which notifications the consumer is interested in. The producer will filter a new notification and send it to the consumer endpoints contained in matching subscriptions (see Figure 42). This pattern resembles the observer pattern known from OO-programming.

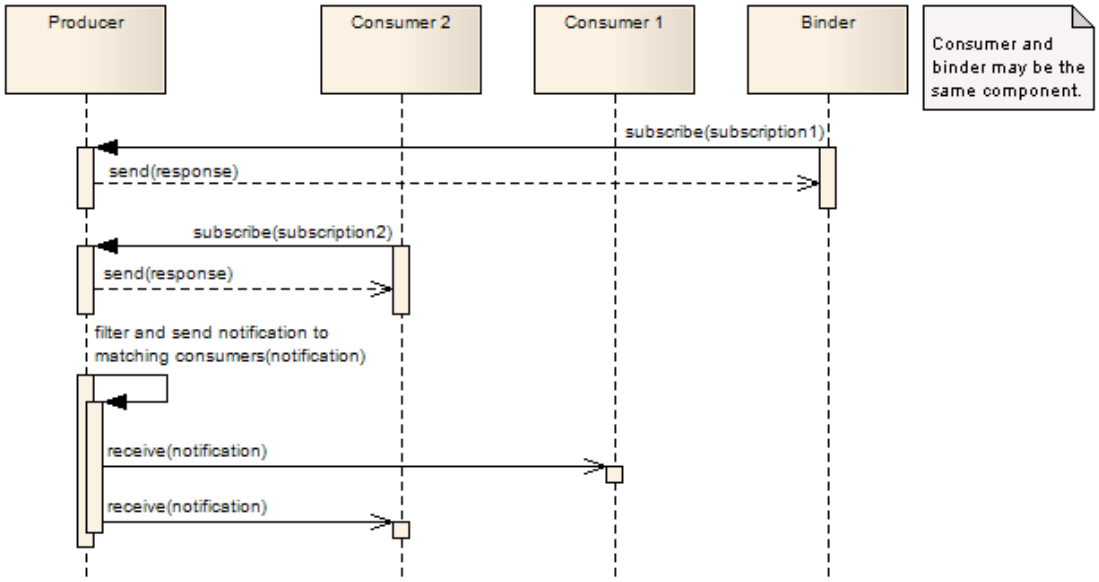

Figure 42 - message sequence in the publish / subscribe messaging pattern involving a producer

We see that this pattern makes use of both the datagram and request-response patterns. The latter is shown with synchronous communication, but subscribing could also be done in an asynchronous way.

A variation of this pattern or rather an extension is when the producer does not fully generate the notifications himself. In this case other publishers may send notifications to the producer, which is now acting as a broker (see Figure 43).

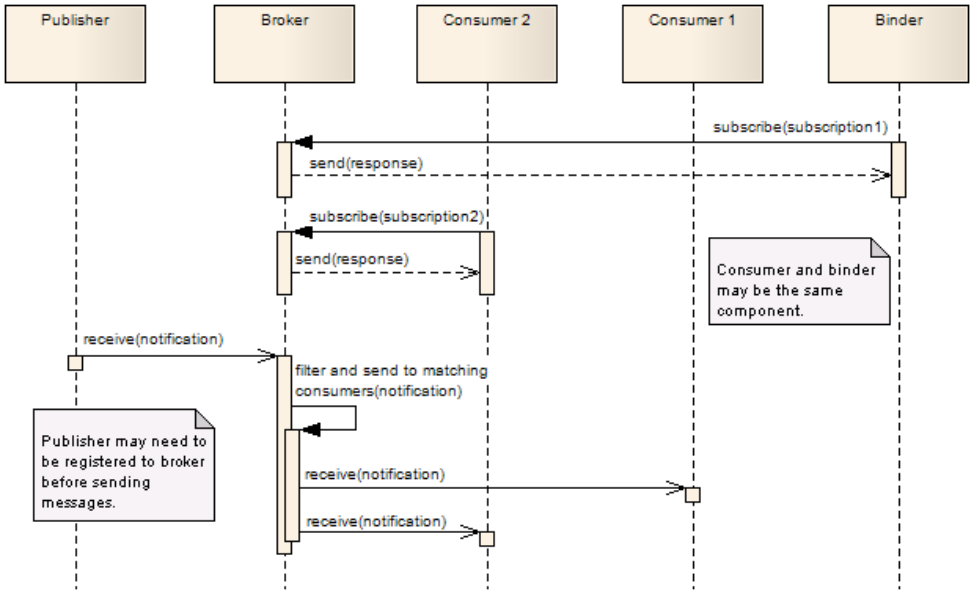

Figure 43 - message sequence in the publish / subscribe messaging pattern involving a broker

The origin of a notification may or may not be of interest to consumers. Thus, whether they receive notifications from a service that is a producer or broker is usually unimportant to them. Which subscription models exist will be explained in the following subclause.

8.1.3.1 Subscription Models

A subscription model characterizes how a new notification is matched against a subscription by the producer. Some models allow only subscription to predefined sets of notifications, while others allow content based filtering or even processing of new information on-the-fly. Faison (2006) lists four basic subscription models: *channel*, *type*, *filter* and *group*. We add another category, namely *event processing* (see Figure 44).

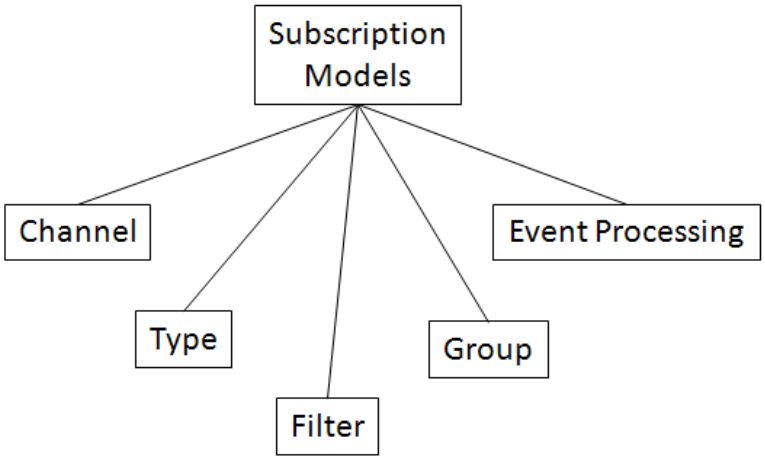

Figure 44 - Hierarchy of subscription models, according to Faison (2006) - modified

In general, a combination of the different models may be supported in one subscription. This would need to be defined per enabling technology. In the following subclauses, we will explain each model - with the exception of event processing, which is discussed in clause 8.5.

8.1.3.1.1 Channels

Notification channels are like television channels - if you view a channel, you receive all program broadcast on that channel. Producers use a defined mapping to associate notifications to channels. All notifications may be published on one channel, but they may also be distributed via several channels. A notification is not necessarily associated with one specific channel - it may be distributed over several channels. Subscribers may listen to several channels and should therefore be sensitive to duplicate notifications.

8.1.3.1.2 Types

Whenever notifications can be distinguished by a given type system, a subscription model that filters notifications based upon their type can be used. Either the notification message itself or its payload can be of different type. An example of different notification types would be an event type hierarchy (see clause 6.5.5). Many systems associate notification types to channels, which may result in one channel per type but also in multiple types per channel.

8.1.3.1.3 Filters

When a subscriber is only interested in a subset of all notifications available at a producer, filter-based subscriptions can be used. The subscriber would define the filter criteria which are checked by the producer every time a new notification is generated. Filters are usually applied on the notification content, with different level of complexity. For example, a large XML structure is not easily matched against a complex filter statement because usually the whole structure needs to be read and processed by the

producer. On the other hand, if notifications consist only of key-value pairs, they are much easier to filter.

8.1.3.1.4 Groups

Subscribers with commonalities, like using the same filter criteria, coming from the same geographic origin or organization etc. can be grouped together. Instead of matching each notification against each subscriber, a producer could match a notification against a group's subscription criteria and then deliver the notification to the whole group. This saves computing time both for a service and a client. A service can generalize the matching problem and thus does not need to perform matching for every single subscription. A client may save time and efforts when creating the subscription criteria by first screening already available groups and simply joining a suitable group. Even if a subscriber does not join a group explicitly, the service can assign a new subscription to an existing group internally if that group has the same criteria. In case that the group is established by the service upon reception of similar subscriptions, we have an *implicit group*. The opposite would be *explicit groups*, which are added to the system at runtime by an administrator (in contrast to *pre-defined groups*) and to which subscribers may join.

If users are assigned to multiple groups, then they might receive duplicate notifications - like in the channel model - because groups may be formed into hierarchies.

8.1.3.2 Realization in SOAP Binding

In the WS-* world, publish / subscribe can be handled with WS-Notification (OASIS, 2006a). WS-Notification makes use of notification topics, which support a mix of the subscription models *channels* and *types*. It also supports pre-defined and implicit *groups* as well as *filters*. A detailed tutorial on how to use WS-Notification is provided in Annex A.

8.1.3.3 Implicit Publish / Subscribe

Clients may be able to request streaming of matching data from a simple storage service, not only delivery of recorded data. This requires the retrieval request to contain a (filter) end time in the future and the service to provide the data stream either by reference or by using the response connection. This approach has been suggested by the OGC community, to enable implied publish/subscribe in already existing storage services without modification or extension of the existing interface and without implementation of an explicit publish/subscribe interface.

The approach requires the connection between client and service to be kept alive and open for as long as the streaming should take place or the client to provide an additional notification endpoint (somehow) so that messages can be delivered asynchronously. The former would require a policy to indicate that the service is capable of streaming data. The latter would require the endpoint information to be added in the request, which is easy in SOAP, but not in POX without an extension property (so would not work with most current OGC data storage services).

Also, adding this mechanism would not enable a generic implementation of publish/subscribe functionality in OWS clients. Existing clients would need to be modified to handle the new concept.

The mechanism can be introduced to OGC services provided that services indicate the available behavior via policies. However, once different transport protocols come into play (e.g. given indirectly by returning a URL to a stream, which may be HTTP but also other protocols) for streaming data, the situation gets complex for clients.

We therefore suggest to use a common, explicit approach for publish/subscribe which both clients and services readily understand and which can be applied to all OWS. For sure, the content type of messages will differ in given use cases, but at least how subscriptions are created and how consumers are notified would then be the same.

8.2 Asynchronous Communication

Usually, a communication between a client and server involves a synchronous request-response message exchange, meaning that the client sends its request to the service and then waits for the services response. Depending upon the situation, the service might not be able to provide the response directly, for various reasons. It depends upon how long the connections between client and service can be kept open and on how long the client is willing to wait for the response for the message exchange to be performed synchronously.

To solve the problem of a connection timeout or the client waiting for the response for an indeterminate amount of time, an asynchronous message exchange can be performed. Here, the connections between the service and client will be closed once the request has been sent. The service will send the response to a given endpoint when the response is available. The address of this endpoint is usually provided by the client but could also be provided by the service when acknowledging the receipt of the request.

Synchrony or Asynchrony in its purest form are defined by the way the client is programmed. If the process which sent the request message to a service waits/blocks until the response is received, the client is programmed to expect synchronous behavior. If the process just sends the request, then continues to perform other computations and simply reacts to the eventually received response, it is asynchronous. What happens in case that the response is not received in time is up to the process that sent the request. Different models could be in effect. For example, when sending a Submit request to a Sensor Planning Service (OGC 07-014r3), the client may indicate until which point in time the definite response from the service (telling the client whether the submitted task was accepted or rejected) has to be sent to the client. If this condition cannot be met, then the request is agreed to be rejected and the client can proceed - for example by sending a new request. Sending a response by establishing a new connection between client and server is often also considered as asynchronous communication. What is meant with asynchrony and asynchronous communication therefore depends upon the context.

In distributed systems like a web service environment, the underlying transport or used application protocol need to provide a mechanism which circumvents connection

timeouts so that a response can be sent asynchronously to a client. In the following, we will describe which mechanisms exist for the bindings commonly used by OGC services.

8.2.1 Realization in SOAP Binding

In the WS-* world, asynchronous message exchanges are handled using WS-Addressing (W3C, 2006). Here a server can indicate whether WS-Addressing is required in communications with the service. Clients can add a WS-Addressing endpoint containing a reply-to property, the value of which denotes the endpoint to which the service will send the response asynchronously. Note that when WS-Addressing is used, the service will always send the response to the reply-to endpoint unless it is not provided or anonymous.

There are numerous tutorials on WS-Addressing. Thus we will not go into more detail on how to use the technology in the SOAP binding of OGC services at this point. Chapter 10.2 provides some more detailed information on the usage of WS-Addressing.

8.2.2 Realization in POX Binding

Many existing OGC specifications use Plain Old XML (POX) requests sent via HTTP POST to a service, retrieving the response synchronously via the HTTP response. This form of RPC can be performed asynchronously by just wrapping the request in a SOAP envelope and using WS-Addressing.

8.2.3 Realization in REST Binding

REST bindings are not common for OGC services yet, thus the experience with this binding type is low. However, we try to provide a suggestion based upon the findings of Tilkov (2008) on how to realize asynchronous message exchanges - or rather resource exchanges - with REST.

Using HTTP, the response code 202 (Accepted) could be used to indicate that "the request has been accepted by the service for processing but that processing has not been finished yet". How would the client then get the response? First of all, we need to define what this response would be. It can be a resource in a given representation but might also just be an http response document for the initial request, in case that this was the expected response from the server. Following the approach that a resource is expected as a response, the client could provide a URI that it expects the server to POST the result to once it is finished. Then subsequent GETting of this URI could be performed to eventually retrieve the server response. The server could also provide such a URI (to a resource which would specifically be created for the request) in the initial response, however then the http response code 201 (Created) seems to be more appropriate. Here the client would also GET the response from the given URI.

8.3 Event Driven Architecture

The Event Driven Architecture (EDA) is an architecture style in computer systems. An architecture is event driven if "some of the components are event driven and

communicate by means of events" (Luckham, Schulte 2008). Other definitions are stricter and require that all components are event driven which is a very rare case in implementation (Luckham, Schulte 2008). Furthermore events shall be processed on arrival and not specify any operations (Chandy, Schulte, 2007a and 2007b).

Event Driven Architectures (EDA) and Service Oriented Architectures (SOA) are complementary. One can use an event based communication model in SOA in addition to a Remote Procedure Call (RPC) based model. In this sense EDA is a special form of SOA.

8.4 Enterprise Service Bus

An Enterprise Service Bus (ESB) is a communication middleware for SOAs. Chappell (2004) defines an Enterprise Service Bus (ESB) as "a standards-based integration platform that combines messaging, web services, data transformation and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications across extended enterprises with transactional integrity". The extended enterprise thereby may include business partners of an organization. Usually the communication (messaging) is based on SOAP with an XML encoded payload. An ESB thereby acts very similar to a hardware bus in a computer. Every member of the bus architecture only reads and writes messages from and to the bus. There are no direct links between single members. This helps when building large scale applications with many services. An ESB can also provide centralized solutions for security aspects like authentication, authorization and single-sign-on.

Figure 45 shows a schematic view of an ESB. Multiple services are connected to it. Some applications might need a translator in order to communicate with the bus. Figure 46 shows the same services and application connected without an ESB. This might result in a lot of different connections but also in the need for more translators.

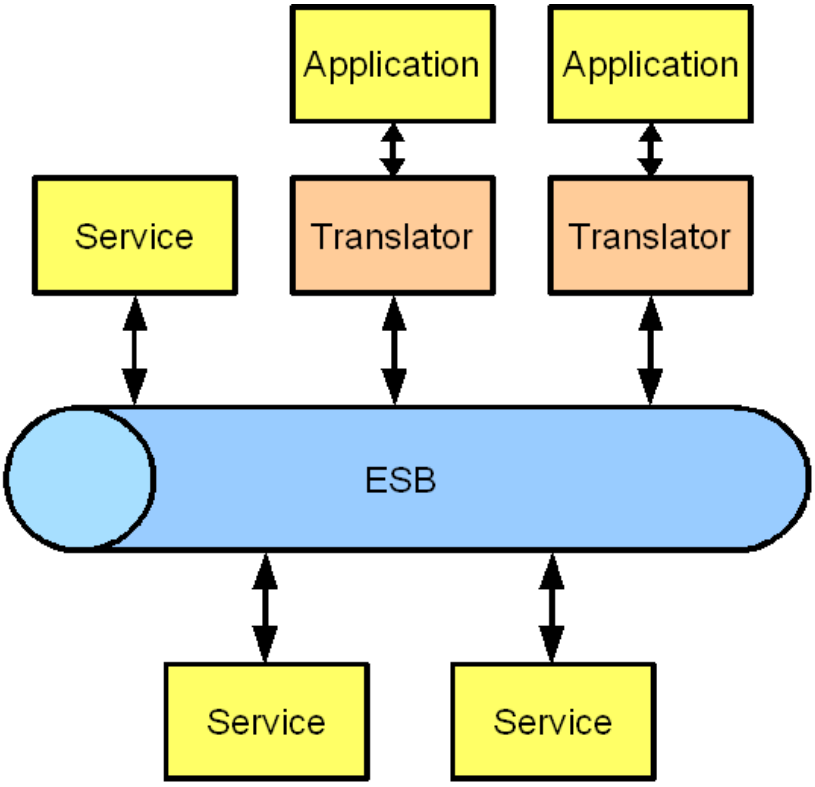

Figure 45 - Distributed system with an ESB

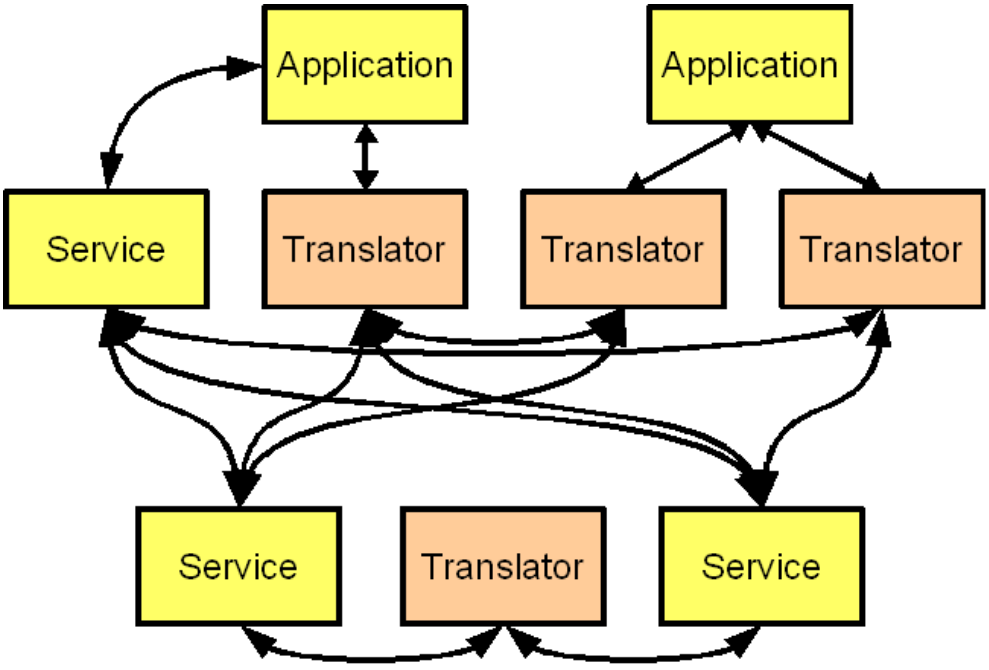

Figure 46 - Distributed system without an ESB

8.5 Event Processing

Events and therefore also some kind of event processing were already used in operating systems of the late 1950's and early 1960's. These days the event processing was more or less context switching.[10] Since then event processing has emerged to "computing that performs operations on events, including reading, creating, transforming and deleting events" (Luckham, Schulte 2008). The component performing the event processing is called *event processor*. Event processing is for instance used in many graphical user interfaces like Java AWT.[11]

Complex Event Processing (CEP) is a specialization and enhancement of event processing. It is defined as "Computing that performs operations on complex events[12], including reading, creating, transforming, or abstracting them" (Luckham, Schulte 2008). The main concepts of CEP are the use of relations between events, event patterns including event filters and constraints, event pattern triggered rules, complex events, event pattern abstraction and hierarchies of events (Luckham 2007). These concepts as well as further topics and terms are described in the following clauses.

8.5.1 Relations Between Events

"Simple" event processing is based on the processing of single events. In contrast CEP relations between multiple events are used to detect situations of interest and to derive information. The most common relations used are timing, causal, spatial and similarity relations like "event A happens before event B", "event C caused event D" or "event E happens close to event F". For instance the event "Ordering a meal" caused the event "Cooking a meal" or the event "Cooking a pizza" is akin to the event "Cooking a burger" whereas it has to be stated that the measurement of similarity is not a simple task (see chapter 6.3.6).

8.5.2 Event Patterns

In the most common form event patterns are rules expressing relations and / or combinations of events which are used for the pattern matching. Examples for event patterns are "event A followed by event B" and "event C and event D happen during one day". Often they are enhanced with guards, attribute conditions that have to be fulfilled like "event E where E.value > 15". When using an event pattern with guards but without relations or combinations of events (like in the last example) the pattern defines an event filter.

The search for and finding of events that match these rules is called pattern matching. The pattern matching is the central operation in CEP performed by so called pattern

---

[10] see A Short History of Complex Event Processing Part 1 - 3 by David Luckham (available at: complexevents.com) for more details.

[11] For more information on AWT see http://en.wikipedia.org/wiki/Abstract_Window_Toolkit.

[12] In this chapter, we do not differentiate between derived and complex events (see chapters 6.3.1 and 6.5.2). A derived event is a specialization of a complex event. The exact types of complex events discussed in this chapter are given by the context.

matchers (also called CEP engines, similar to event processors). The event patterns are expressed via an Event Pattern Language (EPL) like the Rapide-EPL[13], the Esper-EPL[14] or the EML (OGC 08-132).

Event patterns that are used to detect situations that should never occur are called event pattern constraints.

8.5.3 Event Pattern Triggered Rules

Triggering of rules or actions is not a development of CEP. In a graphical user interface such a rule could be "when button X is pressed perform action Y". In CEP one can also define rules and actions to perform that are triggered by event patterns. Example: "if event ‘Meal ordered’ followed by event ’Meal_cooked’ where Meal_ordered.customer_id = Meal_cooked.customer_id then serve meal to customer with id = customer_id".

8.5.4 Complex Events

A complex event is "an event that is an abstraction of other events called its members" (see Luckham, Schulte 2008, chapters 6.3.1 and 6.5.2). These events are also called higher level event because of their abstraction level. An example for a complex event is "buy a car". It is an abstraction of the inner or member events "find a car that one likes", "negotiate the price", "pay the car" and "pick up the car". These events themselves may be complex events. For instance "find a car one likes" consists of the members "read a car magazine", "test car 1", "test car 2" and so on.

8.5.5 Event Pattern Abstraction

The idea of complex events comes together with the idea of event pattern abstraction. The abstraction of a set of events is often done by event pattern triggered rules combining a set of events in a complex event (for instance "event A and event B create event C containing A.value and B.value").

8.5.6 Causality Models

When using complex events one can define multiple abstraction levels. In case of an indoor fire detection system one can define (abstraction) level 1 as the measurements (events) of the sensors. Level 2 can be the first warning level like "high temperature detected" or "smoke detected". These events could again be abstracted in level 3 to the event "fire detected". Additionally there is a set of abstraction rules defined for each of these levels. For instance "if event ‘high temperature detected’ and event ’smoke detected’ then create new event ’fire detected’".

---

[13] see Luckham (2002)

[14] see http://esper.codehaus.org/

Figure 47 shows a causality model with simple events (level 1), complex events (level 2 and 3) as well as the abstraction rules with reference numbers.

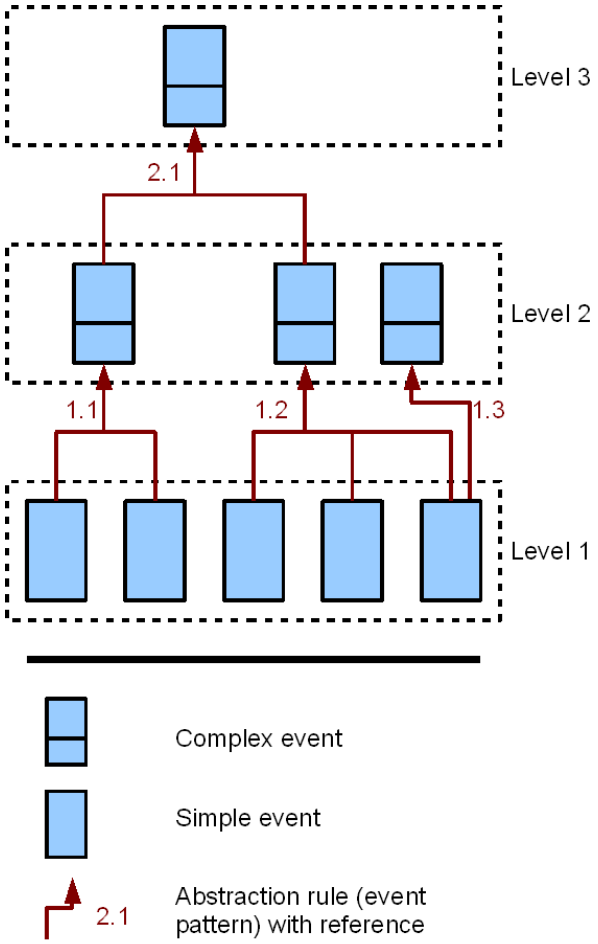

Figure 47 - Causality model and abstraction rules

8.5.7 Causal Vector

The causal vector is an optional property of a complex event. Due to its influence on the pattern matching as well as the design of digital event representations we will discuss it in more detail.

As the name causal vector indicates the property is a list. In this list all (known) member events of the complex event are stored. These member events themselves may be complex events themselves but do not have to be. This concept is known as the design pattern "Composite" (Gamma et al. 1995). One can also represent the causality (all member events and their member events and so on) as a tree. The complex event at hand is the root node and all member events are the children. Simple events are leafs in such a tree while complex member events again are nodes with all their member events as children.

Figure 48 shows a complex event (A) with all its members in a tree representation. The causal vector of event A contains a complex event which itself again has some member events (B1), a complex event with an empty causal vector (B2) and a simple event (B3).

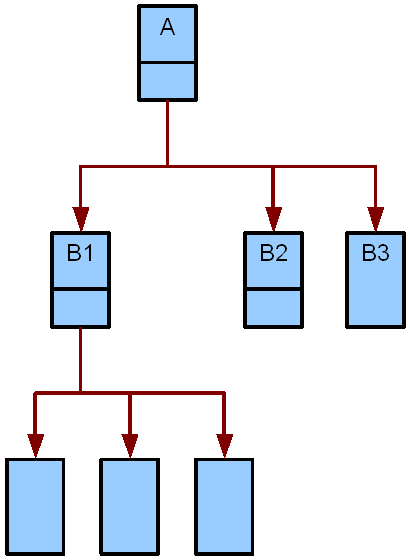

Figure 48 - Tree representation of a complex event

The causal vector stores the causal relationship of one event to other events. Therefore it is possible to reproduce and validate results of pattern matching. It is also possible to use causal operators in event patterns like "event A caused event B" or "event C is parallel to event D". The operator "is parallel to" is also called "is independent to". Furthermore, to use the causal vector it must be defined how it is populated. This is usually done when creating the complex event (for instance via event pattern triggered rules). The event creation rule can be set to create the causal vector of a new event. This has to be defined for each creation rule or for the whole rule execution engine.

When the focus of an application is to validate and reproduce situations in a system it may be an advantage to enrich the complex event with information about the algorithm that was used. This can also be achieved by defining a causality model including the abstraction rules. But when a higher level event is the result of multiple event patterns (or abstraction rules) it can be very difficult to determine the one that was used. Thus it is useful to add or reference the event pattern.

Figure 49 shows an example instance of a causality model for the previously mentioned indoor fire example. The simple events A and B are the measurement events for the temperature and smoke sensors. The event patterns 1.1 and 1.2 check the measured values against a threshold and report warning events (D and E) when the threshold is exceeded. These event patterns are relatively simple filter patterns not using relations between events. The event pattern 2.1 combines the warning events D and E. If they appear closely in time a fire alert is triggered (F). This example also contains a second possible way to trigger a fire alert. Think of a manual fire alert trigger. When its activation (event C) is received the fire alert event (F) is created (event pattern 1.3). Thus

in this example it is useful to have a reference to the event pattern applied to the causal vector of the fire alert event.

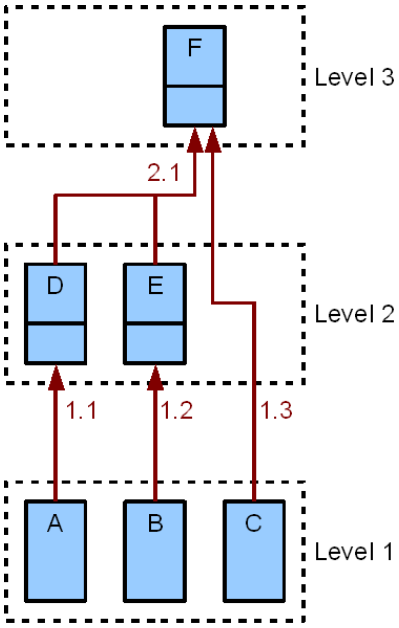

Figure 49 - Exemplary causality model

8.5.8 Event Cloud

The event cloud is the set of all available events. In more detail it is "a partially ordered set of events (poset), either bounded or unbounded, where the partial orderings are imposed by the causal, timing and other relationships between the events" (Luckham, Schulte 2008).

8.5.9 Event Stream Processing

Event Stream Processing (ESP) is a subtype of CEP (Luckham 2006). Instead of operating on the whole event cloud the events arrive only as event streams. Because these streams are contained in the event cloud ESP is considered as a subtype of CEP. The processing of data streams comes with some problems and also advantages. One problem is that data streams are of possibly infinite size, making it impossible to store the events as well as to perform queries on the whole stream or events in complete history. On the other hand, the evaluation of the "followed by" operator is easier because the order of events is maintained in a stream.

In order to allow queries to the history to some degree and also to speed up the pattern matching one can use data views (also called windows) in ESP. These views define a subset of the event stream and only events contained in these views are available for pattern matching. Common views are defined by the maximum number of events they can hold (e.g. the last 25 events) or the time how old the events may be at most (e.g. events from the last 10 minutes). There are also a lot more possible ways to define views, for instance depending on attribute values of the events or even spatial constraints.

Another possibility that arises from the views is that one can use triggered rules to generate values depending on the whole content of a view, for instance the average attribute value of all events received in the last 10 minutes.

8.5.10 Applications

Today, CEP is used in a wide area of applications such as Business Activity Monitoring (BAM), fraud detection (e.g. in financial services), supply chain automation and network monitoring. In many applications the use of sensors and the processing of their measurements (-> events) play an important role, for example RFID or bar code sensors in an airport's baggage control system.

8.6 Different Times of Events

As we said in clause 6.3.4.2.2, every event type may have multiple temporal properties. These properties may express different semantics. Sometimes, though, they may have the same semantics and the only difference is that the time value is given in a different temporal reference frame. In the following, we will discuss different aspects of temporal event properties.

8.6.1 Different Semantics

When working with temporal properties of events one has to be aware of their semantics. Possible times for events are the *time of action*, the *time of detection* or the *time of processing* - among others which can play a role in event processing.

The time of action describes the time when the event happened[15]. It may be a time instant or interval. On the one hand this time seems to be the most usual but on the other hand it may be hard to obtain. For many real world occurrences the time of action has to be calculated or estimated. In case of a wildfire, for example, the start time of the fire will often be unknown because the fire itself is unknown until it is detected.

The time of detection describes the time when an occurrence was detected. This time is easier to obtain because it only has to be recorded instead of calculated or estimated. The time of detection is often a time instant. Multiple times of detection can be aggregated to a time interval (e.g. into the time of action) of a higher level event. For example the detection of high wind speeds (detection of the start of a storm) and later of low wind speeds (detection of the end of a storm) can be aggregated into a storm event with a time interval from start to end.

The time of processing describes the time when an event was processed. This time could be added to the event. The time of processing may be a time instant (start time of the processing) or interval (the processing duration). When multiple processors or processing steps are involved, multiple times of processing may be added to an event. In this case, the following processing steps could be ignored and only the initial start time of the

[15] Note that the time of action is a synonym of *event time* and used in this clause to emphasize the different circumstances under which this time can be assigned to an event.

processing be retained as time of processing or all processing times be aggregated in a collection. It is also possible to aggregate all times to an overall time interval describing the total processing duration. A disadvantage of this solution is that temporal gaps in the processing chain may result in very long processing durations. If the time of processing shall be used it may also be important to know which processor added the time or what kind of processing was performed. This can be difficult when multiple processing steps were performed.

8.6.2 Different Temporal References

Different temporal references and reference systems may be due to various reasons. An event could for example have its origin in Taiwan where the time is GMT +8 and the year 2009 has the number 98 (the year 1912 is year 1 in Taiwan). But there are also other temporal reference systems than the Gregorian calendar. For example, in Java one can represent times in milliseconds since January 1st 1970. There are also logical clocks that are only used to capture temporal relations like the *lamport timestamps* or the *vector clock*[16]. They are used in distributed systems and do not have a fixed (in time) zero value.

In distributed systems one is also faced with the problem of clock synchronization issues. Due to delays in the communication between components it is impossible to fully synchronize two clocks on distributed machines. If a very high temporal resolution is needed this can be a problem.

8.6.3 Time Instant and Time Interval

Time instants are usually used as the type of temporal event properties in the following cases (compare with chapter 6.3.4.2.2): 1. if the event is immediate - like a threshold exceeding - and 2. if the duration of the event is below the desired temporal resolution. The latter may be the case for an *Explosion* event. Here, the explosion itself lasts for a certain duration but it may nevertheless only be modeled as a time instant. In the wild fire example the *Wildfire* event has a time of action which is a time interval and a time instant as time of detection. A *WildfireDetected* event would have a time of action with a time instant as value. Complex events (like the *Wildfire* event, see chapter 8.5.4) usually take place over a time interval (Luckham 2002).

8.6.4 Times of Sub-Events

In case of complex events an event may contain multiple sub-events. They again can be complex events with further sub-events and so on. In addition to the times of the complex event all of its sub-events (and of course sub-sub events etc.) may have multiple temporal properties with values of different types. For these complex events the time of action is often defined as the temporal bounding box of the times of action of all sub events (Luckham 2002). The time of detection of a complex event is the time of the creation of the complex event. This is usually the time of processing of the processing step creating

[16] For more information on these types of clocks see Lamport (1978), Fidge (1988) and Schwarz, Mattern (1994).

the complex event. If the time of processing is a time interval the end time should be assumed as the time of detection.

Creating the temporal bounding box of the different times of sub-events can become complex especially if these times are not provided in the same reference system. These problems have to be addressed when designing an application that makes use of complex events.

8.6.5 Problems With Multiple Times

Having multiple temporal properties in events can cause problems and thus care should be taken when processing these properties. On the one hand, correct selection of a temporal property is essential. This can become difficult if properties of associated events need to be addressed or the type of a temporal property is complex. On the other hand, it is essential to understand what the result of your processing is, especially if they involve temporal properties with different semantics.

8.6.6 Opportunities With Multiple Times

Having temporal event properties of different semantics provides for a more detailed description of an event. This in turn allows for enhanced functionality in detecting patterns in the event cloud and therefore more powerful event processing. In addition, especially during the development phase of an event architecture, properties with time of processing can help to identify gaps or bottlenecks and thus to increase performance.

8.6.7 Recommendation

In order to keep a system as simple as possible but as complex as required it is recommended to start the system development by designing an application schema with all the event types required in that system. Each event type should contain detailed information about the temporal properties involved, including the semantics of these properties, the temporal reference and the type (time instant or interval). It is also recommended to use only one temporal reference frame and representation in the whole system (for example UTC in combination with ISO 8601). Furthermore the rules for creating complex (higher level) events (which should be contained in the application schema) need to define how the mandatory temporal properties of these events are populated.

8.7 Transportation of Events

Events can be transported using different ways. In this chapter a short overview is given which techniques can be used with a focus on event transportation via the internet. The transportation of events inside of an application is not discussed here. When discussing data transportation via the internet one cannot skip the reference to the OSI Reference Model (see Table 4). In this well known model multiple layers involved in data transportation are defined. In this chapter some examples for protocols from the layers four to seven are given, namely: UDP, TCP, HTTP, SMTP, XMPP and RTP. All of these

protocols establish connections from one endpoint to another without dealing with routing. This is solved via protocols of the lower layers.

Table 4 - Layers of the OSI model

Layer number	Layer name
7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data Link
1	Physical

8.7.1 UDP

The User Datagram Protocol (UDP) is a simple transportation protocol. It uses IP based networks to exchange messages (so called datagrams). It is connectionless which means that no handshake is required between two communicating parties prior to the message exchange. Furthermore it is unreliable resulting in datagrams arriving out of order, duplicated or not at all. The latter is a large obstacle when building event driven applications like early warning systems where reliability is a very important factor.

8.7.2 TCP

The Transmission Control Protocol (TCP) is also a transportation protocol. Like UDP it uses IP based networks but in contrast to UDP it is reliable and connection oriented (handshake needed). The cost of reliability is a lower performance thus applications with real time requirements are often based on UDP[17].

8.7.3 HTTP

The Hypertext Transfer Protocol (HTTP) is an application protocol. It is widely used to transfer web pages via the World Wide Web (WWW). It uses TCP as transport protocol. It is not restricted to transfer hypertext encoded messages as its name may suggest. HTTP defines a set of methods that can be used for the communication including GET, POST, PUT and DELETE. These are the main methods used in REST to build web services. In a common web service approach GET and POST are the typical HTTP methods.

8.7.4 SMTP

The Simple Mail Transfer Protocol (SMTP is an application protocol that uses TCP as transport protocol. It is developed for the exchange of emails and is broadly used in mail

[17] Note that real time processing of data does not mean that it is processed live (i.e. without storing) but has to keep predefined time limits. A common event processing application is usually not a real time application, in contrast to VoIP (Voice over IP) or video streaming applications.

servers[18]. The messages exchanged via SMTP can of course contain events, thus using emails for notifications.

8.7.5 XMPP

Like HTTP and SMTP the Extensible Messaging and Presence Protocol (XMPP) is an application protocol that uses TCP for transportation. It was formally designed for instant messaging and presence information exchange via Jabber but is meanwhile used in a broader set of applications. The OGC Sensor Alert Service (OGC 06-028r5) for instance uses XMPP for notification transport.

8.7.6 RTP

The Real-Time Transportation Protocol (RTP) is an application protocol and uses UDP as transport protocol. It is commonly used for the exchange of real-time sensitive data like audio and video streams where the loss of single packets is less important than small delays.

8.8 Acknowledgement and Reliable Messaging

An acknowledgement is a kind of response to a message from the receiver. In this chapter two different usages of acknowledgement are presented. At first reliable messaging is introduced where acknowledgements are used to verify that a message arrived at a given endpoint.

Afterwards the use of acknowledgements to confirm not only the reception but also that an appropriate action has been or will be taken is discussed. The latter usage can be called explicit acknowledgement because the response has to be triggered. This does not mean that a human has to be involved. An automatic system can also acknowledge a notification explicitly, by sending an according message.

8.8.1 Reliable Messaging

Whenever messages are exchanged across system boundaries, reliable delivery becomes important. We want to know whether a message that we sent has reached its recipient. Message delivery in distributed systems has different QoS aspects. The following criteria are of interest: ordered delivery, duplicate elimination and guaranteed receipt. Ordered delivery applies to a sequence of messages sent from a source to a destination. The transport protocol used may mess up the order in which the messages have been transmitted by the source. The destination will need to restore order somehow. To guarantee the receipt of a message sent by the source, a destination usually needs to acknowledge each message it received. If a certain message is not acknowledged, the source should resend it. This strategy is also called *certified* delivery (see Faison 2006). Whenever a source resends a message the destination might receive message duplicates.

[18] Email clients often use SMTP only for sending Emails.

To detect these duplicates and to avoid forwarding them is QoS that a destination may or may not provide.

8.8.1.1 Realization in SOAP Binding

Support for reliability in WS-* environments is provided through the WS-ReliableMessaging specification (OASIS 2009a). Here, certified delivery is applied. A web service may indicate that usage of WS-ReliableMessaging is required for given operations via policies (OASIS 2009b). A notification source may request the creation of a message sequence identifier at the destination which may then be used by the source in the transmission of subsequent messages. Each message in the sequence has to be numbered. The destination has to acknowledge which messages it received when it is requested to do so by the source. If the destination did not acknowledge a given message, the source should re-send the message until an acknowledgement for it was provided or until the sequence is terminated. Such a termination indicates the end of the reliable communication between source and destination.

8.8.1.2 Realization in REST Binding

An HTTP response code of 2** usually means that the message sent to a destination has been received. In case that no HTTP response is provided at all, you do not know whether the message you sent or the response itself was lost. You could simply resend the request. This is safe - meaning that duplicate requests will automatically be ignored - for GET, PUT and DELETE messages, but not for POST. If the semantics of your POST request can be mapped to PUT, then you could switch to using PUT. If that is not an option, one of the approaches mentioned by Tilkov (2008) can be applied. However, there seems to be no established best practice on how to handle reliability in REST at the moment.

8.8.2 Explicit Acknowledgement

We have to be aware that reliable messaging and explicit acknowledgements are not mutual exclusive. The former confirms the arrival of messages, the latter the correct processing. A basic explicit acknowledgement message would again be a notification in the event architecture containing an acknowledgement event. Even in a simple implementation this event would be a complex event as it always has a cause, namely the event that is acknowledged. If more information shall be available the second cause for the acknowledgement, the action that is (to be) taken, can also be referenced as a member. Figure 50 shows an exemplary event taxonomy including acknowledgements.

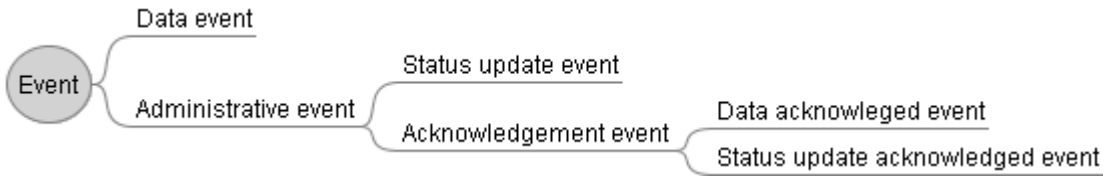

Figure 50 - Exemplary event taxonomy including acknowledgements

To deliver acknowledgement notifications one can in general think of two possible scenarios. On the one hand one can deliver them automatically to the sources of the initial notification. In this case the acknowledgement provider would be an instantiation of a Publisher deriving the target of the acknowledgement notifications from a received notification. On the other hand the acknowledgement provider could be implemented as an instantiation of a Producer allowing to subscribe for acknowledgement notifications. This would make it easier to deliver acknowledgements from one end to another in a chain of components because they don't have to be forwarded through the whole chain. This also reduces network traffic and the danger of the loss of such a notification since every intermediate component is a possible single point of failure for the delivery. Note that the automatic acknowledgement described in the option is still possible in the second one since Producers not necessarily enforce subscriptions.

In addition to modeling explicit acknowledgements as events, they can of course also be implemented via a kind of acknowledgement operation (with request and response) that is specifically invoked by the acknowledging entity, usually by a human operator or responsive personnel.

Usage of specific acknowledgement data types in messages could also signify that a message (given by the current context) is explicitly acknowledged.

8.9 Graceful Stopping and Rollback of Events

As described in chapter 6.3 an event or at least its genuine properties do not change. This is especially true for events but not for the event objects representing these events. A representation is built as the result of a process (e.g. measurements, unsupervised processing) which may be the source of errors. In such cases the event object including its genuine properties has to be changed or updated. This does of course not affect the event itself which does not change in case of measurement errors. In general, an update or change can have multiple reasons. An event object may for instance only be valid until a given time when new or more accurate data is available for the event.

Such a change would have to be performed on every copy of the event object which is very difficult in a distributed system. Because an event source does not have control over the distribution of its events once they have left the system boundary, the only way to change an event is to publish an updated event that references the old event as explained in chapter 6.3.4.2.1.

There is another way to indicate that an event object is only temporary. Think of an automated processing application which publishes non- validated results immediately

after processing. The application adds a "valid until" property to the event object which deprecates the event object automatically once the time has been reached. If the event object has been validated in the meantime, the validation information can be added to a new event that references the non-validated event. This way – though it might look a bit complex on first sight – ensures that all event sinks can update their internal data model with the new information (that the event has been validated).

If the older event objects do not automatically expire they have to be stopped and/or deprecated. Stopping means that they are not further published. In order for this mechanism to be effective the stopping events need to be processed with a higher priority in order to reach all affected publishers. Furthermore, the time between the creation of the initial and the stopping event should be short. Otherwise the chances to stop the distribution of an event in the system might be very low.

When deprecating an event object all recipients of the event have to be informed. This can either be done by using an explicit rollback event which only deprecates an event object or by publishing a new event object which overrides the old one. In the latter case the deprecated object should be referenced (using event-event relationships, see chapter 6.5). In the former case consumers have to be aware of the possibility of rollback events and ensure that they receive them if published.

When designing an event driven system one has to decide which possibilities of stopping / rollback shall be used. This can vary for every event type of the systems event causality model (see chapter 8.5). Furthermore the additional events have to be integrated in the causality model. It has to be defined if all event objects that are derived from a deprecated event object will be deprecated immediately or stay valid. In a system this behavior may differ for different event types but also for different components because every consumer of a stopping or rollback event has to process and possibly forward them on its own. In order to avoid inconsistent event object spread through the whole system stopping and rollback has to be defined carefully. This can also be accompanied by the definition of special channels and topics for stopping and rollback events.

Policies could be used to indicate the behavior executed by a given system component when faced with changes to already processed events.

Furthermore the stopping and rollback events have to be distinguished from events that mark the end of something that happens such as "all clear" events. The latter do not contain changes to an event object and also do not deprecate them. They represent the change of the state of an object or process and therefore provide new knowledge. The event objects representing the previous state stay valid.

9 Conclusion

This report defines the elements of a common OGC Event Architecture.

The first section of the report defined the abstract architecture. First of all, the term event has been defined for the OGC domain. The definition is based upon an investigation of existing event definitions from various fields. The OGC event definition has been related to the OGC baseline, especially the General Feature Model. The specific property characteristics of an event seen as a feature are discussed in detail. Also, a differentiation of the term event to related terms like alert is provided.

The next part of the first section introduces an OGC Event Application Schema, which defines an extensible model and encoding for events, and elaborates on event hierarchies and taxonomies.

After this, fundamental interfaces and service roles are defined. In Annex A, these interfaces are mapped to a SOAP / WS-* implementation binding based upon OASIS standards.

In the second section, the interfaces and roles defined for the abstract architecture are mapped to OGC services in various use cases. Multiple examples on how to apply the event architecture roles are provided, using almost all standard OGC services. The examples also present various event taxonomies which give an idea of the possible information gain when event-enabling the well-known web services.

Technologies related to the Event Architecture have been discussed in the third section. This encompasses messaging patterns like datagram or publish/subscribe messaging – including an overview of different subscription models – and aspects of asynchronous communication. Furthermore, Topics like Event Driven Architecture, Enterprise Service Bus, Event Processing, various aspects of event times, event transport, reliable messaging, event acknowledgement as well as event stopping and rollback have been presented and discussed.

The OGC Event Architecture promises a considerable improvement of information availability, accessibility and distribution. Tight service coupling can be diminished and loose coupling be achieved more easily.

The architecture provides a consistent approach for performing event based communication and processing. OGC service specifications that want to leverage the functionality of the event architecture will first need to define the event types that are relevant for them. This will result in event type hierarchies that might very well be shared (at least partly) across domains. The design of event types is closely related to the definition of feature types. In addition, design of topic namespaces and association of event types to certain topics is an opportunity for service specifications to define addition notification semantics. Usage of topic based subscription has advantages for both clients and services.

Having defined event types and topics, required processing functionality will need to be defined. This report provides an overview of various aspects when processing of events is concerned. This encompasses event processing techniques, filtering, event transformation, transport protocol switching etc. Not all of this functionality is yet covered by standards. Therefore, existing service or encoding specifications like the Web Notification Service and the Event Pattern Markup Language will need to be improved in the future. Projects and Testbeds will show which processing functionality is required and will hopefully lead to evolution of existing specifications as well as introduction of new specifications that implement missing functionality not yet covered by any other standard.

10 Annex A: WS-N Tutorial (informative)

10.1 Publish/Subscribe using WS-Notification

The mostly used communication pattern in Service Oriented Architectures is the request-response pattern. As the demand for more flexible and dynamic services increases, other communication patterns are put in place. One of them is executed in Publish/Subscribe scenarios, where a client subscribes to receive notifications that match given criteria (see clause 8.1.3). Pub/Sub effectively introduces a decoupling between the notification publisher and subscriber.

In the WS-* world, Pub/Sub can be implemented with the WS-Notification standard (OASIS 2006a). Another approach from the W3C exists, called WS-Eventing[19]. This standard is not a W3C recommendation yet; however, work is underway at W3C to bring WS-Eventing to recommendation status[20]. The current version of WS-Eventing has a limited set of functionality when compared to WS-Notification (called WS-N in the remainder of this Annex). However, extension points exist that could be used to add missing functionality. Until WS-Eventing is not a final standard, we suggest to use WS-N for enabling Pub/Sub functionality in all WS-* (sometimes also called SOAP-) bindings / architectural styles in OGC specifications. WS-N, being an approved OASIS standard since 2006, has slowly made its way into the web services world, in standards organizations like the Open Grid Forum (see for example OGF [2006]) and also several implementations (though usually not with the full set of functionality)[21].

Niblett and Graham (2005) provide a high level overview of the WS-N functionality, and this paper is usually referenced when referring to WS-N. However, it does not provide a detailed tutorial on how to implement WS-N in a web service or a client. In this Annex, we will try to provide such urgently needed guidance. Specific examples are illustrated using the OGC Sensor Planning Service (SPS).

This tutorial will provide a quick introduction into WS-Addressing (called WSA in the remainder of this Annex) and the Web Services Resource Framework (from now on called WSRF in this Annex). The former is required technology when using WS-N while the latter is not, though we recommend its use. SOAP and WSDL are also not required by WS-N[22], but the standards are usually used in conjunction. We therefore recommend their combined usage. Other WS-* standards could also be implemented by a web service, like WS-ReliableMessaging for ensuring that messages sent by a communication endpoint are

---

[19] Harmonization of the OASIS and W3C specifications was intended. Unfortunately, these efforts have ceased.

[20] see http://www.w3.org/2002/ws/ra/

[21] Apache MUSE: http://ws.apache.org/muse/, Apache ServiceMix: http://servicemix.apache.org, GLOBUS toolkit: http://www.globus.org/toolkit/

[22] WS-Notification is designed to be independent of a specific transport binding like SOAP, so can in theory also be used directly. Usually, though, SOAP is used as the transport protocol. A project or testbed would be required to prove that WS-Notification can really be applied directly, e.g. via HTTP.

not lost in transit. This shows that functionality can be achieved by composing the WS-* building blocks in one service interface.

10.2 Essentials of WS-Addressing

How to address a web service in the WS-* world is well known. WSA is usually applied to provide detailed information about a communication endpoint that resides in the distributed service network. Many tutorials exist on the Web for WSA, so our explanations will be brief.

WS-N uses WSA to identify entities that produce, broker and/or consume notifications, manage a subscription etc. Operation requests can be sent to these so called endpoints. Figure 51 shows the structure of a WSA endpoint.

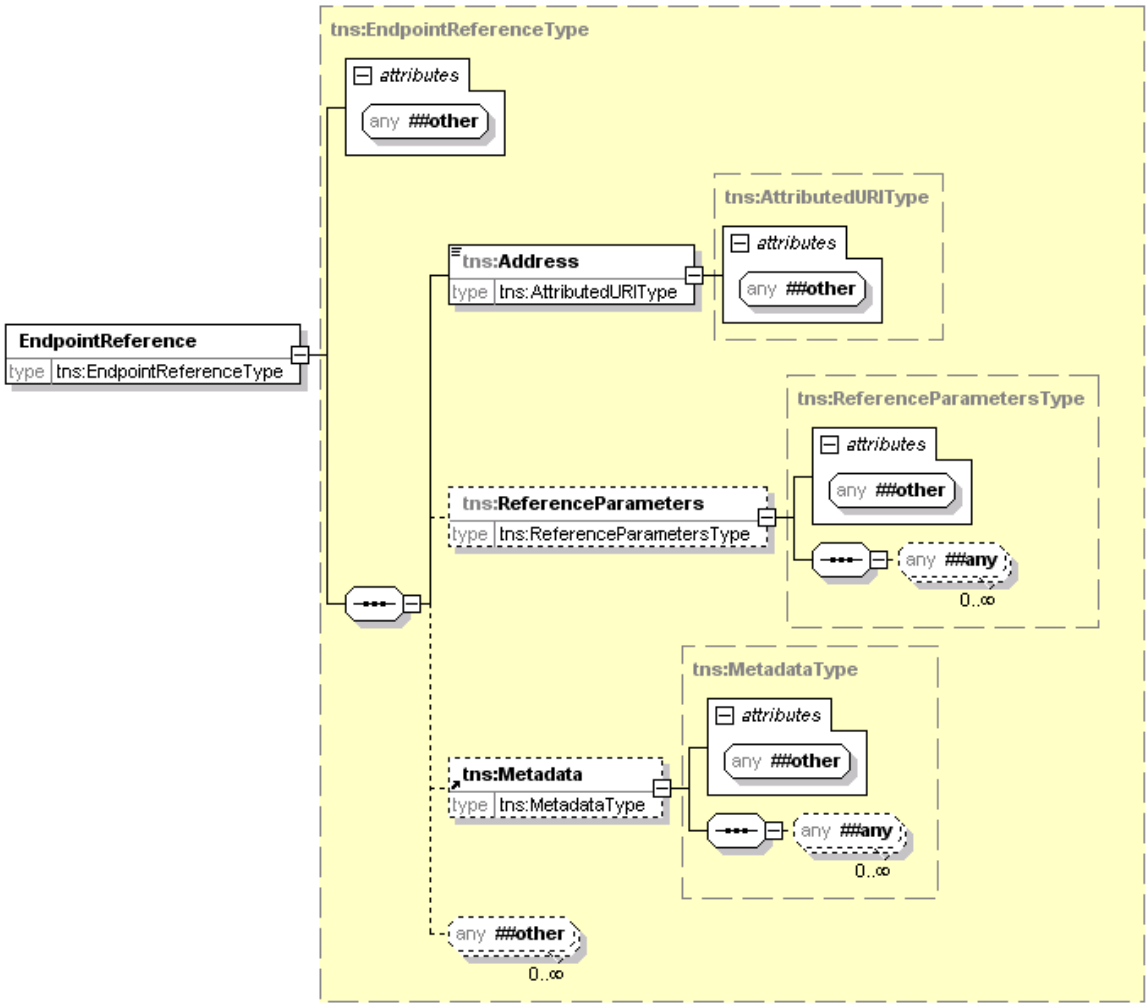

Figure 51 - WS-Addressing EndpointReference in XMLSpy notation

Either the EndpointReference element is used to convey endpoint information in operation requests and responses (and also in resource properties) or an XML element of

type EndpointReferenceType with similar semantics. Only an address (of type URI) is required. In addition, reference parameters and metadata may be provided. The typical extension elements and attributes are available, allowing implementers to add application specific information where they see fit. While metadata shall not be of concern here, a closer look at the reference parameters is advisable, because they are an important means to differentiate specific endpoints that have a common (service) address.

Reference parameters are a list of XML elements from a given namespace. A request to an endpoint described via WSA will use the information from the endpoint reference and add it to the header part of the SOAP message. Have a look at the following figure.

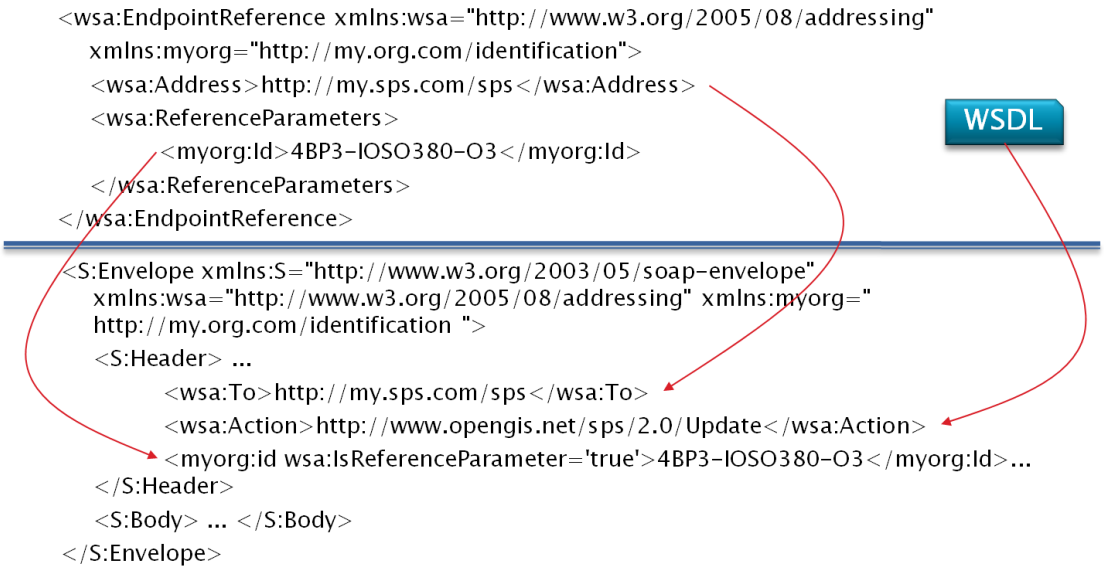

Figure 52 - Handling of WS-Addressing MessageProperties

On the top in Figure 52 one can see endpoint information encoded via the WSA EndpointReference element. The service address is “http://my.sps.com/sps”. However, the final communication endpoint where the SOAP message shown in the lower part shall be delivered to is identified via the additional myorg:Id reference parameter of value “4BP3-IOSO380-O3”. An application that wants to send a message to such an endpoint will add the wsa:To element to the header, with the URI given in the wsa:Address of the endpoint. Furthermore it will add each reference parameter element from the endpoint AS IS to the header - with the only addition of the wsa:IsReferenceParameter attribute to this XML element. When set to true, the attribute will inform message processors that the given XML element (which can also have other child elements) is a reference parameter. In the case of an SPS, multiple tasks that are managed by one service could be differentiated in this way. For the message to be meaningful for the recipient, only the wsa:Action parameter needs to be added. This element identifies the message semantics and is provided for each operation part in the WSDL description of the service. Additional properties one might encounter in an endpoint reference and SOAP header are shown in Listing 1.

Listing 1 - XML Infoset Representation of Message Addressing Properties (W3C, 2006)

```
<wsa:To>xs:anyURI</wsa:To> ?
<wsa:From>wsa:EndpointReferenceType</wsa:From> ?
<wsa:ReplyTo>wsa:EndpointReferenceType</wsa:ReplyTo> ?
<wsa:FaultTo>wsa:EndpointReferenceType</wsa:FaultTo> ?
<wsa:Action>xs:anyURI</wsa:Action>
<wsa:MessageID>xs:anyURI</wsa:MessageID> ?
<wsa:RelatesTo RelationshipType="xs:anyURI"?>xs:anyURI</wsa:RelatesTo> *
<wsa:ReferenceParameters>xs:any*</wsa:ReferenceParameters> ?
```

The ReplyTo should be added to the SOAP header of all operation requests that expect a response of some kind (not shown in Figure 52). It describes an endpoint where the response from the service identified via the wsa:To and wsa:ReferenceParameters should be sent to. This mechanism enables asynchronous communication as explained in chapter 8.2. In the same way, fault messages can be directed to a specific endpoint.

Asynchronous communication should be expected whenever a SOAP message contains a ReplyTo and / or FaultTo header element. However, WSA defines two URIs that have special meaning when used by the client: "http://www.w3.org/2005/08/addressing/anonymous" (from now on called *anonymous-URI*) and "http://www.w3.org/2005/08/addressing/none" (from now on called *none-URI*). When the anonymous-URI is used in the ReplyTo and / or FaultTo elements, the client cannot provide a meaningful URI for the service to send the response or fault to asynchronously. WS-I (2007) defines that in such a case the response shall be sent in the entity body of the HTTP response message, i.e. synchronously. This would mean that a client can decide whether the response should be delivered synchronously or asynchronously. However, the service may impose a policy that prohibits the use of anonymous-URIs by adding a policy statement to its service description, thus the service can still be in charge of deciding whether asynchronous communication is required or not (see Annex B: WSDL Example of a Service using WS-Notification (informative)). What does the none-URI mean, then? Clients use this URI to indicate that no reply or fault message should be sent. This simply means that the service may discard any response that would usually be generated and sent to the client.

A sender that expects a response should also add a wsa:MessageID with unique URI to the header elements so that it can associate a subsequent response (sent asynchronously and containing the same URI in a wsa:RelatesTo element) to the request message. A sender may also identify itself by adding a wsa:From element to the header of request messages.

Sometimes one will encounter the notion of a one-way, request-response or other message exchange pattern. These patterns are defined by W3C (2007a, 2007b, 2007c) with respect to WSDL 1.1 and 2.0.

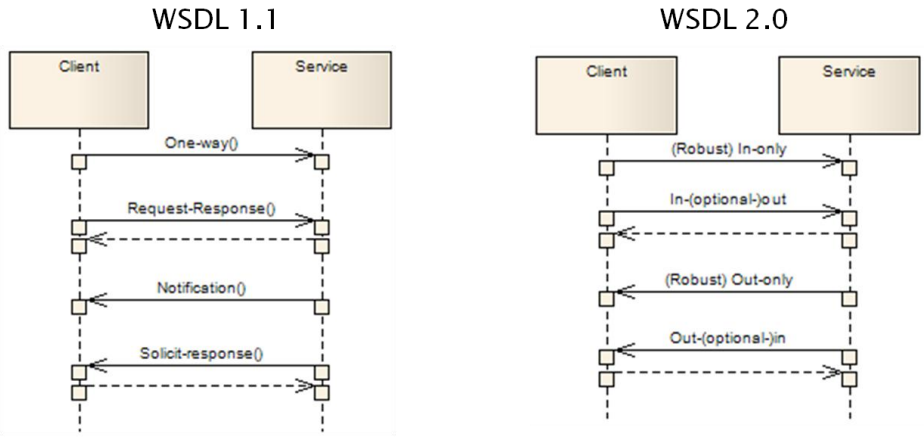

Figure 53 - Message Exchange Patterns described by WS-Addressing

All possible patterns are shown in Figure 53. Note the similarity of the patterns. Only the naming differs. In addition, WSDL 2.0 introduces the notion of robust in-only and out-only. Robust in this case means that no response is expected; only if an exception occurs a fault should be sent so that the sending entity is aware of the failed communication. WSA associated different requirements concerning the usage of the message addressing properties shown in Listing 1 with each pattern. Figure 54 provides a summary of the information that is explained in more detail in W3C (2007a, 2007b, 2007c).

			destination (wsa:To)	action (wsa:Action)	source endpoint (wsa:From)	reply endpoint (wsa:ReplyTo)	fault endpoint (wsa:FaultTo)	message id (wsa:MessageID)	relationship (wsa:RelatesTo)
WSDL 1.1	one-way / notification	request	Y	Y	N	N	N	N	N
	request-response / solicit-response	request	Y	Y	N	Y	N	Y	N
		response	Y	Y	N	N	N	N	Y
WSDL 2.0	in-only / out-only	in message	Y	Y	N	N	N	N	N
	robust in-only / robust out-only	in message	Y	Y	N	reply and / or fault		Y	N
		fault message	Y	Y	N	N	N	N	Y
	in-out / out-in / in-optional-out / out-optional-in	in message	Y	Y	N	Y	N	Y	N
		out message	Y	Y	N	N	N	N	Y
	exchange pattern	**exchange part**	**message addressing properties (mandatory = Yes, optional = No)**						

Figure 54 - WS-Addressing and WSDL Message Exchange Patterns

Together, the patterns shown in Figure 53 make up the basic message exchange patterns from which more sophisticated patterns like the publish/subscribe pattern can be created.

In the pub/sub pattern one would have one request-response (in-out) message exchange for subscribing and multiple one-way (in-only) messages for notifying the subscribed consumer. Note that WS-N neither expects a reply nor a fault when sending a notification to the consumer, thus it is a simple fire-and-forget message as we will see later on. Also note that the Web Services Interoperability Organization (WS-I) Basic Profile 1.1 prohibits use of the notification and solicit-response patterns (WS-I, 2006)[23].

10.3 Essentials of the WS-ResourceFramework

According to OASIS, "a Web Service Resource is the composition of a resource and a Web service through which the resource can be accessed" (OASIS, 2006b), see Figure 55. Furthermore, a WS-Resource is identified by (at least) one WS-Addressing endpoint, has zero or (usually) more accessible properties encoded in XML and may have a lifetime.

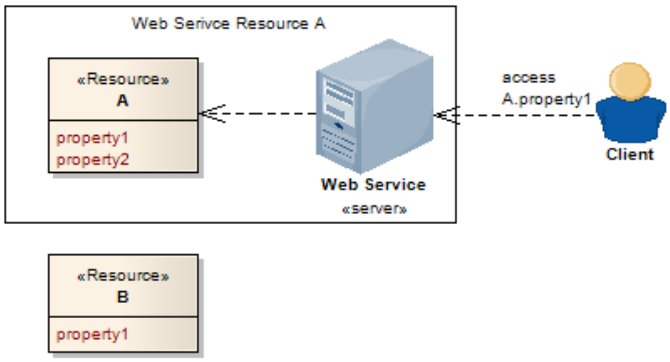

Figure 55 - Accessing a Web Service Resource

So in fact, this does not mean anything else than to access an object, a resource, by invoking operations. This is much like OO-programming, where a class has a set of attributes and implements a set of operations. The difference to a "normal" web service described in WSDL is that a WS-Resource does not only define its interface in WSDL but also provides an XML schema based description of how its properties are structured.

The Web Service Resource Framework provides specifications that define how to express and access WS-Resource properties (OASIS, 2006c), the encoding of a basic fault / exception message (OASIS, 2006d) and how lifetime of resources can be handled in a common way (OASIS, 2006e). In addition, it also provides a specification that deals with the management of web service groups (OASIS, 2006f), but that is not of interest at this point, so we will skip it. Documentation that should also be noted are the application notes provided by OASIS for implementers of the Web Service Resource Framework (OASIS, 2006g). It provides best practices and scenarios that explain how to handle some possible issues implementers might encounter.

---

[23] This is one reason why the notification (out-only) pattern is not applied for notifying a consumer. The real reason is that, as we will see later, performing a notification is in fact doing a (one-way) request at the consumer endpoint.

Now, before we explain the essentials, one word to the "bunch of specifications" we already mentioned: for a simple use case one does not need them all, one can incrementally implement them if wanted and by using these standards one can reuse software also for other use cases. Rather than having one big, all-encompassing specification, one has different, well separated specifications. For sure, some have dependencies on other specifications, but there is a common baseline for enabling simple publish/subscribe scenarios and from which to start when enabling pub/sub.

In the remainder of this section, whenever we talk of a resource, we mean a WS-Resource, unless otherwise stated.

10.3.1 Resource Properties

A resource offers one or more publicly accessible properties (security considerations are a different topic). They are contained in the so called resource properties document. This document is represented by an XML element which needs to be defined, either generally (like in the schema of an OWS), or in the WSDL description of a specific service. The final port type which is implemented by the given service will then have an attribute that references this XML element, see Listing 2.

Listing 2 - indicating the resource properties document structure in a service description

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://www.igsi.eu/sps"
xmlns:igsi="http://www.igsi.eu/sps" xmlns:sps="http://www.opengis.net/sps/2.0"
...>
  <wsdl:types>
  ...
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.igsi.eu/sps">
      <xsd:element name="ServiceResourceProperties">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="wsn-b:FixedTopicSet"/>
            <xsd:element ref="wsn-t:TopicSet" minOccurs="0"/>
            <xsd:element ref="wsn-b:TopicExpression" minOccurs="0"
maxOccurs="unbounded"/>
            <xsd:element ref="wsn-b:TopicExpressionDialect" minOccurs="0"
maxOccurs="unbounded"/>
            <!-- service specific properties could be added, like the service
capabilities -->
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      ...
    </xsd:schema>
  </wsdl:types>
  ...
  <wsdl:portType name="SpsPortType" wsrf-
rp:ResourceProperties="igsi:ServiceResourceProperties">
  ...
  </wsdl:portType>
  <wsdl:binding name="SpsBinding" type="igsi:SpsPortType">...</wsdl:binding>
  <wsdl:service name="igsiSpsService">
    <wsdl:port name="igsiSpsPort" binding="igsi:SpsBinding">...</wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```


This 'reference' only provides the structural definition of the resource property document. To actually retrieve one of the properties (direct child elements of the resource property document element, like the wsn-t:TopicSet in Listing 2), one would send a GetResourceProperty request to the resource (remember that we are talking about a web service(!) resource). This is the only mandatory operation defined by the WS-ResourceProperties specification. Listing 3 shows an exemplary request to retrieve the TopicSet property of the service.

Listing 3 - GetResourceProperty request example

```
<s12:Envelope ...>
  <s12:Header>
    <wsa:To>http://www.igsi.eu:8080/services/sps</wsa:To>
    <wsa:Action>http://docs.oasis-open.org/wsrf/rpw-
2/GetResourceProperty/GetResourcePropertyRequest</wsa:Action>
    <wsa:ReplyTo>http://my.client.com/client/sps</wsa:ReplyTo>
    <wsa:MessageID>msg-0010</wsa:MessageID>
  </s12:Header>
  <s12:Body>
    <wsrf-rp:GetResourceProperty>wsn-t:TopicSet</wsrf-rp:GetResourceProperty>
  </s12:Body>
</s12:Envelope>
```

With this operation, one requests the value of one specific resource property. The response might look something like shown in Listing 4.

Listing 4 - GetResourceProperty response example

```
<?xml version="1.0" encoding="UTF-8"?>
<s12:Envelope ...>
  <s12:Header>
    <wsa:To>http://my.client.com/client/sps</wsa:To>
    <wsa:Action>http://docs.oasis-open.org/wsrf/rpw-
2/GetResourceProperty/GetResourcePropertyResponse</wsa:Action>
    <wsa:RelatesTo>http://my.client.com/uid/msg-0010</wsa:RelatesTo>
  </s12:Header>
  <s12:Body>
    <wsrf-rp:GetResourcePropertyResponse>
    <wstop:TopicSet>...</wstop:TopicSet>
    </wsrf-rp:GetResourcePropertyResponse>
  </s12:Body>
</s12:Envelope>
```

In addition to the mandatory GetResourceProperty operation, the WS-ResourceProperties specification defines several optional operations that are shown in Figure 56.

Figure 56 - Operations defined by WS-ResourceProperties

A service that wants to make use of WS-ResourceProperties may implement as many of these optional operations as it wants to. In fact, these operations represent convenience operations for accessing the whole resource properties document or multiple resource properties at once. Operations also exist for querying, modifying and deleting resource properties. Because these operations are not required when implementing WS-Notification, they will not be explained in more detail[24].

10.3.2 Exception Handling

When a service encounters an exception while handling an incoming request, it usually provides information describing the exception to the client[25]. The SOAP specification defines a basic fault encoding. The actual encoding is determined by the SOAP version in use (1.1 or 1.2). The available information is quite high-level and is designed to convey fault information related to the SOAP message transport. However, both encodings allow the inclusion of application specific exception details. This is where WS-BaseFaults comes into play. It defines a common fault structure that can be extended by specifications like WS-Notification to convey their specific exception information. The schema of the base fault type is shown in Figure 57.

---

[24] OASIS also published a committee draft specification for defining a metadata descriptor format, the WS-ResourceMetadataDescriptor (OASIS, 2006h). This metadata descriptor can be added to the WSDL description of a specific WS-Resource in order to define the mutability (constant, appendable, etc.), modifiability (read-only, read-write, etc.), possible values etc of each property the resource has. This allows a client to get detailed information about which operations are supported per resource property. Without this information, services might respond with a fault when the client tries to perform an invalid modification (like setting a wrong value or writing to a read-only property). WS-ResourceMedataDescriptor has been implemented by the Apache MUSE project.

[25] As explained for WS-Addressing, it depends upon the actual message exchange pattern of the invoked operation whether a fault / exception is transmitted or not.

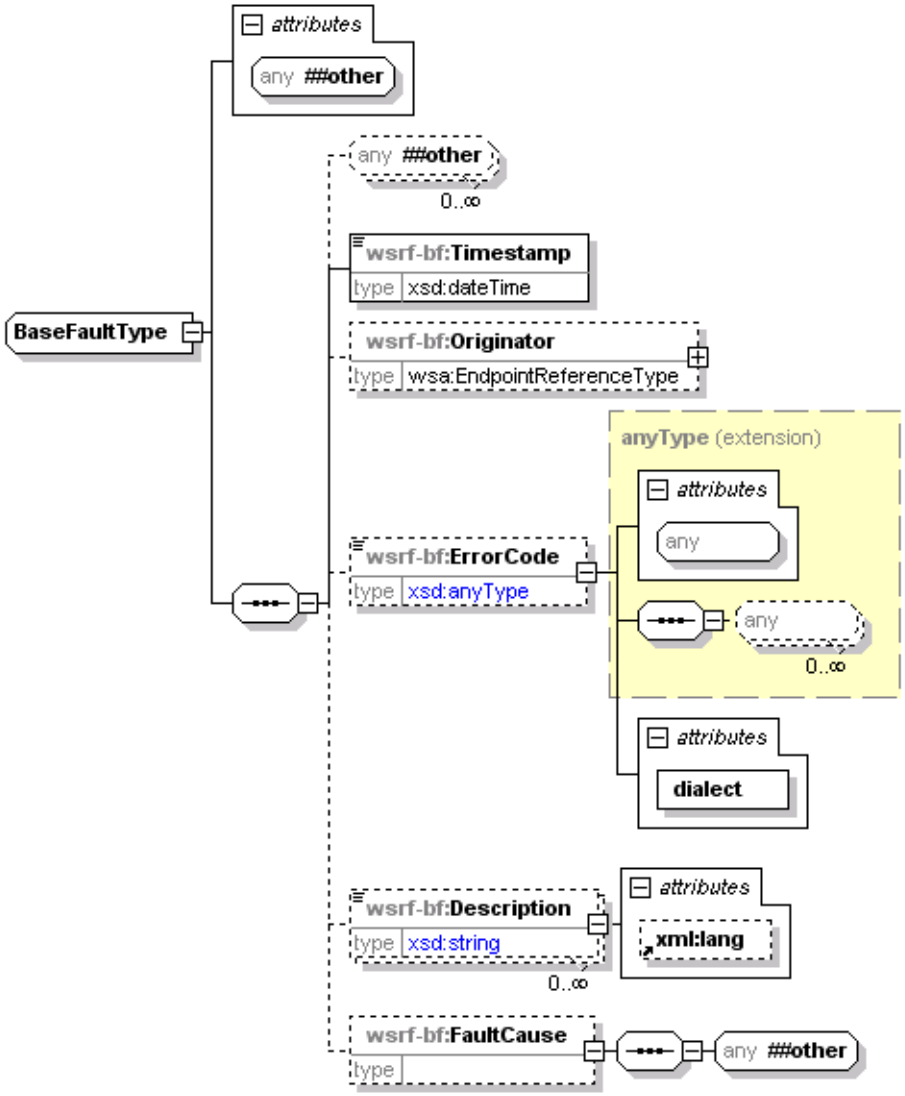

Figure 57 - WS-BaseFaults BaseFaultType in XMLSpy notation

The only mandatory information is the timestamp when the fault occurred. In addition, the fault originator may be indicated via a WS-Addressing endpoint reference. This is especially useful in the situation of nested faults, which may be created by adding causing faults in the according FaultCause element. An error code and descriptions may also be provided. As usual, one has multiple extension points where one may add own attributes and elements. The common case, however, is to extend the BaseFaultType and thus define specific fault element in XML Schema.

We introduced the GetResourceProperty operation before. This operation defines three possible fault conditions, one of which is that the resource where the request was sent to is unknown - see Listing 5.

Listing 5 - Example of a ResourceUnknownFault with SOAP 1.2

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="http://www.w3.org/2003/05/soap-envelope" ...>
  <Header>
    <wsa:Action>http://docs.oasis-open.org/wsrf/fault</wsa:Action>
    ...
  </Header>
  <Body>
    <Fault>
      <Code>
        <Value>Sender</Value>
      </Code>
      <Reason>
        <Text xml:lang="en">No such resource exists</Text>
      </Reason>
      <Detail>
        <wsrf-r:ResourceUnknownFault>
          <wsrf-bf:Timestamp>2009-01-04T12:42:12Z</wsrf-bf:Timestamp>
          <wsrf-bf:Description>Resource unknown</wsrf-bf:Description>
        </wsrf-r:ResourceUnknownFault>
      </Detail>
    </Fault>
  </Body>
</Envelope>
```

Here, the application specific fault is included in a SOAP 1.2 message / fault. If the SOAP binding is not used, then the ResourceUnknownFault would still be used, either as is or by including it inside the binding specific container.

Right now there is no OGC guidance on how to perform exception / fault reporting for OGC service operations when using the SOAP binding. Schäffer (2008) mentions the description of fault handling as a future work item. This report will not go into further details on this problematic, either. However, it can be noted that an exception as currently defined by OWS Common can likely be mapped to a base fault or be an extension of it. When using WS-Notification operations, however, one will deal with faults that comply with the WS-BaseFaults specification.

10.3.3 Resource Lifetime

At times, a resource may have a defined lifetime. This means that the resource is available as long as the end of the lifetime has not been reached. A reservation is an example, as we will see later a registration and subscription are other candidates. The WS-ResourceLifetime specification defines operations for destroying resources and for resetting the termination time of a resource. In addition, it defines two resource properties that can be used to indicate a resource's termination time and the current server time[26].

These operations and resources can be used in the following ways:

- The Destroy operation can be used by a client to immediately destroy the resource that implements this operation. This is independent of the availability of a

[26] In which the termination time is expressed; this is used to mitigate clock synchronization issues.

termination time for the resource. Thus, even if no termination time was set and the resource could live indefinitely, the client may destroy it at any time.

– The resource has a finite lifetime, i.e. will be destroyed automatically once its termination time has passed. This requires the resource to incorporate the current and termination time resource properties (which in this simple case will be set by the service).

– If the SetTerminationTime operation is implemented by the resource, then a client is allowed to reset the termination time resource property (both the termination and current time properties are mandatory for a resource implementing this operation).

Note that the service is the authority to set a termination time (or not) and to allow immediate destruction and resetting of the termination time. So even if the operations are implemented by the resource, the service might refuse to perform them (and respond with corresponding faults) for various reasons.

10.4 WS-Notification

With WS-Notification (WS-N) one has an enterprise grade, standards based solution for integrating Publish/Subscribe functionality in web services. Messages are usually posted on topics, to which a client can subscribe. The push-based publication of notifications is the default use case. However, also a pull-based approach is possible via so called PullPoints (more on that later). In addition to basic Pub/Sub where the publisher is defined by the Pub/Sub service, a brokering solution is defined. Here, the web service acts as a general purpose Pub/Sub service, to which new publishers may be registered by clients.

WS-N consists of three separate specifications: WS-BaseNotification (WSN-B), WS-Topics (WSN-T) and WS-BrokeredNotification (WSN-BR). The dependencies of these three specifications as well as the dependencies to other WS-* specifications will be explained in the following chapter. After that, we will introduce each of the three WSN specifications in more detail.

10.4.1 Specification Dependencies

WS-Notification has the following dependencies on other standards:

– WS-Addressing: required to encode communication endpoint information.

– Web Service Resource Framework: only WS-BaseFaults is required, the other specifications are optional. However, we recommend using the other WSRF specifications that we introduced in this Annex as well.

– SOAP: WSN-B is transport binding independent. Thus, the interface should be applicable to the well-known Plain-Old-XML via HTTP style of implementing

web services[27]. In most existing implementations of WS-N, SOAP (either version 1.1 or 1.2) is used as the underlying binding.

The dependencies between the three packages defined by WS-Notification are as shown in Figure 58.

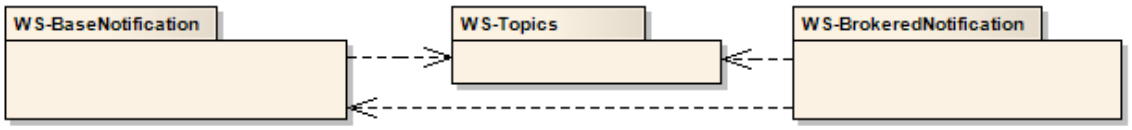

Figure 58 - Package dependencies in WS-Notification

WS-BaseNotification (WSN-B) only depends upon WS-Topics (WSN-T). However, this dependency does not represent a requirement. WSN-B can be implemented without making use of topic based subscriptions. This supports very basic use cases, when for example subscribers can only subscribe for each notification published by a producer.

WSN-B is extended by WS-BrokeredNotification (WSN-BR), which adds broker functionality to the Pub/Sub interface defined by WSN-B. WSN-BR thus depends upon WSN-B however it is not required to implement WSN-T. Only in more complex use cases should topics be used. This is also the case whenever a certain subscription model other than content based filters is of interest and whenever the available event types shall be made explicitly visible.

So let us have a closer look at WS-Topics.

10.4.2 Modeling Notification Topics with WS-Topics

10.4.2.1 Topic Namespace

The first construct one needs to know is the so called topic namespace. This construct represents a hierarchy of notification topics that belong to a given namespace and have semantics as defined for that namespace, see Figure 59.

[27] This should be proved via an interoperability experiment or in an OGC testbed.

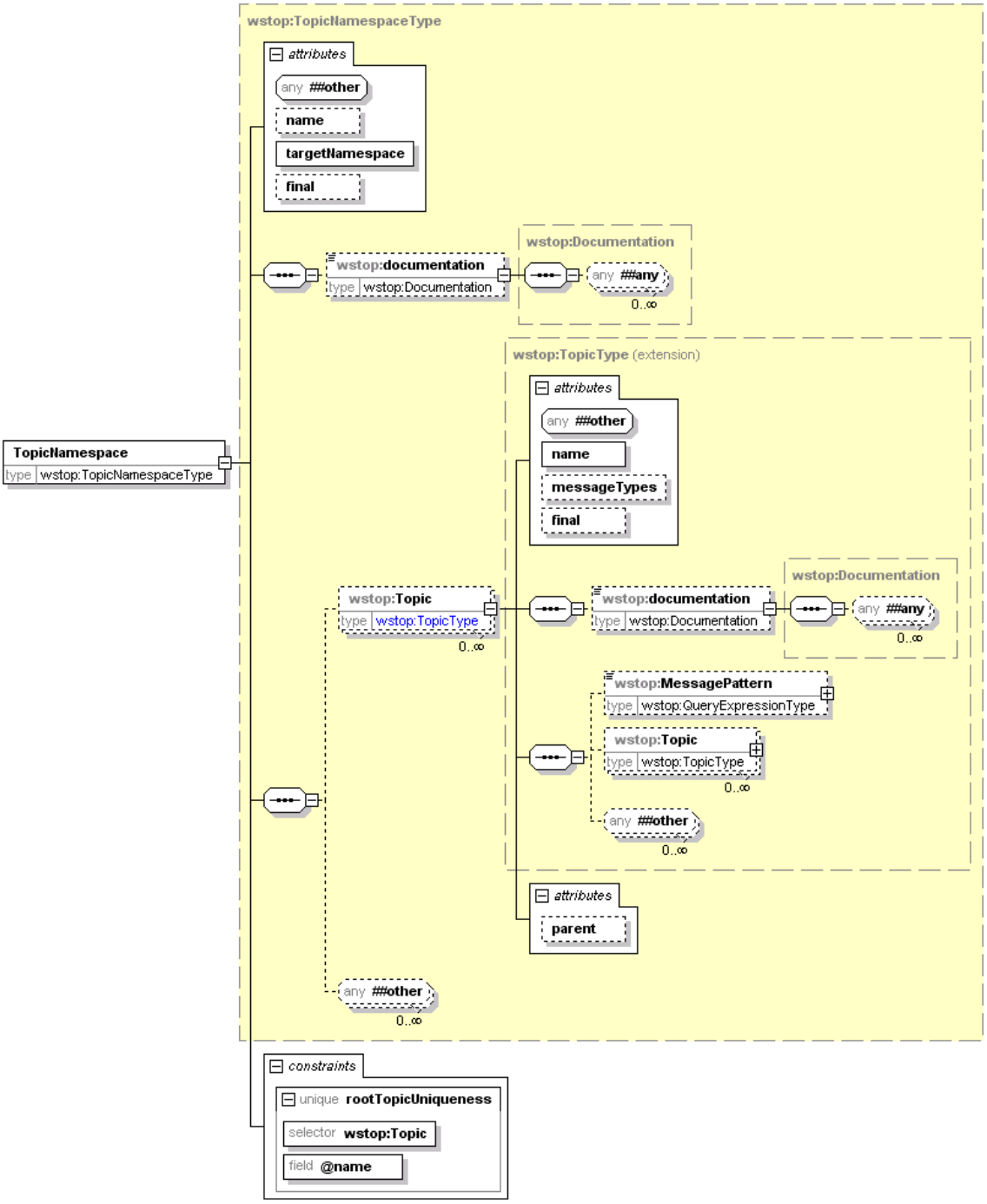

Figure 59 - TopicNamespace element as defined by WS-Topics, in XMLSpy notation

A topic namespace belongs to a given *target namespace* (e.g. http://www.opengis.net/sps/2.0). It may have a *name* and it may be flagged as *final*, meaning that the topic namespace cannot be extended. To clarify: this does not mean that the topics defined in the namespace may not be extended by extension topics (which will be explained later on). It rather means that no new topics with the same target namespace

may be defined later on, whether in another or the same XML document that defines the final topic namespace.

EXAMPLE: if topic namespace A with target namespace A.TN defined three topics, A.1, A.2, A.3 and is marked as final, then no other topic namespace B with the same target namespace A.TN may ever exist that defines a different set of topics. However, a topic namespace C with target namespace C.TN may exist that defines a topic C.1 which can be an extension topic of A.1, A.2 or A.3.

A topic namespace may contain any kind of XML encoded *documentation*. This is one of the available extension points where for example ISO metadata could be added.

The most important information provided by a topic namespace is of course the list of *topics* that belong to that namespace. This list can be empty, which supports use cases in which topics are added dynamically to the topic namespace (of course that topic namespace may then not be final). If the list is not empty, it contains a list of topics, each with a distinct (required) name.

Topics may themselves contain a list of (unique) child-topics, thus forming a topic hierarchy. Each topic has a *name* (of type NCName). Whenever it is known that only messages from a specific set of message types is published on a given topic, these types (expressed as QNames) may be listed in the *messageTypes* attribute of the topic. When such a set of message types is not known, the messageTypes attribute can simply be omitted, meaning that any message type may be published on the topic (within the semantic constraints of that topic). If the topic is *final*, then no extension topic can be defined for it. An extension topic is a root topic of a topic namespace (in the same or other target namespace) which identifies a certain topic from another topic namespace as its parent. Such a topic would then be a child-topic of the parent topic in the topic hierarchy. Extension topics support use cases in which a specification might only define a list of required high-level topics. Specific service implementation can then add more specific sub-topics to these high-level topics, allowing more fine-grained subscriptions. A topic may contain *documentation*, just as a topic namespace. It may also contain a *message pattern*, which defines the content of notifications posted on the topic in more detail. For example, if a NewSensor topic defined the general SensorManagementNotification as the only possible message type of published notifications, it could further constrain those by adding a query expression like "managementCode=newSensor", which evaluates to true for each notification published on that topic. The dialect of the query expression is up to the implementation, for example be an OGC filter expression. WSN-T suggests usage of XPath 1.0, however each implementation is free to use the dialect that best fits.

The topic namespace with an empty target namespace is called the *ad-hoc topic namespace*. This namespace is non-final and can be used by implementations to add new topics to their topic set (explained later on) in an ad-hoc way. However, such ad-hoc topics have "ad-hoc semantics" - clients should not expect the same semantics for an ad-hoc topic that is defined by two different services.

Topics can be formed into hierarchies, thus allowing clients to subscribe to whole subtrees with only one subscription. Content based filtering is also possible. The approach enables execution of the various subscription models introduced in chapter

8.1.3.1. Which subscription model or combination thereof is possible depends upon the semantics that are associated with a given topic.

Let us have a look at some examples. We introduce three topic namespaces: one defining topics applicable to all Sensor Web services (see Listing 6), one defining SPS topics (see Listing 7) and one defining topics that are specific for a given service implementation (see Listing 8).

Listing 6 - SWE Common Service Topic Namespace example

```
<?xml version="1.0" encoding="UTF-8"?>
<wstop:TopicNamespace targetNamespace="http://www.opengis.net/swes/1.0"
xmlns:wstop="http://docs.oasis-open.org/wsn/t-1"
xmlns:swes="http://www.opengis.net/swes/1.0" final="true">
  <wstop:Topic name="ServiceMetadataChanged">
    <wstop:Topic name="OfferingAdded" messageTypes="swes:OfferingReport"/>
    <wstop:Topic name="OfferingRemoved" messageTypes="swes:OfferingReport"/>
    <wstop:Topic name="OfferingChanged" messageTypes="swes:OfferingReport"/>
  </wstop:Topic>
  <wstop:Topic name="SensorMetadataChanged">
    <wstop:Topic name="SensorRecalibrated"
messageTypes="swes:RecalibrationReport"/>
    <wstop:Topic name="SensorMoved" messageTypes="swes:MovementReport"/>
  </wstop:Topic>
</wstop:TopicNamespace>
```

As we can see, the SWE Common Service (SWES) topic namespace defines two immediate topics: ServiceMetadataChanged and SensorMetadataChanged, each with their specific child topics. The topic namespace has the target namespace "http://www.opengis.net/swes/1.0" and is final. While the two root topics have no constraints on the types of published messages, the child topics define such constraints.

Listing 7 - Sensor Planning Service 2.0 Topic Namespace example

```
<?xml version="1.0" encoding="UTF-8"?>
<wstop:TopicNamespace targetNamespace="http://www.opengis.net/sps/2.0"
xmlns:wstop="http://docs.oasis-open.org/wsn/t-1"
xmlns:sps="http://www.opengis.net/sps/2.0" final="false">
  <wstop:Topic name="TaskStatusUpdate">
    <wstop:Topic name="Reserved" messageTypes="sps:ReservationReport"/>
    <wstop:Topic name="Accepted" messageTypes="sps:StatusReport"/>
    <wstop:Topic name="Cancelled" messageTypes="sps:StatusReport"/>
    <wstop:Topic name="Failed" messageTypes="sps:StatusReport"/>
    <wstop:Topic name="Completed" messageTypes="sps:StatusReport"/>
    <wstop:Topic name="NewDataAvailable" messageTypes="sps:StatusReport"/>
    <wstop:Topic name="Expired" messageTypes="sps:StatusReport"/>
  </wstop:Topic>
</wstop:TopicNamespace>
```

Here we see another topic namespace which is not final. This means that namespace extensions can be defined later on. However, right now the topic namespace defines only one root topic (*TaskStatusUpdate*) with a list of child topics. Almost all topics use the same message type, only the *Reservation* topic uses a different one.

Listing 8 - SPS Implementation specific Topic Namespace example

```
<?xml version="1.0" encoding="UTF-8"?>
<wstop:TopicNamespace targetNamespace="http://www.igsi.eu/sps"
xmlns:wstop="http://docs.oasis-open.org/wsn/t-1"
xmlns:sps="http://www.opengis.net/sps/2.0">
  <wstop:Topic name="Suspended" parent="sps:TaskStatusUpdate/Accepted"
messageTypes="sps:StatusReport"/>
  <wstop:Topic name="Resumed" parent="sps:TaskStatusUpdate/Accepted"
messageTypes="sps:StatusReport"/>
</wstop:TopicNamespace>
```

The last topic namespace in fact defines two extension topics for the topic namespace introduced in Listing 7. The two root topics both have the same parent, the *Accepted* topic, a child topic of the *TaskStatusUpdate* topic in the SPS 2.0 target namespace. The *Suspended* and *Resumed* topics thus represent child topics of the *Accepted* topic. The parent topic is identified via an XPath like expression, the so called concrete topic expression dialect as defined by WS-Topics. Figure 60 provides an overview of the three topic namespaces.

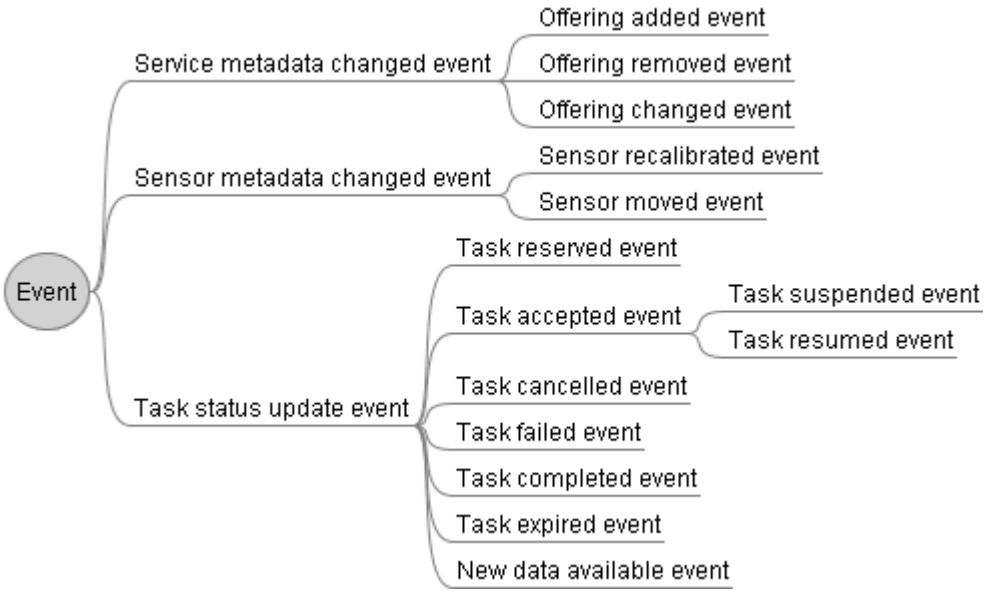

Figure 60 – Conceptual overview of the exemplary topic namespaces

10.4.2.2 Topic Set

Now we know how the topic namespaces are defined. But how does a service actually use them? A web service will define a so called *topic set*. This set will contain all the topics on which the service publishes notifications. This is usually the list of topics defined by one or more topic namespaces, or a subset thereof. However, with the ad-hoc topic namespace as defined in the previous paragraph, a service implementation may also add ad-hoc topics to its topic set.

Each topic that belongs to the topic set is given by a QName consisting of the target namespace from the according topic namespace and the topic name as the local name. Have a look at Listing 9 and its conceptual equivalent shown in Figure 61.

Listing 9 - TopicSet example

```
<?xml version="1.0" encoding="UTF-8"?>
<wstop:TopicSet xmlns:wstop="http://docs.oasis-open.org/wsn/t-1"
xmlns:swes="http://www.opengis.net/swes/1.0"
xmlns:sps="http://www.opengis.net/sps/2.0" xmlns:igsi="http://www.igsi.eu/sps">
  <swes:ServiceMetadataChanged wstop:topic="true">
    <OfferingAdded wstop:topic="true"/>
    <OfferingRemoved wstop:topic="true"/>
    <OfferingChanged wstop:topic="true"/>
  </swes:ServiceMetadataChanged>
  <swes:SensorMetadataChanged>
    <SensorRecalibrated wstop:topic="true"/>
  </swes:SensorMetadataChanged>
  <sps:TaskStatusUpdate wstop:topic="true">
    <Accepted wstop:topic="true">
      <igsi:Suspended wstop:topic="true"/>
      <igsi:Resumed wstop:topic="true"/>
    </Accepted>
    <Cancelled wstop:topic="true"/>
    <Failed wstop:topic="true"/>
    <Completed wstop:topic="true"/>
    <NewDataAvailable wstop:topic="true"/>
  </sps:TaskStatusUpdate>
</wstop:TopicSet>
```

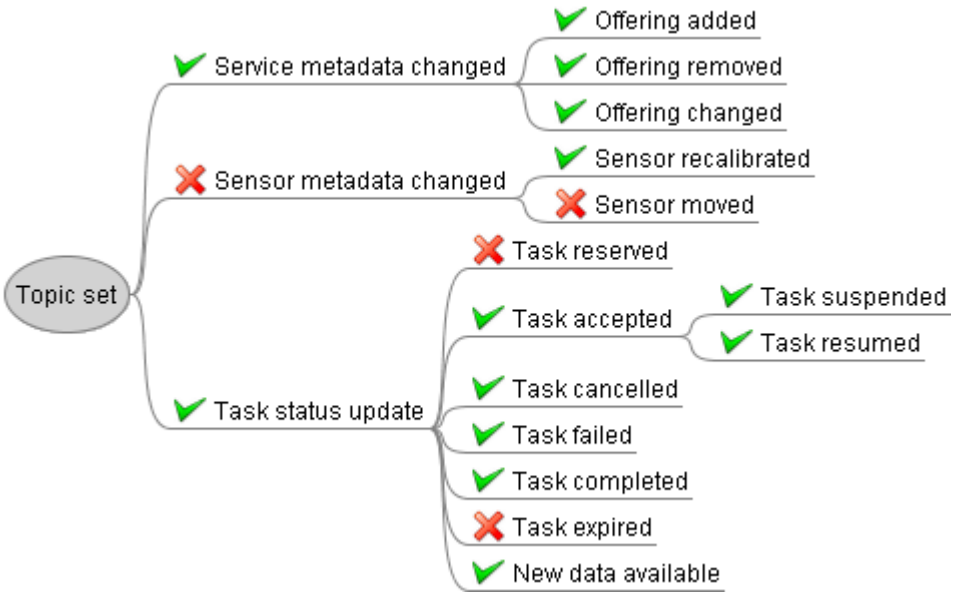

Figure 61 – Conceptual overview of the topic set

We see an example of a topic set as it might be provided by an SPS implementation. It lists all topics defined by the SWES topic namespace for service metadata changes. As said before, each topic is given by a QName, consisting of the topic namespace's target

namespace - in this case http://www.opengis.net/swes/1.0 - and the topic's name. The namespace is omitted for each child topic of the same target namespace. Each topic is contained in the same position of the topic hierarchy as defined by the topic namespace it belongs to.

Note the wstop:topic attribute. If present and set to true, it marks the topic as a member of the services topic set. All topics that are posted on such a topic will be sent to clients that subscribe for such a topic. If the attribute is omitted, the topic is not part of the service's topic set. The service will not post messages that would be posted on that topic and will not accept a subscription that references that topic. An example of this is the *swes:SensorMetadataChanged* topic - it does not have the wstop:topics attribute. However, the *SensorRecalibrated* topic is available. If one compares this to the SWES topic namespace, one will notice that the topic set also does not include the *SensorMoved* topic. The topic set needs to contain the full topic path (as defined by the topic namespace) to each topic marked with the wstop:topics attribute.

When looking at the *TaskStatusUpdate* topic, one will notice that it is also missing some child topics that are defined by the topic namespace (Reserved and Expired). However, the Accepted topic lists the two extension topics as defined by the topic namespace from Listing 8. Because these extension topics are from another target namespace, they need to be represented by a full QName.

No ad-hoc topic namespace is used. If it was used, then the topic set would contain direct child elements from the empty target namespace (so would not have a prefix).

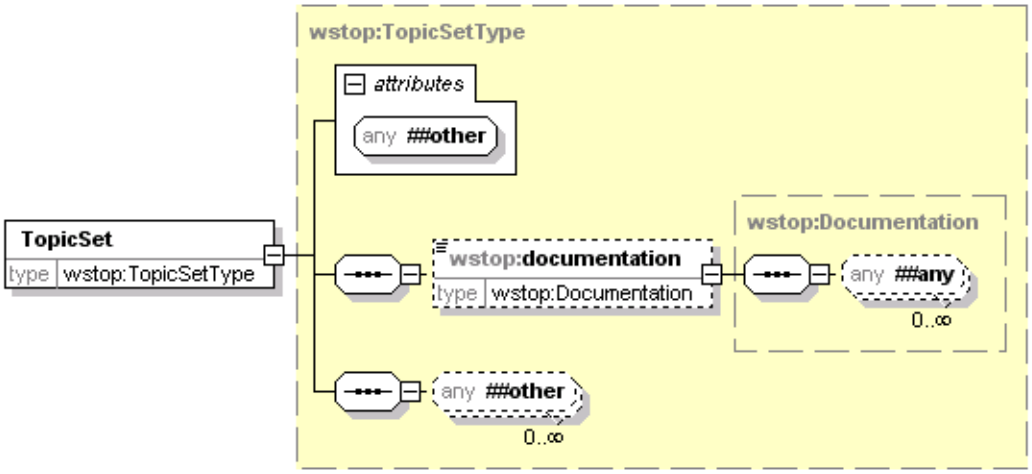

Figure 62 - TopicSet element as defined by WS-Topics, in XMLSpy notation

The schema of the topic set (see Figure 62) is quite simple. The only really defined element is the documentation element, which may contain any metadata. Other than that, no restrictions are made. Services can add the topic elements they want to publish. In addition, other elements can be included, whatever is required by the given application domain. This also shows that a valid topic set could be empty. This supports use cases in

which the topic set is dynamic. For example, a notification broker might initially not have any publisher registered. In this case the topic set would be empty. When new publishers are registered, the topic set would grow.

10.4.3 Examples of Defined Notification Messages

The Web Service Resource Framework contains three examples for standard topic namespaces. The ResourceLifetime topic namespace as defined by WS-ResourceLifetime and the ResourcePropertiesTopicNamespace and AnyResourcePropertyValueChange topic namespaces defined in the WS-ResourceProperties specification. Some schemas of message types allowed by the WSRF topic namespaces are shown in Figure 63 and Figure 64.

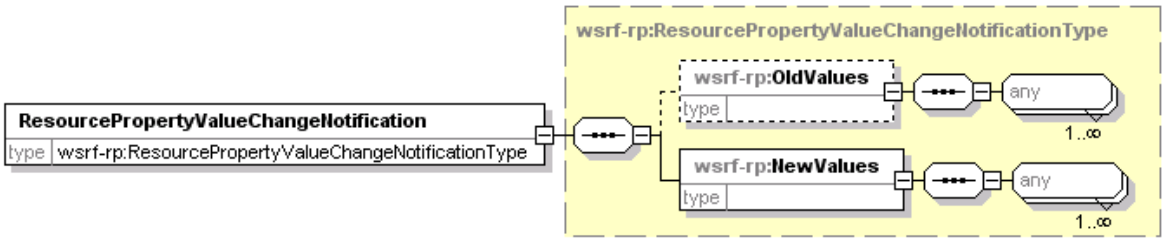

Figure 63 - ResourcePropertyValueChangeNotification element in XMLSpy notation

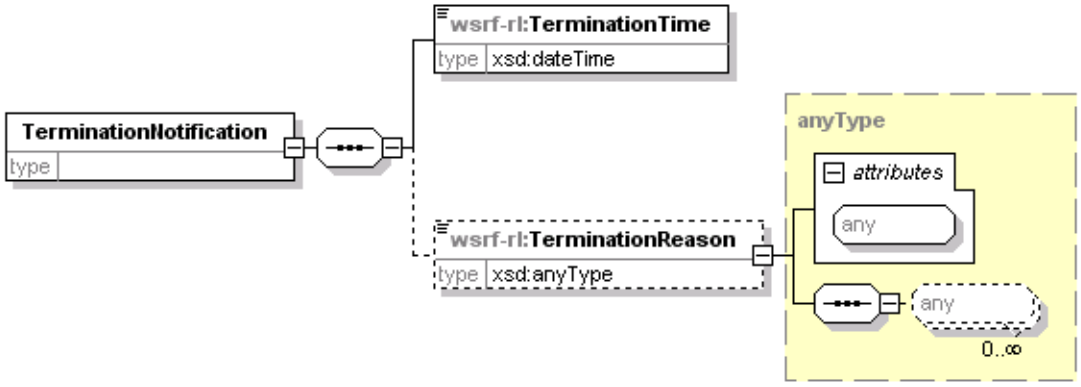

Figure 64 - TerminationNotification element in XMLSpy notation

We do not want to go into too much detail at this point. Just take these two schema definitions and the topic namespaces as examples how a web service specification like those defined by the OGC could also define their specific topic namespaces, their semantics as well as according message type schema.

Note that although the example message types are defined in XML schema and are not intended for direct human consumption, nothing prevents one from defining multiple message types (e.g. encoded in XHTML) that express the same information and publish them on different topics. In addition one can have intermediary services - or clients - that transform the information contained in XML encoded messages to human readable form,

e.g. using templates or stylesheets. The Web Notification Service would be a good candidate to be enhanced to support such functionality.

10.4.4 Basic Pub/Sub Functionality

The core functionality of WS-Notification is made available through three interfaces: the NotificationProducer, NotificationConsumer and SubscriptionManager (see Figure 65).

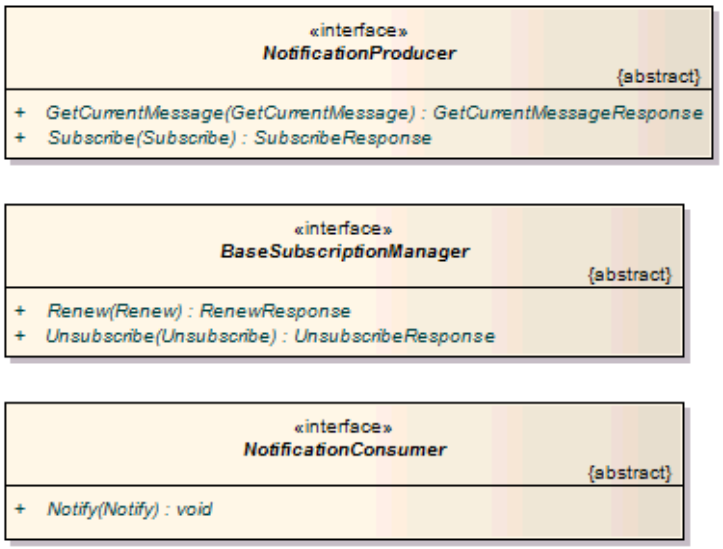

Figure 65 - Mandatory Interfaces defined by WS-BaseNotification

The NotificationProducer interface is a factory for subscriptions. It provides the Subscribe method. In addition it implements the GetCurrentMessage operation, which can be used to retrieve the last notification that was published on a given topic. The GetCurrentMessage operation is a convenience operation. If the NotificationProducer does not have the functionality to cache all last messages from all topics or if it does not make use of topics, then it would return a fault message as defined by WSN-B. As this makes the GetCurrentMessage operation more or less optional, we will not describe it. The Subscribe operation is the most interesting one for us at this point. We will describe the other interfaces (BaseSubscriptionManager and NotificationConsumer) when we explain the Subscribe method.

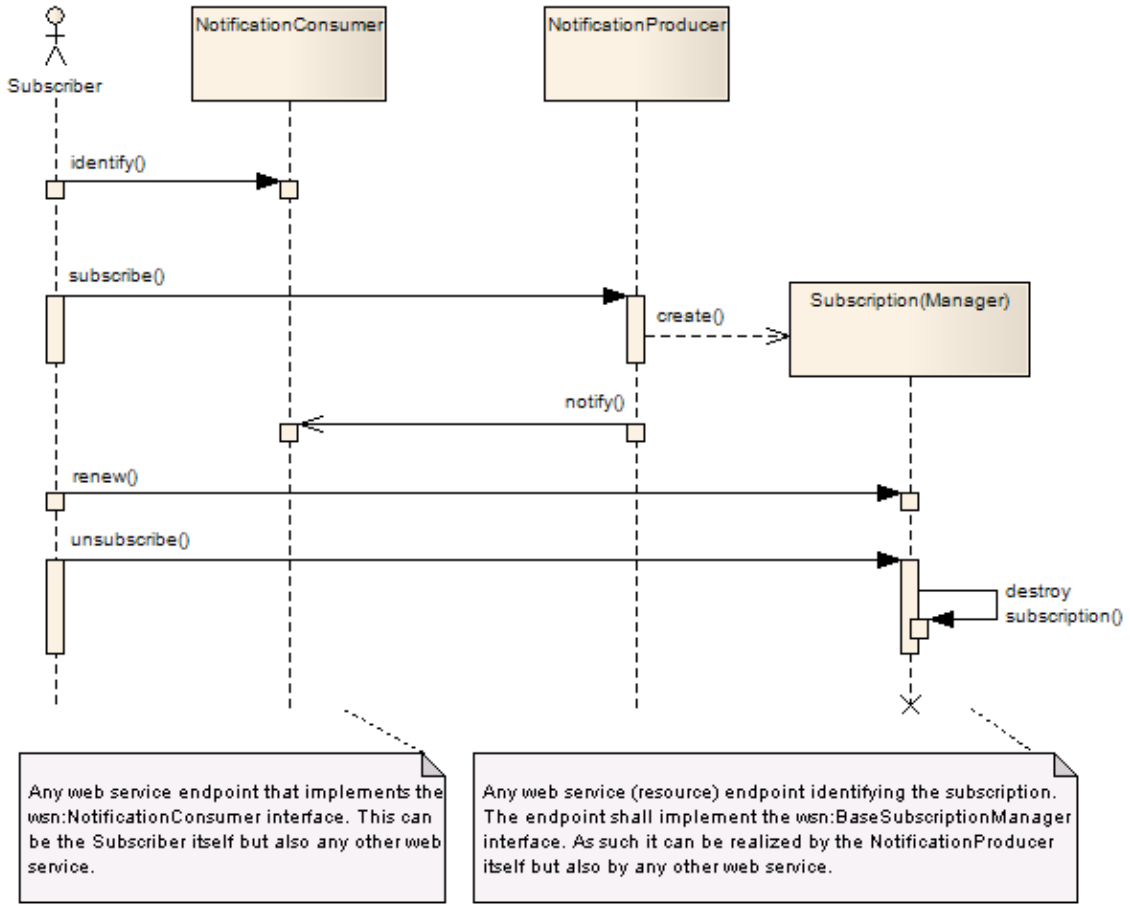

Figure 66 - Basic Interactions in Pub/Sub using WS-Notification

Figure 66 shows the basic interactions involved when doing Pub/Sub with WS-Notification. First of all, we have four entities: the Subscriber, the NotificationConsumer, the NotificationProducer and the SubscriptionManager. The latter three obviously implement the mandatory WSN-B interfaces. Which application implements them is not defined. Thus, the NotificationConsumer and Subscriber may very well be the same service. The same may be true for the NotificationProducer and SubscriptionManager. Another example is when a NotificationProducer service also implements the NotificationConsumer interface and subscribes at another NotificationProducer; as an example of service chaining.

The Subscriber first needs to know the endpoint of the entity that shall receive all messages published for the intended subscription. Usually, this is a priori knowledge - in some use cases the consumer might be identified on the fly, for example based on its geographic location or role in general. The Subscriber will then invoke the subscribe operation at the NotificationProducer, which will - given that the subscription request is acceptable - create a subscription and return the endpoint of the entity responsible for managing the subscription in the subscription response. Figure 67 shows the schema of a subscription request

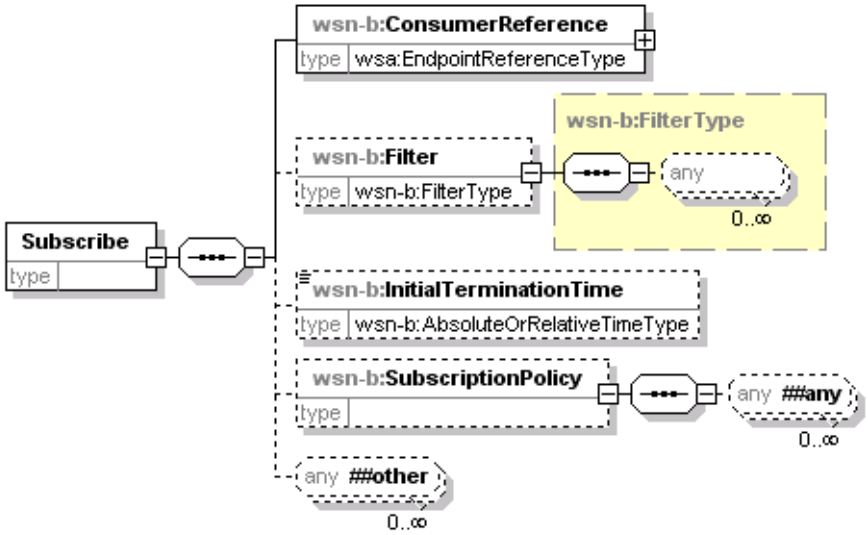

Figure 67 - Subscribe request element as defined by WS-BaseNotification in XMLSpy notation

The only required information is the WSA endpoint reference that identifies the consumer of all notifications matching the subscription. In addition, a filter may be provided. Which filters are possible is defined in more detail in the WSN-B specification. The filter element may contain a list of filter statements that either identify one or more topics, query upon notification content or upon the properties of the entity that produced the notification. The latter is a specific case which we are not going to cover in more detail in this tutorial. All filter statements are connected with a logical AND, i.e. if a new notification does not satisfy all filter statements, it does not match the subscription criteria and thus will not be sent to the notification consumer. In addition, the Subscriber may request that the subscription may be terminated at a given time. The time is either given as an absolute date time or as duration relative to the current time. Therefore subscriptions are stateful resources. Wait a minute; are we talking about a WS-Resource? Yes we are; we will come back to this when we explain the SubscribeResponse.

Back to the Subscribe request: in addition to the properties already explained, a Subscribe request may also contain some subscription policies and any extension needed by the application. Until now (and to our best knowledge), no standard subscription policies have been defined except for the UseRaw policy (to which we will come back later) - this is an opportunity for the OGC when specific Pub/Sub behavior needs to be specified.

An exemplary Subscribe request is shown in Listing 10.

Listing 10 - Subscribe request example

```
<?xml version="1.0" encoding="UTF-8"?>
<wsn-b:Subscribe xmlns:wsn-b="http://docs.oasis-open.org/wsn/b-2"
xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <wsn-b:ConsumerReference>
    <wsa:Address>http://www.igsi.eu/consumer</wsa:Address>
  </wsn-b:ConsumerReference>
  <wsn-b:Filter>
    <wsn-b:TopicExpression Dialect="http://www.w3.org/TR/1999/REC-xpath-
19991116">//*[namespace-uri()='http://www.opengis.net/sps/2.0' and local-
name()='TaskStatusUpdate']//*[@*[namespace-uri()='http://docs.oasis-
open.org/wsn/t-1' and local-name()='topic']='true']</wsn-b:TopicExpression>
  </wsn-b:Filter>
</wsn-b:Subscribe>
```

NOTE: The operation might also be conducted using SOAP. If WSA is used, then the WSA action URI of the request would be http://docs.oasis-open.org/wsn/bw-2/NotificationProducer/SubscribeRequest.

This request contains a topic expression, identifying all topics in the topic set that are descendants of the SPS 2.0 TaskStatusUpdate topic and having a wstop:topic attribute with value 'true'. The rather tedious constructs with namespace-uri() and local-name() functions are used to prevent problems with different namespace prefixes that might arise when evaluating the XPath 1.0 construct against the notification producer's topic set. All notifications published on the identified topics shall be sent to the consumer with address 'http://www.igsi.eu/consumer'.

The service, accepting the subscription, could send a response like shown in Listing 11.

Listing 11 – Subscribe response example

```
<?xml version="1.0" encoding="UTF-8"?>
<wsn-b:SubscribeResponse xmlns:wsn-b="http://docs.oasis-open.org/wsn/b-2"
xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsn-b:SubscriptionReference>
        <wsa:Address>http://my.sps.com/subscription/123</wsa:Address>
    </wsn-b:SubscriptionReference>
    <wsn-b:CurrentTime>2009-04-12T15:28:00+02:00</wsn-b:CurrentTime>
    <wsn-b:TerminationTime>2009-10-12T15:28:00+02:00</wsn-b:TerminationTime>
</wsn-b:SubscribeResponse>
```

NOTE: The operation might also be conducted using SOAP. If WSA is used, then the WSA action URI of the response would be http://docs.oasis-open.org/wsn/bw-2/NotificationProducer/SubscribeResponse.

In the response, the service accepted the subscription request. It created a subscription which terminates at 2009-10-12T15:28:00+02:00 server time (current server time is 2009-0412T15:28:00+02:00). The subscription thus has a scheduled termination time. The entity responsible for managing the subscription (and thus representing the subscription) is located at the given endpoint reference (with address http://my.sps.com/subscription/123 - note that in the example no WSA reference properties are used, though a given implementation is of course allowed to use them). The schema of the SubscribeResponse is quite simple and shown in Figure 68.

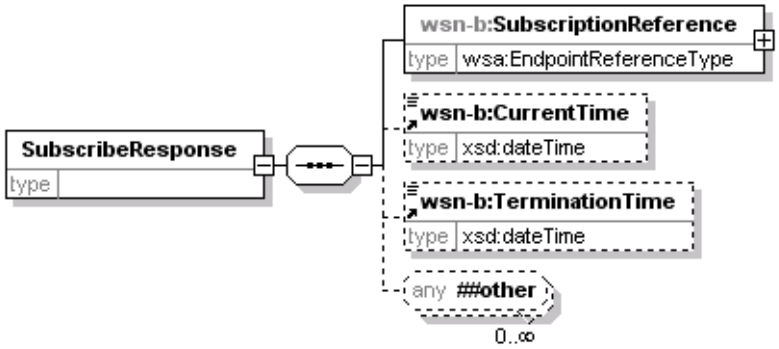

Figure 68 - SubscribeResponse element as defined by WS-BaseNotification in XMLSpy notation

The subscription reference provides the endpoint address of the SubscriptionManager. If the subscription has a defined lifetime (i.e. does not live indefinitely) the current (server) time and (subscription) termination time (relative to the server time) are provided. In addition, extension properties may be added as required by the application.

The WS-BaseNotification specification defines several faults that might occur when invoking the Subscribe operation. They are extensions of the WS-BaseFaults type. We will not go into more detail about these faults at this time. For more information, please consult the WS-BaseNotification specification.

When clients want to remove a certain subscription, they send a simple Unsubscribe request to the manager endpoint of their subscription (the SubscriptionReference, provided in the SubscribeResponse). If the subscription has a finite lifetime, then clients can also renew them by invoking the Renew operation. This is one way to alter a subscriptions lifetime. Another one is to use the operations defined by WS-ResourceLifetime. However, in that case the subscription needs to be implemented as a WS-Resource. So let's talk about this.

If a subscription is modeled as a WS-Resource (this can be recognized by looking at the WSDL description of the SubscriptionManager), it has to implement the GetResourceProperty operation defined by WS-ResourceProperties and the Destroy operation defined by WS-ResourceLifetime (for immediate destruction). It also needs to implement some properties. The overall construct is shown in Figure 69.

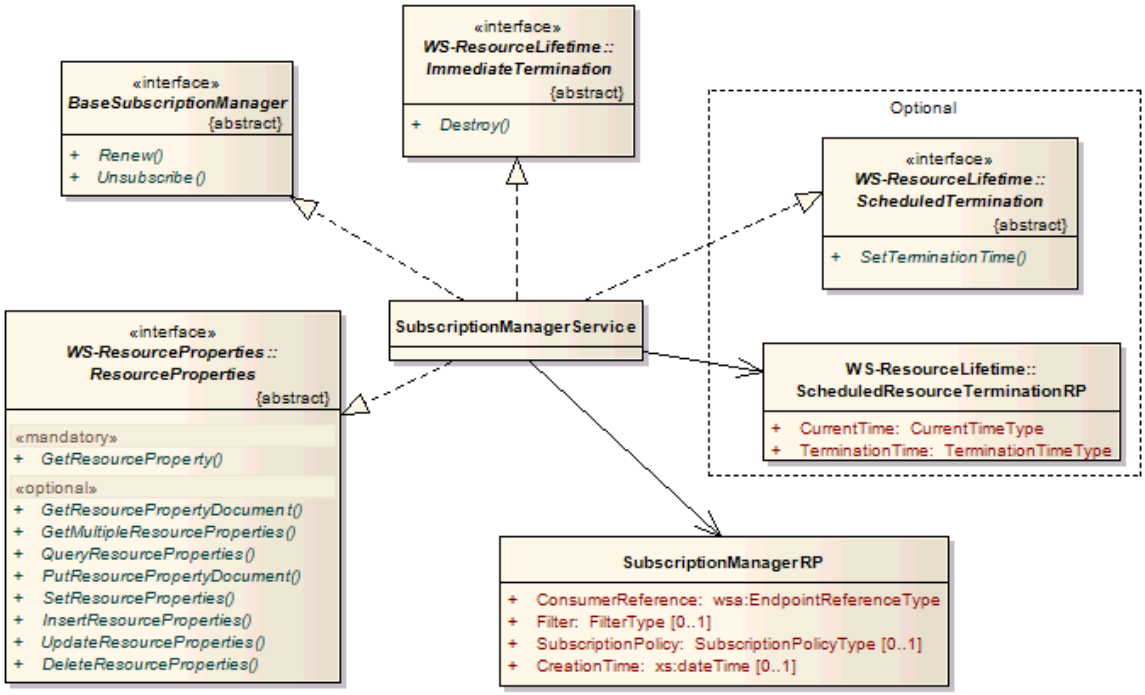

Figure 69 - SubscriptionManager as WS-Resource

The mandatory resource properties (contained in the SubscriptionManagerRP class) reflect the information provided by the client in the Subscribe request and by the service in the SubscribeResponse. Implementation of the ScheduledTermination interface from WS-ResourceLifetime and the according resource properties (CurrentTime and TerminationTime) are optional. However, if the subscription has a scheduled termination time (indicated by the service when including the CurrentTime and TerminationTime in the SubscribeResponse) then the interface and resource properties associated with scheduled resource termination become mandatory.

The default way to end a subscription is the Unsubscribe operation. In addition, if the subscription is modeled as a WS-Resource, the Destroy operation can be used. If the resource has scheduled termination time, then the subscription can also terminate by having reached its termination time.

If the Subscription(Manager) can be a WS-Resource, what about the NotificationProducer? It can be modeled as a WS-Resource as well - see Figure 70.

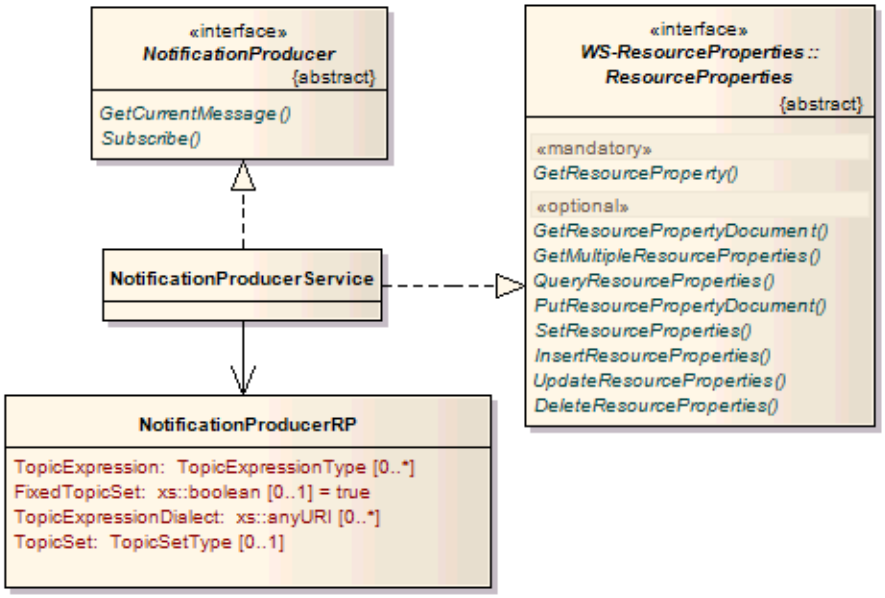

Figure 70 - NotificationProducer as WS-Resource

A NotificationProducer usually has a topic set (if it uses topics to structure its notifications). If so, then it may indicate whether the topic set is fixed or not. For the convenience of the client, the service may also provide a list of so called topic expressions. These topic expressions use one of the (topic expression) dialects defined by WS-Topics to identify topics that are supported by the NotificationProducer. In that sense, the information the topic expressions provide is complementary to that contained in the topic set. However, clients may use the given topic expressions in their subscription requests and can be reasonably sure that they will receive notifications matching the expressions. This in fact is a way for services to provide pre-configured subscription topics[28].

Now we know how to model notification topics. We know how to include the topic set into the metadata of a service. We also know how to subscribe at the service. What we have not covered yet is the NotificationConsumer interface which defines the Notify operation. This is a one-way operation (see chapter 10.2). The schema of the operation is shown in Figure 71.

[28] With the definition of a suitable content model and dialect for these topic expressions, the list could easily be transformed into a webpage for clients to choose the pre-configured subscription topics that are relevant to them.

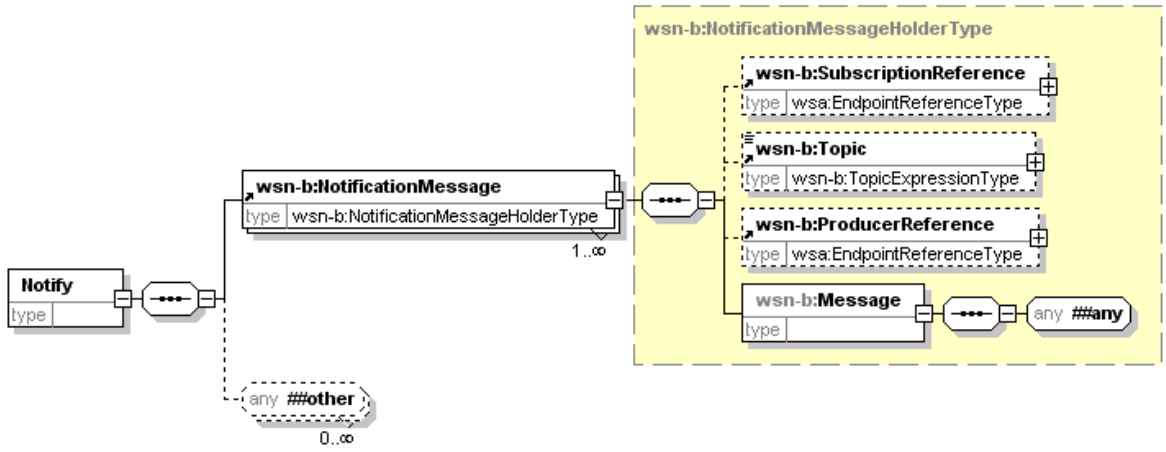

Figure 71 - Notify request element as defined by WS-BaseNotification in XMLSpy notation

We see that a notify request may contain a list of NotificationMessages (and other application specific extensions). Each notification message may reference the subscription which it matches. This helps clients to associate incoming messages with their subscriptions. In addition, the entity that produced the message may be referenced. The topic on which the notification was posted may also be identified. Finally, the message itself is - of course - provided. One can see that the message may be any XML element, the structure of it is defined for example by certain service specifications like a WMS, WFS or SPS. Listing 12 shows an example Notify request.

Listing 12 - Notify request example

```
<?xml version="1.0" encoding="UTF-8"?>
<wsnt:Notify xmlns:wsnt="http://docs.oasis-open.org/wsn/b-2"
xmlns:wsa="http://www.w3.org/2005/08/addressing"
xmlns:sps="http://www.opengis.net/sps/2.0">
  <wsnt:NotificationMessage>
    <wsnt:SubscriptionReference>
      <wsa:Address>http://www.example.org/SubscriptionManager</wsa:Address>
    </wsnt:SubscriptionReference>
    <wsnt:Topic Dialect="http://docs.oasis-open.org/wsn/t-
1/TopicExpression/Concrete">
          sps:TaskStatusUpdate/NewDataAvailable
        </wsnt:Topic>
    <wsnt:ProducerReference>
      <wsa:Address>http://www.example.org/NotificationProducer</wsa:Address>
    </wsnt:ProducerReference>
    <wsnt:Message>
      <sps:StatusReport>
        <!-- contents omitted for brevity -->
      </sps:StatusReport>
    </wsnt:Message>
  </wsnt:NotificationMessage>
</wsnt:Notify>
```

NOTE: The operation might also be conducted using SOAP. If WSA is used, then the WSA action URI of the request would be http://docs.oasis-open.org/wsn/bw-2/NotificationConsumer/Notify.

We said before that WSN-B defined a UseRaw subscription policy. Now is the time to explain what the policy is used for. When a client adds the policy to its subscribe request, it indicates that the service shall send all messages to the given consumer in raw notation, i.e. by not including it in the Notify (request) wrapper. In our example, the sps:StatusReport would be sent directly. If SOAP is used, then the raw message would be put into the SOAP body. However, the WSA action URI in the SOAP header would still be that as defined for the Notify request.

Now we have covered - with some excursions - the basic Pub/Sub functionality of WS-Notification. We discussed how to model topics, how to provide this information for a given service, how to subscribe, how to manage a subscription and how to notify consumers. How notifications are published to and by the NotificationProducer is up to that service.

10.4.5 Advanced Functionality

The WS-BaseNotification and WS-BrokeredNotification define additional functionality which will be explained briefly in the following subclauses.

10.4.5.1 Pausable Subscriptions

Sometimes, it is useful to pause a subscription. In such use cases, the BaseSubscriptionManager interface can be extended so that the SubscriptionManager implements the PausableSubscriptionManager interface, see Figure 72.

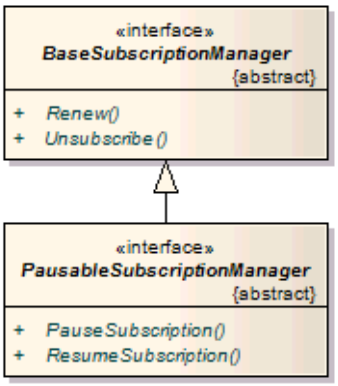

Figure 72 - PausableSubscriptionManager interface as defined by WS-BaseNotification

The interface adds two operations with which a subscription can simply be paused and resumed.

Pausing of subscriptions is relevant in use cases where on-demand publishing is used.

Figure 73 provides an overview of the possible states and transitions of a pausable subscription.

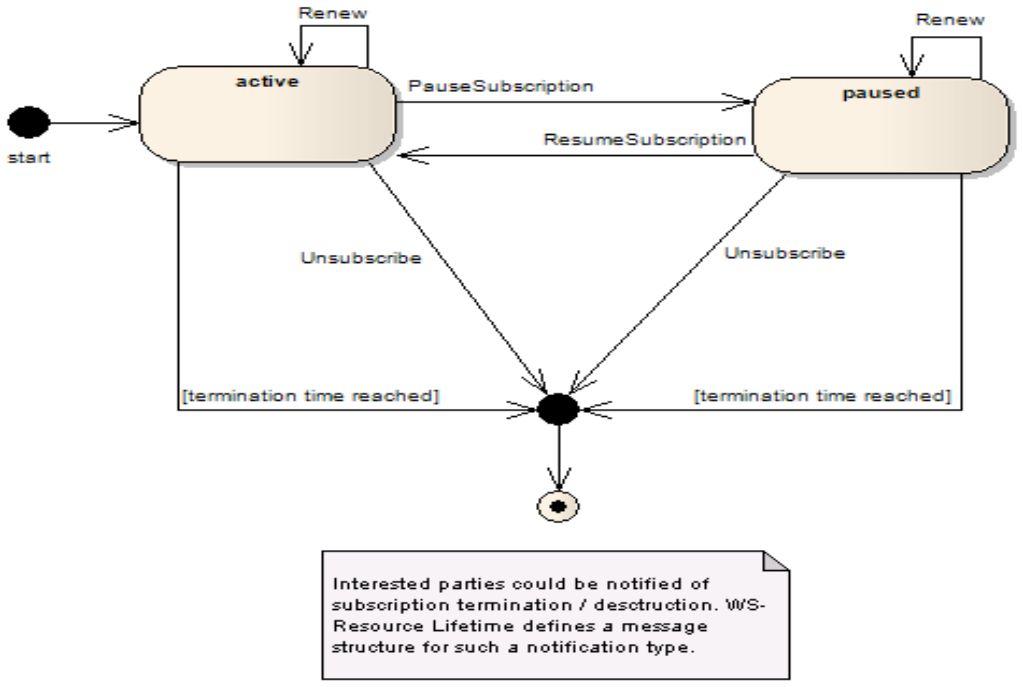

Figure 73 - Possible states and transitions of a pausable subscription

Note that WSN-B does not define a resource property to indicate in which state the subscription is in (active or paused). If required, this property would need to be added.

10.4.5.2 Pulling Notifications

WS-Notification is using push-style notification by default. However, in some use cases, especially where firewalls or dynamic IP addresses of clients are a problem, pulling notifications appears to be the better option. This functionality is offered by WS-BaseNotification. Subscribers need to create a so called pull point which represents a message cache to which the NotificationProducer sends messages to. Figure 74 shows the relevant interfaces.

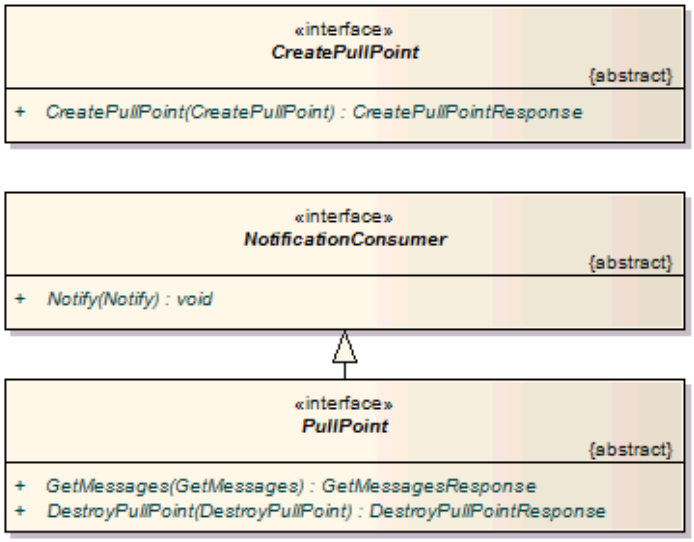

Figure 74 - PullPoint interfaces as defined by WS-BaseNotification

First we have the CreatePullPoint factory interface. Any service may implement this interface. When the CreatePullPoint operation is invoked by a client, a PullPoint is created. This is a web service that implements the NotificationConsumer interface. This means that its endpoint can be used as the consumer reference in a Subscribe request. The NotificationProducer will then send all matching notifications to the PullPoint. Clients can periodically access the PullPoint and retrieve cached messages via the GetMessages operation. When the PullPoint is no longer needed, it can be removed by invoking the DestroyPullPoint operation. Figure 75 shows these interactions in more detail.

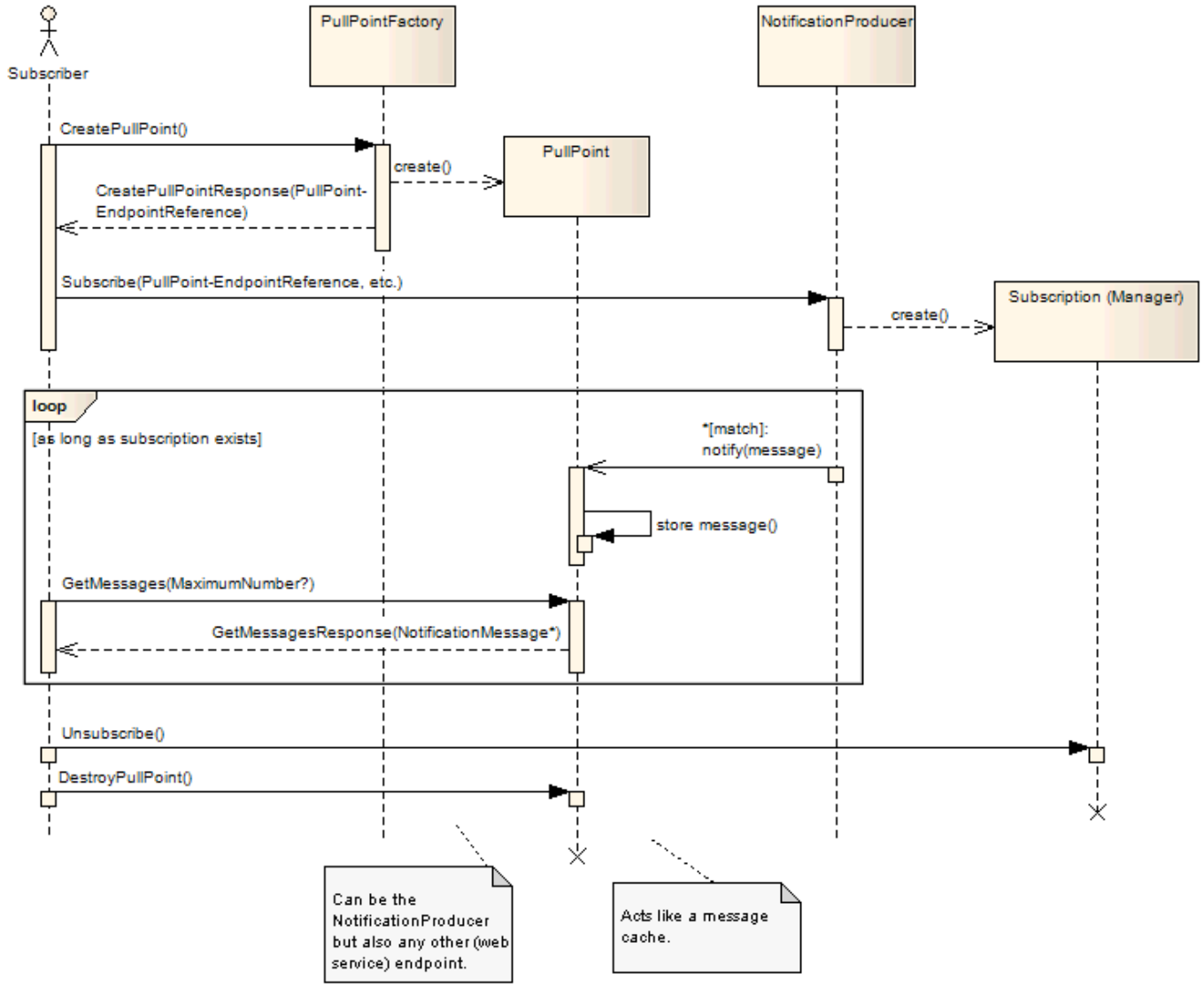

Figure 75 – Pull style notification with WS-Notification using the PullPoint mechanism

A PullPoint may also be modeled as a WS-Resource. In that case it will at least implement the immediate termination interface defined by WS-ResourceLifetime. It may also implement scheduled termination behavior.

10.4.5.3 Reliable Notification

Notifying a consumer is performed with a one-way message exchange, i.e. the NotificationProducer fires a message (either in raw form or encoded in the Notify element) and then continues with normal operation. The NotificationConsumer is not obliged to send any form of response and the NotificationProducer does not expect one. However, in some use cases, especially in security sensitive use cases, reliable delivery of messages is a requirement.

Acknowledgement of message receipt can be implemented by adding WS-ReliableMessaging functionality as mentioned in chapter 8.8. In addition, the functionality of the underlying transport layer can be used, for example HTTP response codes if HTTP is used for transporting SOAP messages. The latter is not always sufficient, because the correct usage of HTTP response codes when using SOAP is not always adhered to. WS-I (2007) defines HTTP codes 200 (Ok) and 202 (Accepted) for indication of successful outcome of an HTTP request, even if no SOAP response is delivered.

10.4.5.4 Notification Brokering

WS-BrokeredNotification defines how a NotificationProducer can be extended to function as a NotificationBroker. A broker (see chapter 6.6.1.5) supports dynamic registration of new notification publishers. This is in contrast to a NotificationProducer, which has its own set of publishers. A NotificationBroker realizes the NotificationProducer, NotificationConsumer and RegisterPublisher interface, see Figure 76.

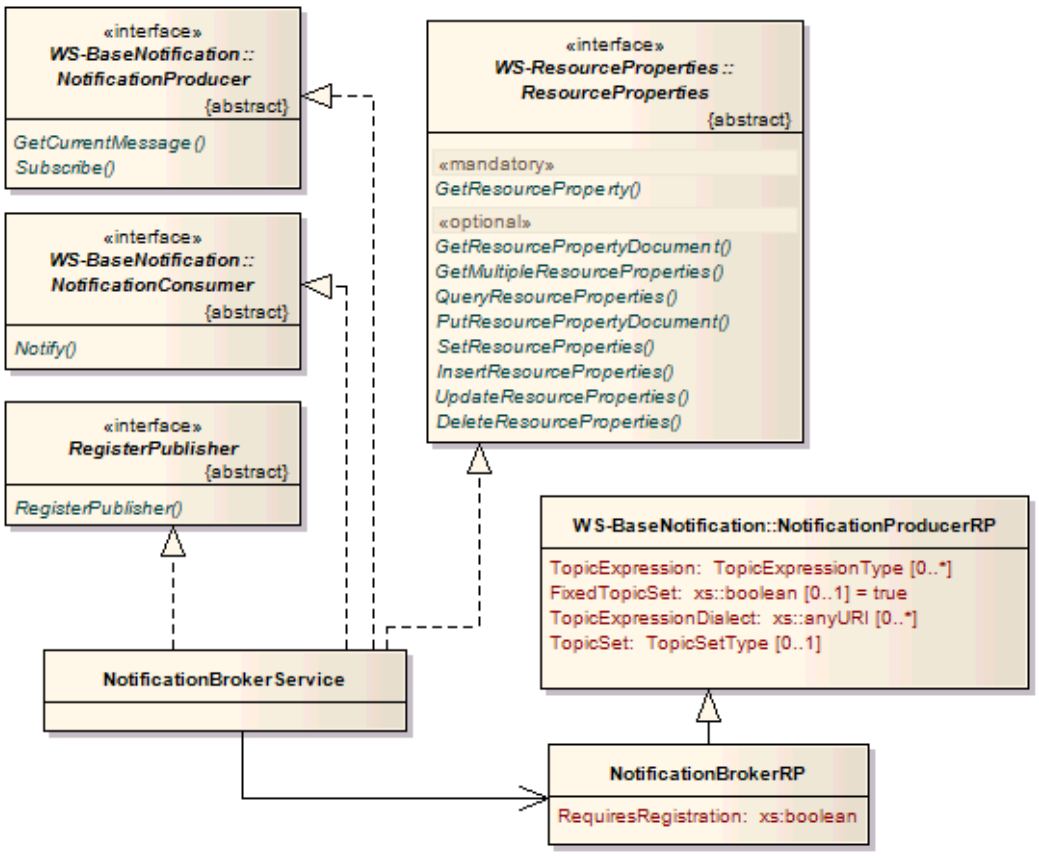

Figure 76 - NotificationBroker as WS-Resource

The RegisterPublisher operation is used to register new publishers at the service. We do not want to go into too much detail at this point. Suffice to say that upon accepting a new

publisher, the broker will create a PublisherRegistration which is managed by a so called PublisherRegistrationManager. The entity that registered the publisher will get a reference to this manager back in the RegisterPublisherResponse. This is much like the handling of subscriptions.

If a notification broker is implemented as a WS-Resource, it uses the same resource properties as a NotificationProducer, only adding a flag which tells clients whether new publishers need to be registered before they may publish notifications to the service or not. If requiresRegistration equals *false* then any publisher may send notifications to the broker. Thus, if security is of interest and the broker is part of an open network, registration of new publishers is likely going to be a requirement.

Figure 77 shows the interactions of the various entities involved in a brokered Pub/Sub scenario.

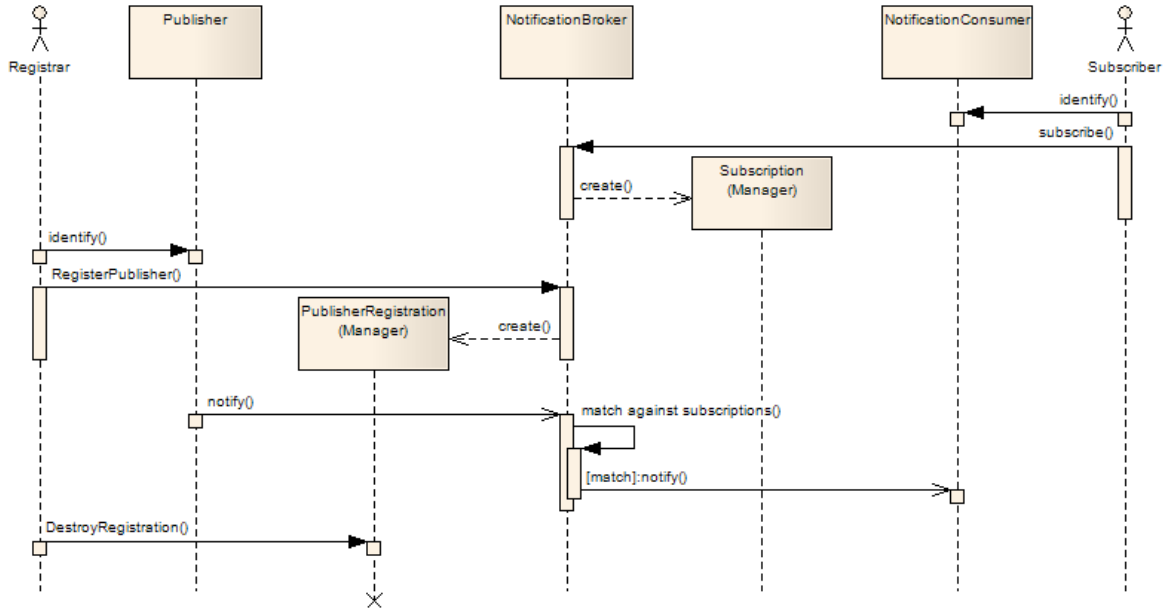

Figure 77 - Interactions in brokered Pub/Sub using WS-Notification

10.4.5.5 WS-Notification Resource Properties & OWS Capabilities

The information required performing non-trivial pub/sub functionality like available dialects, a service's topic set etc. is usually provided by WS-Notification using resource properties. Resource properties are usually retrieved via the WS-ResourceProperties operations and defined in the WSDL description of a service. OGC Web Services that use neither WSDL nor the Web Service Resource Framework will nevertheless need to provide the information which enables clients to perform meaningful subscriptions. The place to put service metadata in OWS is usually the Capabilities document. At this point, we do not want to discuss whether the information of the Capabilities document should rather be provided via the mechanisms defined by WS-ResourceProperties - though such a discussion would certainly be interesting.

An attempt to add notification metadata to a services capabilities document has been made by the SPS and SWE Common 2.0 SWGs. The NotificationProducerMetadata data type shown in Figure 78 can be added as a contents section to the Capabilities document of an OWS.

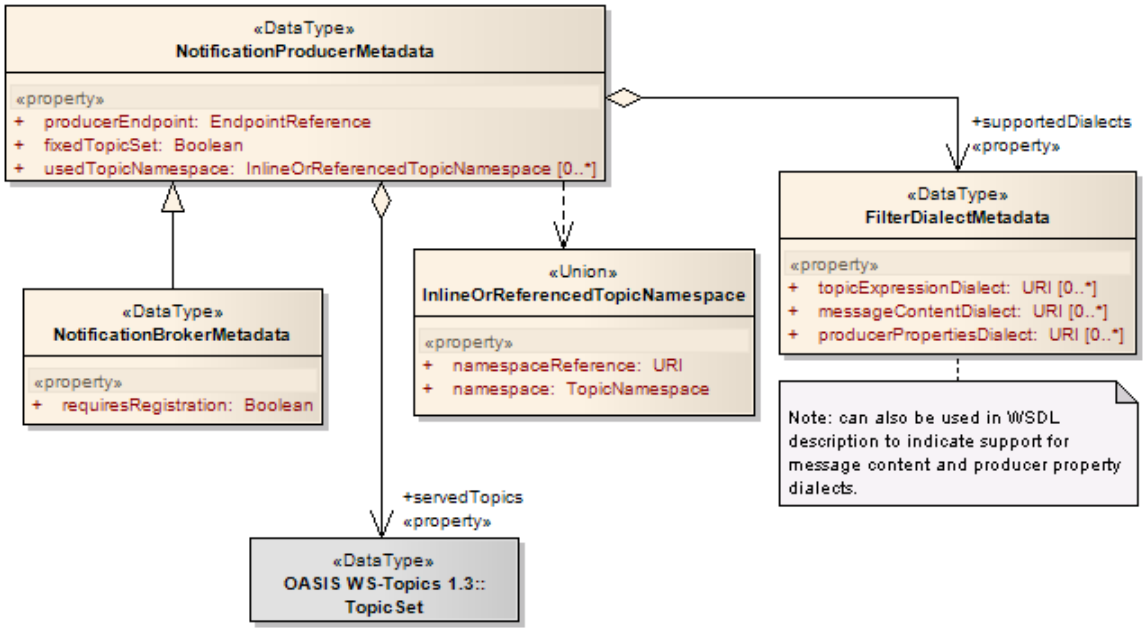

Figure 78 - Notification Metadata Data Types for inclusion in an OWS's Capabilities Document

The information contained in the various classes resembles more or less the resource properties defined for a NotificationProducer and -Broker. However, the data types shown are not final and should be used with care.

10.5 Conclusion

WS-Notification is suited to implement Pub/Sub functionality in OGC Web Services, at least for the SOAP/WS-* architectural style. Because of its transport binding independence, it is likely also suitable for the common OWS architectural style "Plain Old XML via HTTP". However, this needs to be proven via experiments (like an Interoperability Experiment or Testbed).

The interfaces defined by OASIS enable the implementation of the Consumer, Registrar and Provider interfaces[29], and thus also for the derived / aggregated interfaces and roles. In fact, this should enable the implementation of the OGC Event Architecture for SOAP/WS-* and POX style OWS.

[29] see chapter 6.6.2

Only a small set of the functionality specified by WS-Notification is mandatory. This already enables basic Publish/Subscribe functionality. More advanced functionality like resource properties, lifetime, pull-points, topics, brokering etc. is optional and can be leveraged if required. Multiple extension points are offered which enables the OGC to specify additional functionality / behavior required by the geospatial domain. Examples are notification and subscription policies that are not sufficiently defined by WS-Notification.

Problems that might occur with firewalls and dynamic IP addresses of consumers when pushing notifications can be solved by using the PullPoint mechanism offered by WS-Notification. In addition, other solutions that include an intermediary service (like a Web Notification Service) are possible.

To fully enable Pub/Sub in OWS using WS-Notification, some data types need to be defined to implement missing functionality (like a GetCapabilities contents section containing required resource property information or resource properties for indicating available filter dialects etc). An attempt to close these gaps is currently underway in the SPS 2.0 and SWE Common 2.0 SWGs, which intend to leverage WS-Notification for enabling Pub/Sub in the SOAP style of SPS 2.0 - and possibly also other SWE services.

WS-Notification was not intended as an implementation of the REST(ful) architectural style. However, it might be possible to map the principal functionality. This work would ideally be performed in a dedicated testbed (thread).

11 Annex B: WSDL Example of a Service using WS-Notification (informative)

The following listing contains the WSDL 1.1 description of a fictitious Sensor Planning Service that implements WS-Notification. The SPS specific operations are not included, except for the GetCapabilities and DescribeSensor operations.

The example WSDL description shows the usage of the WS-ResourceProperties operations and how to indicate resource properties in the port description. It shows how to implement the NotificationProducer port type defined by WS-BaseNotification. Finally, it also contains a policy statement to indicate that WS-Addressing shall be used by clients; anonymous endpoint references are not allowed.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://www.igsi.eu/sps"
xmlns:igsi="http://www.igsi.eu/sps"
xmlns:sps="http://www.opengis.net/sps/2.0"
xmlns:wsa="http://www.w3.org/2005/08/addressing"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdl-soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsrf-r="http://docs.oasis-open.org/wsrf/r-2"
xmlns:wsrf-rl="http://docs.oasis-open.org/wsrf/rl-2"
xmlns:wsrf-bf="http://docs.oasis-open.org/wsrf/bf-2"
xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/rp-2"
xmlns:wsn-b="http://docs.oasis-open.org/wsn/b-2"
xmlns:wsn-br="http://docs.oasis-open.org/wsn/br-2"
xmlns:wsn-t="http://docs.oasis-open.org/wsn/t-1"
xmlns:sml="http://www.opengis.net/sensorML/1.0.1"
xmlns:swe="http://www.opengis.net/swe/2.0.0"
xmlns:swes="http://www.opengis.net/swes/1.0.0"
xmlns:ows="http://www.opengis.net/ows/1.1"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
name="igsi">

  <wsdl:types>
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.w3.org/1999/xlink">
      <xsd:include schemaLocation="http://schemas.opengis.net/xlink/1.0.0/xlinks.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified" targetNamespace="urn:us:gov:ic:ism:v2">
      <xsd:include schemaLocation="http://schemas.opengis.net/ic/2.0/IC-ISM-v2.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.w3.org/2001/SMIL20/Language">
      <xsd:include schemaLocation="http://schemas.opengis.net/gml/3.1.1/smil/smil20-
language.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.w3.org/2001/SMIL20/">
      <xsd:include
schemaLocation="http://schemas.opengis.net/gml/3.1.1/smil/smil20.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.opengis.net/ows/1.1">
      <xsd:include schemaLocation="http://schemas.opengis.net/ows/1.1.0/owsAll.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.opengis.net/gml">
      <xsd:include schemaLocation="http://schemas.opengis.net/gml/3.1.1/base/gml.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.opengis.net/swe/2.0.0">
      <xsd:include schemaLocation="http://schemas.opengis.net/sweCommon/2.0.0/swe.xsd"/>
</xsd:schema>
```



```
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.opengis.net/sps/2.0.0">
      <xsd:include schemaLocation=" http://schemas.opengis.net/sps/2.0.0/sps.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.opengis.net/swes/1.0.0">
      <xsd:include
schemaLocation="http://schemas.opengis.net/sweServiceCommon/1.0.0/swes.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.opengis.net/sensorML/1.0.1">
      <xsd:include
schemaLocation="http://schemas.opengis.net/sensorML/1.0.1/sensorML.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified"
targetNamespace="http://www.w3.org/2005/08/addressing">
      <xsd:include schemaLocation="http://www.w3.org/2005/08/addressing/ws-addr.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified" targetNamespace="http://docs.oasis-
open.org/wsrf/rl-2">
      <xsd:include schemaLocation="http://docs.oasis-open.org/wsrf/rl-2.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified" targetNamespace="http://docs.oasis-
open.org/wsrf/rp-2">
      <xsd:include schemaLocation="http://docs.oasis-open.org/wsrf/rp-2.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified" targetNamespace="http://docs.oasis-
open.org/wsrf/r-2">
      <xsd:include schemaLocation="http://docs.oasis-open.org/wsrf/r-2.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified" targetNamespace="http://docs.oasis-
open.org/wsn/b-2">
      <xsd:include schemaLocation="http://docs.oasis-open.org/wsn/b-2.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified" targetNamespace="http://docs.oasis-
open.org/wsn/t-1">
      <xsd:include schemaLocation="http://docs.oasis-open.org/wsn/t-1.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified" targetNamespace="http://docs.oasis-
open.org/wsn/br-2">
      <xsd:include schemaLocation="http://docs.oasis-open.org/wsn/br-2.xsd"/>
    </xsd:schema>
    <xsd:schema elementFormDefault="qualified" targetNamespace="http://www.igsi.eu/sps">
      <xsd:element name="ServiceResourceProperties">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="wsn-b:FixedTopicSet"/>
            <xsd:element ref="wsn-t:TopicSet" minOccurs="0"/>
            <xsd:element ref="wsn-b:TopicExpression" minOccurs="0"
maxOccurs="unbounded"/>
            <xsd:element ref="wsn-b:TopicExpressionDialect" minOccurs="0"
maxOccurs="unbounded"/>
             <xsd:element ref="wsn-br:RequiresRegistration"/>
            <!-- service specific properties could be added, like the service
capabilities -->
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:complexType name="ServiceExceptionType">
        <xsd:complexContent>
          <xsd:extension base="wsrf-bf:BaseFaultType">
            <xsd:choice>
              <xsd:element ref="ows:ExceptionReport"/>
              <xsd:element ref="ows:Exception"/>
            </xsd:choice>
          </xsd:extension>
        </xsd:complexContent>
      </xsd:complexType>
      <xsd:element name="ServiceException" type="igsi:ServiceExceptionType"/>
    </xsd:schema>
  </wsdl:types>
```

```
<wsdl:message name="DestroyRequestMessage">
  <wsdl:part name="body" element="wsrf-rl:Destroy"/>
</wsdl:message>
<wsdl:message name="DestroyResponseMessage">
  <wsdl:part name="body" element="wsrf-rl:DestroyResponse"/>
</wsdl:message>
<wsdl:message name="ResourceNotDestroyedFaultMessage">
  <wsdl:part name="fault" element="wsrf-rl:ResourceNotDestroyedFault"/>
</wsdl:message>
<wsdl:message name="ResourceUnknownFaultMessage">
  <wsdl:part name="fault" element="wsrf-r:ResourceUnknownFault"/>
</wsdl:message>
<wsdl:message name="ResourceUnavailableFaultMessage">
  <wsdl:part name="fault" element="wsrf-r:ResourceUnavailableFault"/>
</wsdl:message>
<wsdl:message name="SetTerminationTimeRequestMessage">
  <wsdl:part name="body" element="wsrf-rl:SetTerminationTime"/>
</wsdl:message>
<wsdl:message name="SetTerminationTimeResponseMessage">
  <wsdl:part name="body" element="wsrf-rl:SetTerminationTimeResponse"/>
</wsdl:message>
<wsdl:message name="UnableToSetTerminationTimeFaultMessage">
  <wsdl:part name="fault" element="wsrf-rl:UnableToSetTerminationTimeFault"/>
</wsdl:message>
<wsdl:message name="TerminationTimeChangeRejectedFaultMessage">
  <wsdl:part name="fault" element="wsrf-rl:TerminationTimeChangeRejectedFault"/>
</wsdl:message>
<wsdl:message name="GetResourcePropertyDocumentRequestMessage">
  <wsdl:part name="body" element="wsrf-rp:GetResourcePropertyDocument"/>
</wsdl:message>
<wsdl:message name="GetResourcePropertyDocumentResponseMessage">
  <wsdl:part name="body" element="wsrf-rp:GetResourcePropertyDocumentResponse"/>
</wsdl:message>
<wsdl:message name="GetResourcePropertyRequestMessageMessage">
  <wsdl:part name="body" element="wsrf-rp:GetResourceProperty"/>
</wsdl:message>
<wsdl:message name="GetResourcePropertyResponseMessage">
  <wsdl:part name="body" element="wsrf-rp:GetResourcePropertyResponse"/>
</wsdl:message>
<wsdl:message name="InvalidResourcePropertyQNameFaultMessage">
  <wsdl:part name="fault" element="wsrf-rp:InvalidResourcePropertyQNameFault"/>
</wsdl:message>
<wsdl:message name="GetMultipleResourcePropertiesRequestMessage">
  <wsdl:part name="body" element="wsrf-rp:GetMultipleResourceProperties"/>
</wsdl:message>
<wsdl:message name="GetMultipleResourcePropertiesResponseMessage">
  <wsdl:part name="body" element="wsrf-rp:GetMultipleResourcePropertiesResponse"/>
</wsdl:message>
<wsdl:message name="QueryResourcePropertiesRequestMessage">
  <wsdl:part name="body" element="wsrf-rp:QueryResourceProperties"/>
</wsdl:message>
<wsdl:message name="QueryResourcePropertiesResponseMessage">
  <wsdl:part name="body" element="wsrf-rp:QueryResourcePropertiesResponse"/>
</wsdl:message>
<wsdl:message name="UnknownQueryExpressionDialectFaultMessage">
  <wsdl:part name="fault" element="wsrf-rp:UnknownQueryExpressionDialectFault"/>
</wsdl:message>
<wsdl:message name="InvalidQueryExpressionFaultMessage">
  <wsdl:part name="fault" element="wsrf-rp:InvalidQueryExpressionFault"/>
</wsdl:message>
<wsdl:message name="QueryEvaluationErrorFaultMessage">
  <wsdl:part name="fault" element="wsrf-rp:QueryEvaluationErrorFault"/>
</wsdl:message>
<wsdl:message name="SetResourcePropertiesRequestMessage">
  <wsdl:part name="body" element="wsrf-rp:SetResourceProperties"/>
</wsdl:message>
<wsdl:message name="SetResourcePropertiesResponseMessage">
  <wsdl:part name="body" element="wsrf-rp:SetResourcePropertiesResponse"/>
</wsdl:message>
<wsdl:message name="InvalidModificationFaultMessage">
  <wsdl:part name="fault" element="wsrf-rp:InvalidModificationFault"/>
```



```
</wsdl:message>
<wsdl:message name="UnableToModifyResourcePropertyFaultMessage">
  <wsdl:part name="fault" element="wsrf-rp:UnableToModifyResourcePropertyFault"/>
</wsdl:message>
<wsdl:message name="SetResourcePropertyRequestFailedFault">
  <wsdl:part name="fault" element="wsrf-rp:SetResourcePropertyRequestFailedFault"/>
</wsdl:message>
<wsdl:message name="SubscribeRequestMessage">
  <wsdl:part name="body" element="wsn-b:Subscribe"/>
</wsdl:message>
<wsdl:message name="SubscribeResponseMessage">
  <wsdl:part name="body" element="wsn-b:SubscribeResponse"/>
</wsdl:message>
<wsdl:message name="SubscribeCreationFailedFaultMessage">
  <wsdl:part name="fault" element="wsn-b:SubscribeCreationFailedFault"/>
</wsdl:message>
<wsdl:message name="TopicExpressionDialectUnknownFaultMessage">
  <wsdl:part name="fault" element="wsn-b:TopicExpressionDialectUnknownFault"/>
</wsdl:message>
<wsdl:message name="InvalidFilterFaultMessage">
  <wsdl:part name="fault" element="wsn-b:InvalidFilterFaultMessage"/>
</wsdl:message>
<wsdl:message name="InvalidProducerPropertiesExpressionFaultMessage">
  <wsdl:part name="fault" element="wsn-b:InvalidProducerPropertiesExpressionFault"/>
</wsdl:message>
<wsdl:message name="InvalidMessageContentExpressionFault">
  <wsdl:part name="fault" element="wsn-b:InvalidMessageContentExpressionFault"/>
</wsdl:message>
<wsdl:message name="UnrecognizedPolicyRequestFaultMessage">
  <wsdl:part name="fault" element="wsn-b:UnrecognizedPolicyRequestFault"/>
</wsdl:message>
<wsdl:message name="UnsupportedPolicyRequestFaultMessage">
  <wsdl:part name="fault" element="wsn-b:UnsupportedPolicyRequestFault"/>
</wsdl:message>
<wsdl:message name="NotifyMessageNotSupportedFaultMessage">
  <wsdl:part name="fault" element="wsn-b:NotifyMessageNotSupportedFault"/>
</wsdl:message>
<wsdl:message name="UnacceptableInitialTerminationTimeFaultMessage">
  <wsdl:part name="fault" element="wsn-b:UnacceptableInitialTerminationTimeFault"/>
</wsdl:message>
<wsdl:message name="GetCurrentMessageRequestMessage">
  <wsdl:part name="body" element="wsn-b:GetCurrentMessage"/>
</wsdl:message>
<wsdl:message name="GetCurrentMessageResponseMessage">
  <wsdl:part name="body" element="wsn-b:GetCurrentMessageResponse"/>
</wsdl:message>
<wsdl:message name="InvalidTopicExpressionFaultMessage">
  <wsdl:part name="fault" element="wsn-b:InvalidTopicExpressionFault"/>
</wsdl:message>
<wsdl:message name="TopicNotSupportedFaultMessage">
  <wsdl:part name="fault" element="wsn-b:TopicNotSupportedFault"/>
</wsdl:message>
<wsdl:message name="MultipleTopicsSpecifiedFaultMessage">
  <wsdl:part name="fault" element="wsn-b:MultipleTopicsSpecifiedFault"/>
</wsdl:message>
<wsdl:message name="NoCurrentMessageOnTopicFaultMessage">
  <wsdl:part name="fault" element="wsn-b:NoCurrentMessageOnTopicFault"/>
</wsdl:message>
<wsdl:message name="NotifyMessage">
  <wsdl:part name="body" element="wsn-b:Notify"/>
</wsdl:message>
<wsdl:message name="RegisterPublisherRequestMessage">
  <wsdl:part name="body" element="wsn-br:RegisterPublisher"/>
</wsdl:message>
<wsdl:message name="RegisterPublisherResponseMessage">
  <wsdl:part name="body" element="wsn-br:RegisterPublisherResponse"/>
</wsdl:message>

<!-- ============= MESSAGES ============= -->
<!-- SPS specific messages -->
<wsdl:message name="GetCapabilitiesRequestMessage">
  <wsdl:part name="body" element="sps:GetCapabilities"/>
```

```
  </wsdl:message>
  <wsdl:message name="GetCapabilitiesResponseMessage">
    <wsdl:part name="body" element="sps:Capabilities"/>
  </wsdl:message>
  <wsdl:message name="DescribeSensorRequestMessage">
    <wsdl:part name="body" element="swes:DescribeSensor"/>
  </wsdl:message>
  <wsdl:message name="DescribeSensorResponseMessage">
    <wsdl:part name="body" element="sml:SensorML"/>
  </wsdl:message>
  <wsdl:message name="ServiceExceptionMessage">
    <wsdl:part name="fault" type="igsi:ServiceException"/>
  </wsdl:message>


  <!-- ================================================== -->
  <!-- ==================== igsi PortType =================== -->
  <!-- ================================================== -->
  <wsdl:portType name="SpsPortType" wsrf-
rp:ResourceProperties="igsi:ServiceResourceProperties">

    <!-- ============= implements GetCapabilities ============= -->
    <wsdl:operation name="GetCapabilities">
      <wsdl:input wsa:Action="http://www.opengis.net/sps/2.0/GetCapabilitiesRequest"
message="sps:GetCapabilitiesRequestMessage"/>
      <wsdl:output wsa:Action="http://www.opengis.net/sps/2.0/GetCapabilitiesResponse"
message="sps:GetCapabilitiesResponseMessage"/>
      <wsdl:fault name="Exception" message="igsi:ServiceExceptionMessage"/>
    </wsdl:operation>

    <!-- ============= implements DescribeSensor  ============= -->
    <wsdl:operation name="DescribeSensor">
      <wsdl:input wsa:Action="http://www.opengis.net/swes/1.0/DescribeSensorRequest"
message="swes:DescribeSensorRequestMessage"/>
      <wsdl:output wsa:Action="http://www.opengis.net/swes/1.0/DescribeSensorResponse"
message="swes:DescribeSensorResponseMessage"/>
      <wsdl:fault name="Exception" message="igsi:ServiceExceptionMessage"/>
    </wsdl:operation>
    <!-- ============= realizes NotificationConsumer ============= -->
    <wsdl:operation name="Notify">
      <wsdl:input wsa:Action="http://docs.oasis-open.org/wsn/bw-
2/NotificationConsumer/Notify" message="igsi:NotifyMessage"/>
    </wsdl:operation>
    <!-- ============= realizes NotificationProducer ============= -->
    <wsdl:operation name="Subscribe">
      <wsdl:input wsa:Action="http://docs.oasis-open.org/wsn/bw-
2/NotificationProducer/SubscribeRequest" message="igsi:SubscribeRequestMessage"/>
      <wsdl:output wsa:Action="http://docs.oasis-open.org/wsn/bw-
2/NotificationProducer/SubscribeResponse" message="igsi:SubscribeResponseMessage"/>
      <wsdl:fault name="ResourceUnknownFault"
message="igsi:ResourceUnknownFaultMessage"/>
      <wsdl:fault name="InvalidFilterFault" message="igsi:InvalidFilterFaultMessage"/>
      <wsdl:fault name="TopicExpressionDialectUnknownFault"
message="igsi:TopicExpressionDialectUnknownFaultMessage"/>
      <wsdl:fault name="InvalidTopicExpressionFault"
message="igsi:InvalidTopicExpressionFaultMessage"/>
      <wsdl:fault name="TopicNotSupportedFault"
message="igsi:TopicNotSupportedFaultMessage"/>
      <wsdl:fault name="InvalidProducerPropertiesExpressionFault"
message="igsi:InvalidProducerPropertiesExpressionFaultMessage"/>
      <wsdl:fault name="InvalidMessageContentExpressionFault"
message="igsi:InvalidMessageContentExpressionFaultMessage"/>
      <wsdl:fault name="UnacceptableInitialTerminationTimeFault"
message="igsi:UnacceptableInitialTerminationTimeFaultMessage"/>
      <wsdl:fault name="UnrecognizedPolicyRequestFault"
message="igsi:UnrecognizedPolicyRequestFaultMessage"/>
      <wsdl:fault name="UnsupportedPolicyRequestFault"
message="igsi:UnsupportedPolicyRequestFaultMessage"/>
      <wsdl:fault name="NotifyMessageNotSupportedFault"
message="igsi:NotifyMessageNotSupportedFaultMessage"/>
      <wsdl:fault name="SubscribeCreationFailedFault"
message="igsi:SubscribeCreationFailedFaultMessage"/>
```



```
    </wsdl:operation>
    <wsdl:operation name="GetCurrentMessage">
      <wsdl:input wsa:Action="http://docs.oasis-open.org/wsn/bw-
2/NotificationProducer/GetCurrentMessageRequest"
message="igsi:GetCurrentMessageRequestMessage"/>
      <wsdl:output wsa:Action="http://docs.oasis-open.org/wsn/bw-
2/NotificationProducer/GetCurrentMessageResponse"
message="igsi:GetCurrentMessageResponseMessage"/>
      <wsdl:fault name="ResourceUnknownFault"
message="igsi:ResourceUnknownFaultMessage"/>
      <wsdl:fault name="TopicExpressionDialectUnknownFault"
message="igsi:TopicExpressionDialectUnknownFaultMessage"/>
      <wsdl:fault name="InvalidTopicExpressionFault"
message="igsi:InvalidTopicExpressionFaultMessage"/>
      <wsdl:fault name="TopicNotSupportedFault"
message="igsi:TopicNotSupportedFaultMessage"/>
      <wsdl:fault name="NoCurrentMessageOnTopicFault"
message="igsi:NoCurrentMessageOnTopicFaultMessage"/>
      <wsdl:fault name="MultipleTopicsSpecifiedFault"
message="igsi:MultipleTopicsSpecifiedFaultMessage"/>
    </wsdl:operation>

    <!-- ========= realizes RegisterPublisher ======= -->
    <wsdl:operation name="RegisterPublisher">
      <wsdl:input wsa:Action="http://docs.oasis-open.org/wsn/brw-
2/RegisterPublisher/RegisterPublisherRequest"
message="igsi:RegisterPublisherRequestMessage"/>
      <wsdl:output wsa:Action="http://docs.oasis-open.org/wsn/brw-
2/RegisterPublisher/RegisterPublisherResponse"
message="igsi:RegisterPublisherResponseMessage"/>
      <wsdl:fault name="ResourceUnknownFault"
message="igsi:ResourceUnknownFaultMessage"/>
      <wsdl:fault name="InvalidTopicExpressionFault"
message="igsi:InvalidTopicExpressionFaultMessage"/>
      <wsdl:fault name="TopicNotSupportedFault"
message="igsi:TopicNotSupportedFaultMessage"/>
      <wsdl:fault name="PublisherRegistrationRejectedFault"
message="igsi:PublisherRegistrationRejectedFaultMessage"/>
      <wsdl:fault name="PublisherRegistrationFailedFault"
message="igsi:PublisherRegistrationFailedFaultMessage"/>
      <wsdl:fault name="UnacceptableInitialTerminationTimeFault"
message="igsi:UnacceptableInitialTerminationTimeFaultMessage"/>
    </wsdl:operation>

    <!-- =========== realizes some WS-ResourceProperties operations ========== -->
    <wsdl:operation name="GetResourcePropertyDocument">
      <wsdl:input wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/GetResourcePropertyDocument/GetResourcePropertyDocumentRequest"
name="GetResourcePropertyDocumentRequest"
message="igsi:GetResourcePropertyDocumentRequestMessage"/>
      <wsdl:output wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/GetResourcePropertyDocument/GetResourcePropertyDocumentResponse"
name="GetResourcePropertyDocumentResponse"
message="igsi:GetResourcePropertyDocumentResponseMessage"/>
      <wsdl:fault name="ResourceUnknownFault"
message="igsi:ResourceUnknownFaultMessage"/>
      <wsdl:fault name="ResourceUnavailableFault"
message="igsi:ResourceUnavailableFaultMessage"/>
    </wsdl:operation>

    <wsdl:operation name="GetResourceProperty">
      <wsdl:input wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/GetResourceProperty/GetResourcePropertyRequest" name="GetResourcePropertyRequest"
message="igsi:GetResourcePropertyRequestMessage"/>
      <wsdl:output wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/GetResourceProperty/GetResourcePropertyResponse" name="GetResourcePropertyResponse"
message="igsi:GetResourcePropertyResponseMessage"/>
      <wsdl:fault name="ResourceUnknownFault"
message="igsi:ResourceUnknownFaultMessage"/>
      <wsdl:fault name="ResourceUnavailableFault"
message="igsi:ResourceUnavailableFaultMessage"/>
```



```
      <wsdl:fault name="InvalidResourcePropertyQNameFault"
message="igsi:InvalidResourcePropertyQNameFaultMessage"/>
    </wsdl:operation>

    <wsdl:operation name="GetMultipleResourceProperties">
      <wsdl:input wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/GetMultipleResourceProperties/GetMultipleResourcePropertiesRequest"
name="GetMultipleResourcePropertiesRequest"
message="igsi:GetMultipleResourcePropertiesRequestMessage"/>
      <wsdl:output wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/GetMultipleResourceProperties/GetMultipleResourcePropertiesResponse"
name="GetMultipleResourcePropertiesResponse"
message="igsi:GetMultipleResourcePropertiesResponseMessage"/>
      <wsdl:fault name="ResourceUnknownFault"
message="igsi:ResourceUnknownFaultMessage"/>
      <wsdl:fault name="ResourceUnavailableFault"
message="igsi:ResourceUnavailableFaultMessage"/>
      <wsdl:fault name="InvalidResourcePropertyQNameFault"
message="igsi:InvalidResourcePropertyQNameFaultMessage"/>
    </wsdl:operation>

    <wsdl:operation name="QueryResourceProperties">
      <wsdl:input wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/QueryResourceProperties/QueryResourcePropertiesRequest"
name="QueryResourcePropertiesRequest"
message="igsi:QueryResourcePropertiesRequestMessage"/>
      <wsdl:output wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/QueryResourceProperties/QueryResourcePropertiesResponse"
name="QueryResourcePropertiesResponse"
message="igsi:QueryResourcePropertiesResponseMessage"/>
      <wsdl:fault name="ResourceUnknownFault"
message="igsi:ResourceUnknownFaultMessage"/>
      <wsdl:fault name="ResourceUnavailableFault"
message="igsi:ResourceUnavailableFaultMessage"/>
      <wsdl:fault name="UnknownQueryExpressionDialectFault"
message="igsi:UnknownQueryExpressionDialectFaultMessage"/>
      <wsdl:fault name="InvalidQueryExpressionFault"
message="igsi:InvalidQueryExpressionFaultMessage"/>
      <wsdl:fault name="QueryEvaluationErrorFault"
message="igsi:QueryEvaluationErrorFaultMessage"/>
    </wsdl:operation>

    <wsdl:operation name="SetResourceProperties">
      <wsdl:input wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/SetResourceProperties/SetResourcePropertiesRequest" name="SetResourcePropertiesRequest"
message="igsi:SetResourcePropertiesRequestMessage"/>
      <wsdl:output wsa:Action="http://docs.oasis-open.org/wsrf/rpw-
2/SetResourceProperties/SetResourcePropertiesResponse"
name="SetResourcePropertiesResponse"
message="igsi:SetResourcePropertiesResponseMessage"/>
      <wsdl:fault name="ResourceUnknownFault"
message="igsi:ResourceUnknownFaultMessage"/>
      <wsdl:fault name="ResourceUnavailableFault"
message="igsi:ResourceUnavailableFaultMessage"/>
      <wsdl:fault name="InvalidModificationFault"
message="igsi:InvalidModificationFaultMessage"/>
      <wsdl:fault name="UnableToModifyResourcePropertyFault"
message="igsi:UnableToModifyResourcePropertyFaultMessage"/>
      <wsdl:fault name="InvalidResourcePropertyQNameFault"
message="igsi:InvalidResourcePropertyQNameFaultMessage"/>
      <wsdl:fault name="SetResourcePropertyRequestFailedFault"
message="igsi:SetResourcePropertyRequestFailedFaultMessage"/>
    </wsdl:operation>
  </wsdl:portType>


  <!-- ============================ ======================== -->
  <!-- ===================== BINDING ======================== -->
  <!-- ============================================================ -->
  <wsdl:binding name="SpsBinding" type="igsi:SpsPortType">
    <wsdl-soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
```



```
    <!-- use of WS-Addressing, especially of the wsa:Action header in SOAP messages, is
required -->
    <wsa:UsingAddressing wsdl:required="true" />

    <!-- Subject requires WS-Addressing and requires the use of non-anonymous response
EPRs -->
    <wsp:Policy>
        <wsa:Addressing>
            <wsp:Policy>
                <wsa:NonAnonymousResponses/>
            </wsp:Policy>
        </wsa:Addressing>
    </wsp:Policy>

    <wsdl:operation name="GetCapabilities">
      <wsdl-soap12:operation/>
      <wsdl:input>
        <wsdl-soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <wsdl-soap12:body use="literal" />
      </wsdl:output>
      <wsdl:fault name="Exception">
        <wsdl-soap12:fault use="literal" name="Exception"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="DescribeSensor">
      <wsdl-soap12:operation/>
      <wsdl:input>
        <wsdl-soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <wsdl-soap12:body use="literal" />
      </wsdl:output>
      <wsdl:fault name="Exception">
        <wsdl-soap12:fault use="literal" name="Exception"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="Notify">
      <wsdl-soap12:operation/>
      <wsdl:input>
        <wsdl-soap12:body use="literal" />
      </wsdl:input>
    </wsdl:operation>

    <wsdl:operation name="Subscribe">
      <wsdl-soap12:operation/>
      <wsdl:input>
        <wsdl-soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <wsdl-soap12:body use="literal" />
      </wsdl:output>
      <wsdl:fault name="ResourceUnknownFault">
        <wsdl-soap12:fault use="literal" name="ResourceUnknownFault"/>
      </wsdl:fault>
      <wsdl:fault name="InvalidFilterFault">
        <wsdl-soap12:fault use="literal" name="InvalidFilterFault"/>
      </wsdl:fault>
      <wsdl:fault name="TopicExpressionDialectUnknownFault">
        <wsdl-soap12:fault use="literal" name="TopicExpressionDialectUnknownFault"/>
      </wsdl:fault>
      <wsdl:fault name="InvalidTopicExpressionFault">
        <wsdl-soap12:fault use="literal" name="InvalidTopicExpressionFault"/>
      </wsdl:fault>
      <wsdl:fault name="TopicNotSupportedFault">
        <wsdl-soap12:fault use="literal" name="TopicNotSupportedFault"/>
      </wsdl:fault>
      <wsdl:fault name="InvalidProducerPropertiesExpressionFault">
```



```
        <wsdl-soap12:fault use="literal"
name="InvalidProducerPropertiesExpressionFault"/>
      </wsdl:fault>
      <wsdl:fault name="InvalidMessageContentExpressionFault">
        <wsdl-soap12:fault use="literal" name="InvalidMessageContentExpressionFault"/>
      </wsdl:fault>
      <wsdl:fault name="UnacceptableInitialTerminationTimeFault">
        <wsdl-soap12:fault use="literal" name="UnacceptableInitialTerminationTimeFault"/>
      </wsdl:fault>
      <wsdl:fault name="UnrecognizedPolicyRequestFault">
        <wsdl-soap12:fault use="literal" name="UnrecognizedPolicyRequestFault"/>
      </wsdl:fault>
      <wsdl:fault name="UnsupportedPolicyRequestFault">
        <wsdl-soap12:fault use="literal" name="UnsupportedPolicyRequestFault"/>
      </wsdl:fault>
      <wsdl:fault name="NotifyMessageNotSupportedFault">
        <wsdl-soap12:fault use="literal" name="NotifyMessageNotSupportedFault"/>
      </wsdl:fault>
      <wsdl:fault name="SubscribeCreationFailedFault">
        <wsdl-soap12:fault use="literal" name="SubscribeCreationFailedFault"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="GetCurrentMessage">
      <wsdl-soap12:operation/>
      <wsdl:input>
        <wsdl-soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <wsdl-soap12:body use="literal" />
      </wsdl:output>
      <wsdl:fault name="ResourceUnknownFault">
        <wsdl-soap12:fault use="literal" name="ResourceUnknownFault"/>
      </wsdl:fault>
      <wsdl:fault name="TopicExpressionDialectUnknownFault">
        <wsdl-soap12:fault use="literal" name="TopicExpressionDialectUnknownFault"/>
      </wsdl:fault>
      <wsdl:fault name="InvalidTopicExpressionFault">
        <wsdl-soap12:fault use="literal" name="InvalidTopicExpressionFault"/>
      </wsdl:fault>
      <wsdl:fault name="TopicNotSupportedFault">
        <wsdl-soap12:fault use="literal" name="TopicNotSupportedFault"/>
      </wsdl:fault>
      <wsdl:fault name="NoCurrentMessageOnTopicFault">
        <wsdl-soap12:fault use="literal" name="NoCurrentMessageOnTopicFault"/>
      </wsdl:fault>
      <wsdl:fault name="MultipleTopicsSpecifiedFault">
        <wsdl-soap12:fault use="literal" name="MultipleTopicsSpecifiedFault"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="RegisterPublisher">
      <wsdl-soap12:operation/>
      <wsdl:input>
        <wsdl-soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <wsdl-soap12:body use="literal" />
      </wsdl:output>
      <wsdl:fault name="ResourceUnknownFault">
        <wsdl-soap12:fault use="literal" name="ResourceUnknownFault"/>
      </wsdl:fault>
      <wsdl:fault name="InvalidTopicExpressionFault">
        <wsdl-soap12:fault use="literal" name="InvalidTopicExpressionFault"/>
      </wsdl:fault>
      <wsdl:fault name="TopicNotSupportedFault">
        <wsdl-soap12:fault use="literal" name="TopicNotSupportedFault"/>
      </wsdl:fault>
      <wsdl:fault name="PublisherRegistrationRejectedFault">
        <wsdl-soap12:fault use="literal" name="PublisherRegistrationRejectedFault"/>
      </wsdl:fault>
      <wsdl:fault name="PublisherRegistrationFailedFault">
```



```
    <wsdl-soap12:fault use="literal" name="PublisherRegistrationFailedFault"/>
  </wsdl:fault>
  <wsdl:fault name="UnacceptableInitialTerminationTimeFault">
    <wsdl-soap12:fault use="literal" name="UnacceptableInitialTerminationTimeFault"/>
  </wsdl:fault>
</wsdl:operation>

<wsdl:operation name="GetResourcePropertyDocument">
  <wsdl-soap12:operation/>
  <wsdl:input name="GetResourcePropertyDocumentRequest">
    <wsdl-soap12:body use="literal" />
  </wsdl:input>
  <wsdl:output name="GetResourcePropertyDocumentResponse">
    <wsdl-soap12:body use="literal" />
  </wsdl:output>
  <wsdl:fault name="ResourceUnknownFault">
    <wsdl-soap12:fault use="literal" name="ResourceUnknownFault"/>
  </wsdl:fault>
  <wsdl:fault name="ResourceUnavailableFault">
    <wsdl-soap12:fault use="literal" name="ResourceUnavailableFault"/>
  </wsdl:fault>
</wsdl:operation>

<wsdl:operation name="GetResourceProperty">
  <wsdl-soap12:operation/>
  <wsdl:input name="GetResourcePropertyRequest">
    <wsdl-soap12:body use="literal" />
  </wsdl:input>
  <wsdl:output name="GetResourcePropertyResponse">
    <wsdl-soap12:body use="literal" />
  </wsdl:output>
  <wsdl:fault name="ResourceUnknownFault">
    <wsdl-soap12:fault use="literal" name="ResourceUnknownFault"/>
  </wsdl:fault>
  <wsdl:fault name="ResourceUnavailableFault">
    <wsdl-soap12:fault use="literal" name="ResourceUnavailableFault"/>
  </wsdl:fault>
  <wsdl:fault name="InvalidResourcePropertyQNameFault">
    <wsdl-soap12:fault use="literal" name="InvalidResourcePropertyQNameFault"/>
  </wsdl:fault>
</wsdl:operation>

<wsdl:operation name="GetMultipleResourceProperties">
  <wsdl-soap12:operation/>
  <wsdl:input name="GetMultipleResourcePropertiesRequest">
    <wsdl-soap12:body use="literal" />
  </wsdl:input>
  <wsdl:output name="GetMultipleResourcePropertiesResponse">
    <wsdl-soap12:body use="literal" />
  </wsdl:output>
  <wsdl:fault name="ResourceUnknownFault">
    <wsdl-soap12:fault use="literal" name="ResourceUnknownFault"/>
  </wsdl:fault>
  <wsdl:fault name="ResourceUnavailableFault">
    <wsdl-soap12:fault use="literal" name="ResourceUnavailableFault"/>
  </wsdl:fault>
  <wsdl:fault name="InvalidResourcePropertyQNameFault">
    <wsdl-soap12:fault use="literal" name="InvalidResourcePropertyQNameFault"/>
  </wsdl:fault>
</wsdl:operation>

<wsdl:operation name="QueryResourceProperties">
  <wsdl-soap12:operation/>
  <wsdl:input name="QueryResourcePropertiesRequest">
    <wsdl-soap12:body use="literal" />
  </wsdl:input>
  <wsdl:output name="QueryResourcePropertiesResponse">
    <wsdl-soap12:body use="literal" />
  </wsdl:output>
  <wsdl:fault name="ResourceUnknownFault">
    <wsdl-soap12:fault use="literal" name="ResourceUnknownFault"/>
  </wsdl:fault>
```

```
      <wsdl:fault name="ResourceUnavailableFault">
        <wsdl-soap12:fault use="literal" name="ResourceUnavailableFault"/>
      </wsdl:fault>
      <wsdl:fault name="UnknownQueryExpressionDialectFault">
        <wsdl-soap12:fault use="literal" name="UnknownQueryExpressionDialectFault"/>
      </wsdl:fault>
      <wsdl:fault name="InvalidQueryExpressionFault">
        <wsdl-soap12:fault use="literal" name="InvalidQueryExpressionFault"/>
      </wsdl:fault>
      <wsdl:fault name="QueryEvaluationErrorFault">
        <wsdl-soap12:fault use="literal" name="QueryEvaluationErrorFault"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="SetResourceProperties">
      <wsdl-soap12:operation/>
      <wsdl:input name="SetResourcePropertiesRequest">
        <wsdl-soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output name="SetResourcePropertiesResponse">
        <wsdl-soap12:body use="literal" />
      </wsdl:output>
      <wsdl:fault name="ResourceUnknownFault">
        <wsdl-soap12:fault use="literal" name="ResourceUnknownFault"/>
      </wsdl:fault>
      <wsdl:fault name="ResourceUnavailableFault">
        <wsdl-soap12:fault use="literal" name="ResourceUnavailableFault"/>
      </wsdl:fault>
      <wsdl:fault name="InvalidModificationFault">
        <wsdl-soap12:fault use="literal" name="InvalidModificationFault"/>
      </wsdl:fault>
      <wsdl:fault name="UnableToModifyResourcePropertyFault">
        <wsdl-soap12:fault use="literal" name="UnableToModifyResourcePropertyFault"/>
      </wsdl:fault>
      <wsdl:fault name="InvalidResourcePropertyQNameFault">
        <wsdl-soap12:fault use="literal" name="InvalidResourcePropertyQNameFault"/>
      </wsdl:fault>
      <wsdl:fault name="SetResourcePropertyRequestFailedFault">
        <wsdl-soap12:fault use="literal" name="SetResourcePropertyRequestFailedFault"/>
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="igsiSpsService">
    <wsdl:port name="igsiSpsPort" binding="igsi:SpsBinding">
      <wsdl-soap12:address location="http://www.igsi.eu:8080/services/sps"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```


Bibliography

Allen, J. F., Ferguson, G. (1994), *Actions and Events in Interval Temporal Logic.* Available at: https://dspace.lib.rochester.edu/retrieve/1531/94.tr521.Actions_and_events_in_interval_temporal_logic.ps downloaded on March 19th 2009.

Chandy, K. M., Schulte, W. R. (2007a), *What is Event Driven Architecture (EDA) and Why Does it Matter?* Available at: http://complexevents.com/?p=212 downloaded on February 25th 2008.

Chandy, K. M., Schulte, W. R. (2007b), *How Event Processing Improves Business Decision Making.* Available at: http://complexevents.com/?p=227 downloaded on February 25th 2008.

Chappell, D. A. (2004), *Enterprise Service Bus.* O'Reilly, Sebastopol, USA.

Faison, T. (2006), *Event-Based Programming: Taking Events to the Limit.* Apress, Berkley USA.

Fidge, C. J. (1988), *Timestamps in message-passing systems that preserve the partial ordering.* In Proceedings of the 11th Australian Computer Science Conference (ACSC'88), pp. 56-66, February 1988.

Galton, Worboys (2005), Processes and Events in Dynamic Geo-Networks. Available at: http://www.springerlink.com/content/lg3026t886qt3265/ downloaded on March 19th 2009.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns.* Addison-Wesley, Reading, MA, 1995.

Gruber, T. (2007), *Ontology.* Available at: http://tomgruber.org/writing/ontology-definition-2007.htm downloaded on March 19th 2009. (To appear in the *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.), Springer-Verlag, 2008).

Heinbockel, W. J. et al. (2007), *Common Event Expression.* MITRE white paper, available at: http://cee.mitre.org/docs/Common-Event-Expression-White-Paper.pdf downloaded on February 13th, 2009.

Lamport, L. (1978), *Time, clocks, and the ordering of events in a distributed system.* In: Communications of the ACM, Volume 21, Number 7, July 1978.

Luckham, D. (2002), *The Power of Events.* Addison-Wesley, Boston, USA.

Luckham, D. (2006), *What's the Difference Between ESP and CEP?* Available at: http://complexevents.com/?p=103 downloaded on February 22nd 2008

Luckham, D. (2007), *A Brief Overview of the Concepts of CEP*. Available at: http://complexevents.com/wp-content/uploads/2008/07/overview-of-concepts-of-cep.pdf downloaded on March 19th 2009

Luckham, D., Schulte, R. (2008), *Event Processing Glossary - Version 1.1.* Available at: http://www.ep-ts.com/component/option,com_docman/task,doc_download/gid,66/Itemid,84/ downloaded on February 13th, 2009.

Mühl, G. et al. (2006), *Distributed Event-Based Systems.* Springer, Secaucus, USA.

Niblett, P., Graham, S. (2005), *Events and Service-oriented architecture: The OASIS Web Service Notification specifications*, IBM System Journal, 44(4).

OASIS (2006a), *Web Services Notification (WSN), Version 1.3.* Available at: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws downloaded on March 18th 2009.

OASIS (2006b), *Web Services Resource 1.2*, Available at: http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf, downloaded on 7th of April 2009.

OASIS (2006c), *Web Services Resource Properties 1.2*, Available at: http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf, downloaded on 9th of April 2009.

OASIS (2006d), *Web Services Base Faults 1.2*, Available at: http://docs.oasis-open.org/wsrf/wsrf-ws_base_faults-1.2-spec-os.pdf, downloaded on 9th of April 2009.

OASIS (2006e), *Web Services Resource Lifetime 1.2*, Available at: http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-os.pdf, downloaded on 9th of April 2009.

OASIS (2006f), *Web Services Service Group 1.2*, Available at: http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-os.pdf, downloaded on 9th of April 2009.

OASIS (2006g), *WSRF Application Notes*, Available at: http://docs.oasis-open.org/wsrf/wsrf-application_notes-1.2-cd-02.pdf, downloaded on 7th of April 2009.

OASIS (2006h), *Web Services Resource Metadata 1.0*, Available at: http://docs.oasis-open.org/wsrf/wsrf-ws_resource_metadata_descriptor-1.0-spec-cs-01.pdf, downloaded on 7th of April 2009.

OASIS (2009a), *Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.2*. Available at: http://docs.oasis-open.org/ws-rx/wsrm/v1.2/wsrm.html downloaded on March 11th 2009.

OASIS (2009b), *Web Services Reliable Messaging Policy Assertion (WS-RM Policy) Version 1.2.* Available at: http://docs.oasis-open.org/ws-rx/wsrmp/v1.2/wsrmp.html downloaded on March 11th 2009.

OGF (2006), *OGSA WSRF Basic Profile 1.0*, Available at: www.ogf.org/documents/GFD.72.pdf, accessed on 6th of April 2009.

OMG (2007), *OMG Unified Modeling Language (OMG UML) Superstructure, V2.1.2*. Available at: http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML downloaded on March 19th 2009.

Schäffer, B. (2008), *OWS 5 SOAP/WSDL Common Engineering Report*, Available at: http://portal.opengeospatial.org/files/?artifact_id=26521, accessed on 9th of April 2009

Schwarz, R., Mattern, F. (1994), *Detecting Causal Relationships in Distributed Computations: in Search of the Holy Grail.* In Distributed Computing, Volume 7, Number 3, Springer 1994.

Tilkov, S. (2008), *Addressing Doubts about REST.* Available at: http://www.infoq.com/articles/tilkov-rest-doubts downloaded on March 11th 2009.

W3C (2001), *Web Services Description Language (WSDL) 1.1*, available at: http://www.w3.org/TR/wsdl, accessed on April 14th, 2009

W3C (2004), *RDF Primer*, Available at: http://www.w3.org/TR/rdf-primer/, accessed on 14th of April, 2009

W3C (2006), *Web Services Addressing 1.0 - Core*, W3C Recommendation. Available at: http://www.w3.org/TR/ws-addr-core/ downloaded on March 18th 2009.

W3C (2007a), Web *Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts*, W3C Recommendation. Available at: http://www.w3.org/TR/wsdl20-adjuncts/ downloaded on March 18th 2009.

W3C (2007b), *Web Services Description Language (WSDL) Version 2.0: Additional MEPs*, W3C Recommendation. Available at: http://www.w3.org/TR/wsdl20-additional-meps/ downloaded on March 18th 2009.

W3C (2007c), *Web Services Addressing 1.0 - Metadata*, Available at: http://www.w3.org/TR/ws-addr-metadata/, downloaded on 6th of April 2009.

Worboys, M. (2005), *Event-oriented approaches to geographic phenomena.* International Journal of Geographical Information Science, 19:1 , pp.1 - 28. Available at: http://dx.doi.org/10.1080/13658810412331280167 downloaded on March 18th 2009.

Worboys, M., Hornsby, K. (2005), *From Objects to Events: GEM, the Geospatial Event Model.* Available at: http://www.springerlink.com/content/0ekce2p1rr2578dr/ downloaded on March 19th 2009.

WS-I (2006), *Basic Profile Version 1.1*, Available at: http://www.ws-i.org/Profiles/BasicProfile-1.1.html, accessed on 6th of April 2009

WS-I (2007), *Basic Profile Version 2.0*, Available at: http://www.ws-i.org/Profiles/BasicProfile-2_0(WGD).html, accessed on 8th of April 2009

