**Open Geospatial Consortium**

Submission Date: <yyyy-mm-dd>

Approval Date: <yyyy-mm-dd>

Publication Date: <2020-06-15>

External identifier of this OGC® document: http://www.opengis.net/doc/whitepaper/Pub-Sub

Internal reference number of this OGC® document: 20-081

Category: OGC® White Paper

Editor: Sara Saeedi

# OGC Publish-Subscribe White Paper



**Warning**

This document is not an OGC Standard. This document is an OGC White Paper and is therefore 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, an OGC White Paper should not be referenced as required or mandatory technology in procurements.

| | |
|---|---|
| Document type: | OGC® White Paper |
| Document subtype: | |
| Document stage: | Draft |
| Document language: | English |

## License Agreement

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD.

THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER'S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR's sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR standards or specifications. This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

# Table of Contents


1. References . . . 6
2. Terms and definitions . . . 7
   2.1. Abbreviated terms . . . 8
   2.2. UML notation . . . 9
3. Overview . . . 11
4. Asynchronous Operations . . . 13
   4.1. Behavior Models . . . 13
      4.1.1. Request-Response . . . 13
      4.1.2. Delayed Response . . . 14
      4.1.3. Standing Request . . . 14
      4.1.4. Synchronization . . . 15
      4.1.5. Publish-Subscribe . . . 15
      4.1.6. Broadcast . . . 16
   4.2. Notification and Alert . . . 16
      4.2.1. Callback . . . 17
      4.2.2. Polling . . . 17
      4.2.3. Stored response queue . . . 17
      4.2.4. Man in the Loop . . . 17
   4.3. Filtering . . . 17
      4.3.1. Event (RSS, SNMP) . . . 17
      4.3.2. Topic Hierarchy . . . 17
      4.3.3. Query expression (Standing Query) . . . 17
      4.3.4. Check Point . . . 18
5. OGC's PubSub Implementation Standard . . . 19
   5.1. Overview . . . 19
   5.2. OGC's PubSub Features . . . 19
   5.3. Related Work . . . 23
   5.4. Basic PubSub 1.0 extension for the generic OWS . . . 24
      5.4.1. Conceptual model . . . 24
      5.4.2. Required Capabilities components . . . 25
   5.5. Support to legacy components . . . 29
6. PubSub in OGC SensorThings . . . 32
   6.1. OGC IMIST IOT Pilot . . . 32
7. WMO OpenWIS use of AMQP . . . 34
8. Air Traffic Control Simulators . . . 37
9. AsyncAPI description . . . 40
   9.1. Concepts . . . 40
      9.1.1. Application . . . 40

9.1.2. Bindings . . . 40
9.1.3. Channel . . . 40
9.1.4. Consumer . . . 40
9.1.5. Message . . . 41
9.1.6. Producer . . . 41
9.1.7. Protocol . . . 41
9.2. The AsyncAPI Specification . . . 41
9.2.1. AsyncAPI Object . . . 41
9.2.2. Components Object . . . 42
9.2.3. Server Objects . . . 43
9.2.4. Channel Object . . . 44
9.2.5. Channel Item Object . . . 45
9.2.6. Operation Object . . . 46
9.2.7. Server Binding Object . . . 47
9.2.8. Channel Binding Object . . . 48
9.2.9. Message Binding Object . . . 48
9.2.10. Operation Binding Object . . . 48
9.2.11. Parameters . . . 48
9.2.12. Message Object . . . 49
9.3. Resources . . . 50
10. Future Work . . . 51
Annex A: Revision History . . . 52
Annex B: Bibliography . . . 53

## i. Abstract

<Insert Abstract Text here>

## ii. Keywords

The following are keywords to be used by search engines and document catalogues.

ogcdoc, OGC document, Asynchorous Patterns, Publication-Subscribe, Event-based Architecture.

## iii. Preface

**NOTE**

Insert Preface Text here. Give OGC specific commentary: describe the technical content, reason for document, history of the document and precursors, and plans for future work. > 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 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.

## iv. Document contributor contact points

The following organizations submitted this Document to the Open Geospatial Consortium (OGC). All questions regarding this submission should be directed to the editor or the submitters:

**Contacts**

| Name | Organization | Role |
|---|---|---|
| Sara Saeedi | University of Calgary | Editor |
| Chuck Heazel | Heazeltech LLC | Contributor |
| Don Sullivan | NOAA | Contributor |
| Lorenzo Bigagli | CNR | Contributor |
| Steve Liang | University of Calgary | Contributor |
| Teodor Hanchevici | Kongsberg Geospatial | Contributor |

# Chapter 1. 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.

- OGC 06-121r3, OGC® Web Services Common Specification, version 1.1.0 (9 February 2007)
- OGC 13-131, OGC® Publish/Subscribe Interface Standard 1.0 - Core
- OGC 13-133, OGC® Publish/Subscribe Interface Standard 1.0 - SOAP Protocol Binding Extension
- OGC 13-132, OGC® Publish/Subscribe Interface Standard 1.0 - RESTful Protocol Binding Extension (draft, in progress)
- OGC 07-006r1, OpenGIS® Catalogue Services Specification, version 2.0.2, corrigendum 2
- OGC 07-045, OpenGIS® Catalogue Services Specification 2.0.2 - ISO Metadata Application Profile, version 1.0
- OGC 07-110r4, CSW-ebRIM Registry Service - Part 1: ebRIM profile of CSW, version 1.0.1, corrigendum 1
- OGC 07-144r4, CSW-ebRIM Registry Service - Part 2: Basic extension package, version 1.0.1, corrigendum 1
- OGC 08-103r2, CSW-ebRIM Registry Service - Part 3: Abstract Test Suite, version 1.0.1
- OGC 16-137r2, Testbed-12 PubSub / Catalog Engineering Report (12 May 2017)
- OASIS international open standards consortium MQTT Version 5.0
- OASIS international open standards consortium ODATA Version 4.0
- Distributed Interactive Simulation (DIS) IEEE Standard 1278
- High-Level Architecture (HLA) IEEE Standard 1516

# Chapter 2. Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard [OGC 06-121r9] shall apply. In addition, the following terms and definitions apply.

- Broker | Brokering Publisher

  ```
  Intermediary between Subscribers and other Publishers which have been previously
  registered with the broker. The broker is not the original producer of messages,
  but only acts as a message middleman, re-publishing messages received from other
  Publishers and decoupling them from their Subscribers
  ```

- Message

  ```
  A container within which data (such as XML, binary data, or other content) is
  transported. Messages may include additional information beyond data, including
  headers or other information used for routing or security purposes
  ```

- Publish–subscribe pattern

  ```
  A messaging pattern where message publishers categorize and send a message to a
  class of subscribers without knowledge of subscribers.
  Similarly, subscribers express interest in one or more classes of messages and
  receive them without knowledge of their publishers.
  ```

- Publication

  ```
  A uniquely identified aggregation of messages published by a Publisher over time. A
  Publisher may offer any number of publications that Subscribers may subscribe to
  ```

- Publisher

  ```
  An entity that offers publications to Subscribers; supports subscription management
  (subscribe, unsubscribe) and is responsible for filtering and matching messages of
  interest to active subscriptions
  ```

- Receiver

  ```
  An entity that receives messages from Senders; may (but need not) be the original
  Subscriber
  ```

- Reliable Publisher

```
A publisher of messages that offers capabilities to detect and recover from message delivery losses, whether caused by network failures, software failures, hardware failures, or other causes
```

- Sender

```
An entity that sends messages to Receivers; may (but need not) be the initial creator/producer of the data in the message payload
```

- Subscriber

```
An entity that creates a subscription at a Publisher; may (but need not) be the Receiver of delivered messages
```

- Subscription

```
Expression of interest in all or part of a publication offered by a Publisher. When a subscription has been created, the Publisher delivers messages that match the subscription criteria to the Receiver defined in the subscription
```

## 2.1. Abbreviated terms

- ATC Air Traffic Control
- ATM Air Traffic Management
- ADS–B Automatic Dependent Surveillance–Broadcast
- CFP Call for Participation
- CSW Catalogue Service for the Web
- CHI Computer-Human Interface
- DDS Data Distribution Service
- DIS Distributed Interactive Simulation
- DWG Domain Working Group
- ERAM En Route Automation Modernization
- FAA Federal Aviation Administration
- HLA High-Level Architecture
- HMI Human-Machine Interface
- HTTP Hypertext Transfer Protocol
- IEEE Institute of Electrical and Electronics Engineer
- InterCOM Intercommunication

- IoT Internet of Things
- MEP Message Exchange Pattern
- MQTT Message Queuing Telemetry Transport
- OAI-PMH Open Archives Initiative Protocol for Metadata Harvesting
- OASIS Organization for the Advancement of Structured Information Standards
- OData Open Data Protocol
- OGC Open Geospatial Consortium
- OMG Object Management Group
- OpenWIS Open WMO Information system
- OpenAPI Open API Standard
- OWS OGC Web Services
- PSR Primary Surveillance Radar
- PubSub Publish–subscribe
- QoS Quality of Service
- RFQ Request for Quotation
- SOS Sensor Observation Service
- SSE Server-Sent Events
- SSR Secondary Surveillance Radar
- STARS Standard Terminal Automation Replacement System
- SWG Standards Working Group
- TRACON Terminal Radar Approach CONtrol
- UAS Unmanned Aviation System
- URL Uniform Resource Locator
- UML Unified Modelling Language
- VoIP Voice over Internet Protocol
- VRR Voice Recognition/Response
- WIS WMO Information system
- WPS Web Processing Service
- WCS Web Coverage Service
- WFS web Feature Service
- XML eXtensible Markup Language

## 2.2. UML notation

Most diagrams that appear in this document are presented using the UML 2 static structure diagram, as described in Subclause 5.2 of [OGC 06-121r9].

All classes in this document are extensible and may be extended with application- or domain-specific content via Extension blocks.

| | |
|---|---|
| **NOTE** | The UML shown in this document is considered conceptual and abstract, and should not be interpreted as an implementation strategy for bindings that extend and implement a standard. For example, TM_Instant from ISO 19108 may be used to represent time instants for conceptual clarity, but bindings and implementations of this document may realize TM_Instant as a GML TimeInstant, an ISO 8601 date string, or any other representation that is consistent with TM_Instant. |

# Chapter 3. Overview

The Open Geospatial Consortium (OGC) has conducted significant work on event-based models and architectures. The publish-subscribe model results in less network traffic and timely responses to manage event-based model such as urgent, unpredictable pieces of information, hazard warnings and sensor data.

Since 2010, these efforts have been gathered under the scope of the PubSub SWG(Standards Working Group) and resulted in the adoption of the Publish-Subscribe Interface Standard 1.0 (in short, PubSub) in February 2016. The OGC PubSub describes a mechanism to support publish-subscribe requirements across OGC service interfaces, such as Sensor Observation Service (SOS) and Catalogue Service for the Web (CSW), and data types, such as coverages, features, and observations.

OGC PubSub is intended as an overarching model to complement OGC service interfaces with publish-subscribe capabilities, alongside the primarily addressed request/reply model.

OGC Testbed 12 initiative in 2016 addressed the means to incorporate forms of asynchronous service interaction, including publish-subscribe message patterns for Web Processing Service (WPS), Web Coverage Service (WCS), Web Feature Service (WFS) and in application domains such as the Sensor Web and Aviation. [Bigagli2017]

The subtask description in this testbed distinguished among three different approaches to handle asynchronous interaction with OGC Web services:

- WPS facades;
- Specific extensions to each OGC Web Service with asynchronous request/response capabilities;
- OGC PubSub.

MQ Telemetry Transport, known as MQTT (Message Queuing Telemetry Transport), is an OASIS (Organization for the Advancement of Structured Information Standards) standard for the lightweight and resource-constrained publish/subscribe messaging protocol. The main concepts of MQTT are topic, and publish and subscribe functions. When a new message is published to a topic, anyone subscribing to that topic will receive a notification including that message.

For Internet of Things (IoT) applications, OGC SensorThings API is following Open Data Protocol(ODATA) for managing the sensing resources. In addition to supporting Hypertext Transfer Protocol(HTTP), OGC SensorThings API has the extension for supporting MQTT for the creation and real-time retrieval of sensor Observations. The rest of this white paper is organized in the following sections:

Section 4 introduces the asynchronous operations and models in any interaction between two software entities.

Section 5 summarizes the effort of OGC Publish/Subscribe Interface Standard and illustrates its features in a generic OWS extension.

Section 6 describes MQTT adoption for the OGC SensorThings API.

Section 7 examines the use of PubSub services to disseminate Meteorological data.

Section 8 illustrates the use of publish-subscribe model in aviation.

Section 9 discusses AsyncAPI spec as an additional tool for use in developing OGC Web API standards.

Section 10 concludes the future work and actions.

# Chapter 4. Asynchronous Operations

An asynchronous operation is any interaction between two software entities where the concluding action does not immediately follow the initiating action. There are several different models for Asynchronous Operations. This section attempts to describe some of the more common ones. An understanding of the scope of this space will help discuss and compare the alternatives OGC faces in asynchronous services.

This section will look at three aspects of asynchronous operations:

1. The overall pattern of behavior
2. How the two entities rendezvous
3. The criteria for selecting response messages

## 4.1. Behavior Models

### 4.1.1. Request-Response

Request-Response is a synchronous behavior model. A request is issued by one entity and a response is provided in return (Figure 1). Asynchronous behaviors can best be understood in contrast to this model.

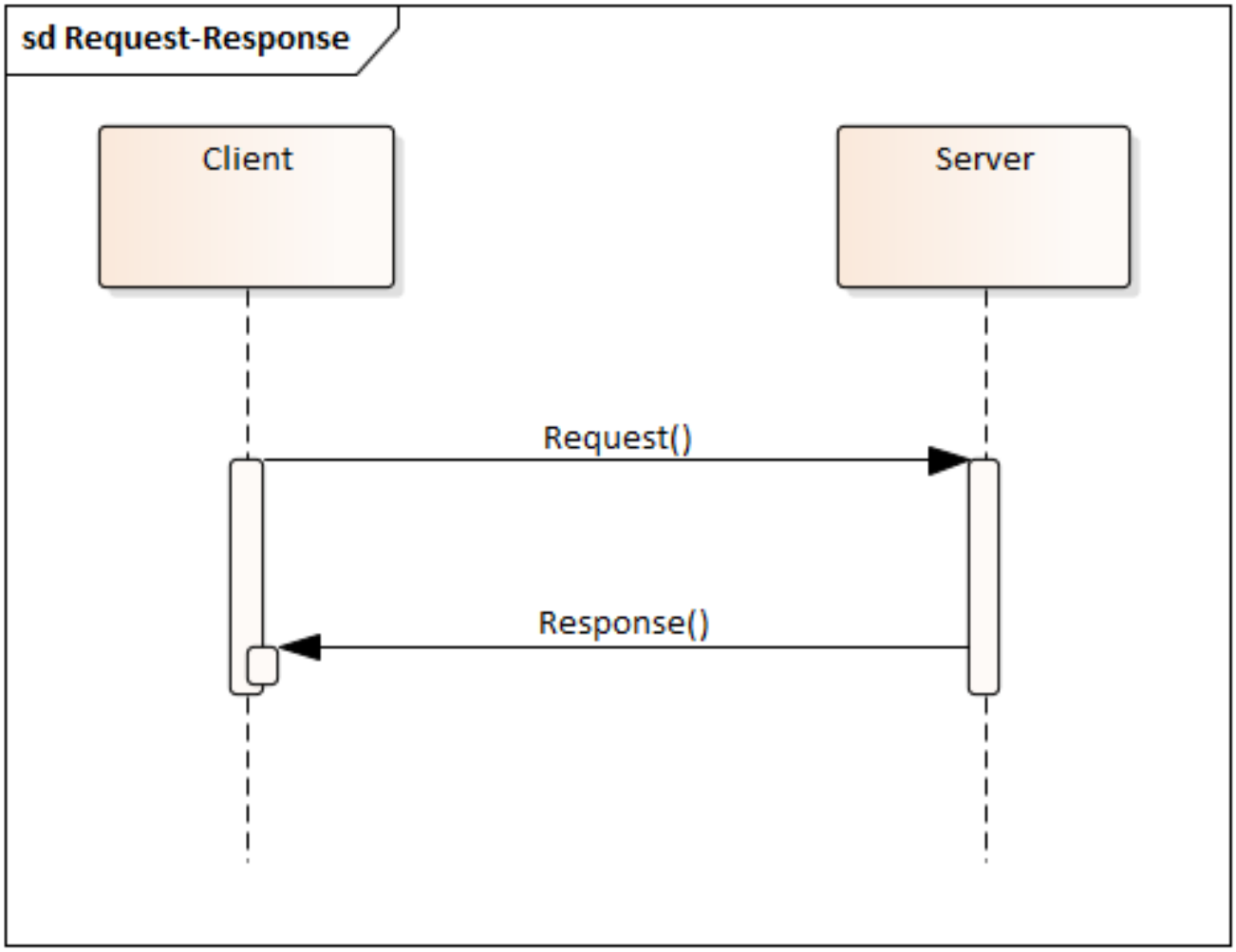



*Figure 1. Request-Response behavior model*

### 4.1.2. Delayed Response

This model differs from Reqest-Response in that the immediate response is not the final response. Rather it acknowedges successful receipt of the request. The final response is to be delivered latter (Figure 2).

A key part of this pattern is the callback. A callback is an http accessible resource which can receive information (responses) from the server for the client.

[Callback] | *images/Callback.png*

*Figure 2. Delayed Response behavior model*

### 4.1.3. Standing Request

A Standing Request is a variation of the Delayed Response pattern. While a Delayed Response performs a single operation, a Standing Request is active until instructed to stop (Figure 3).

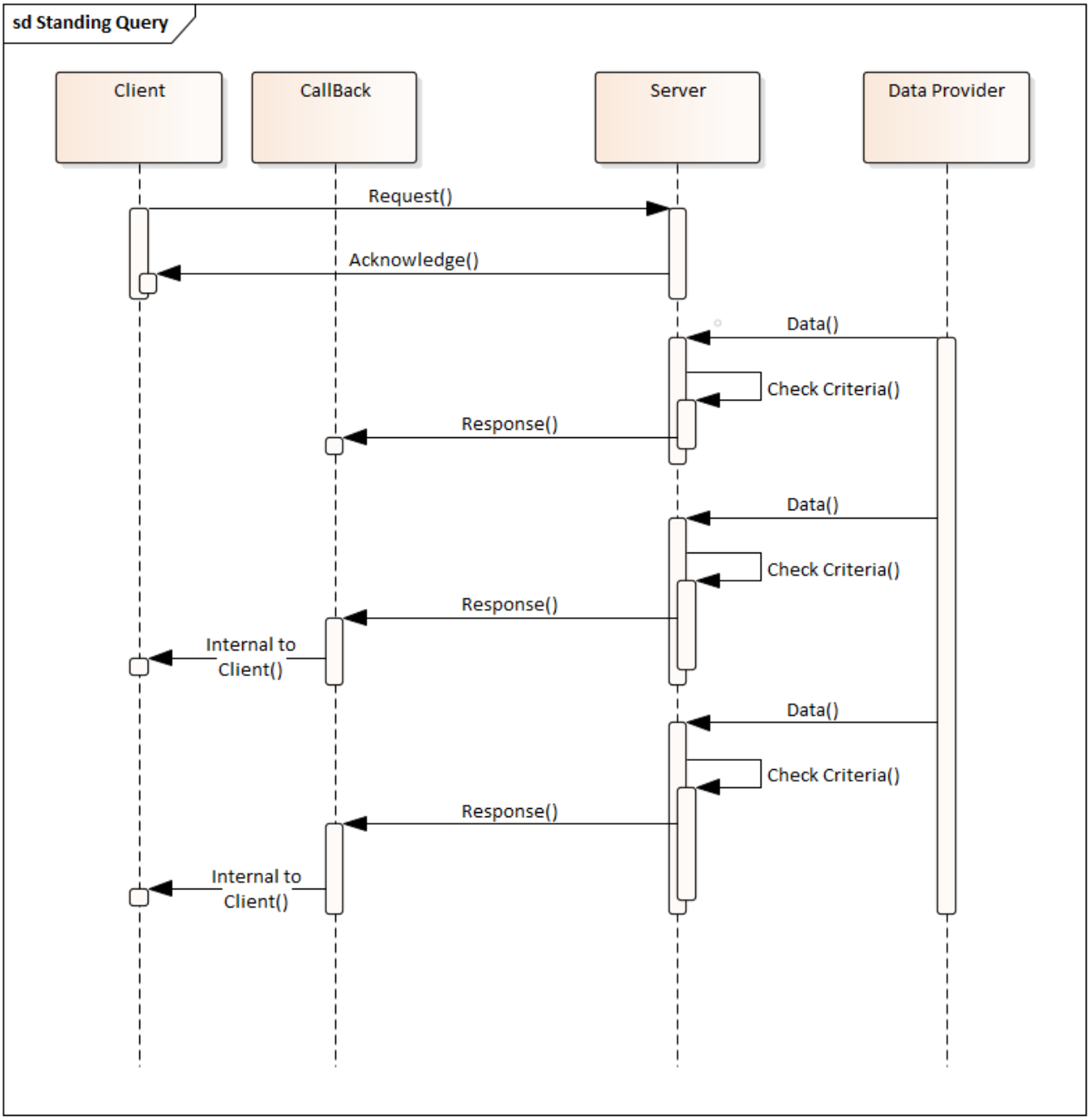



*Figure 3. Standing Request behavior model*

### 4.1.4. Synchronization

The Synchronization pattern supports a scenario where communication between the message producer and consumer is intermittent. When communication is possible, they perform whatever transactions are needed to synchronize their states, then establish a checkpoint for that state. Both parties can then continue to operate independently until the next synchronization opportunity arrives.

### 4.1.5. Publish-Subscribe

A messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead to categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly,

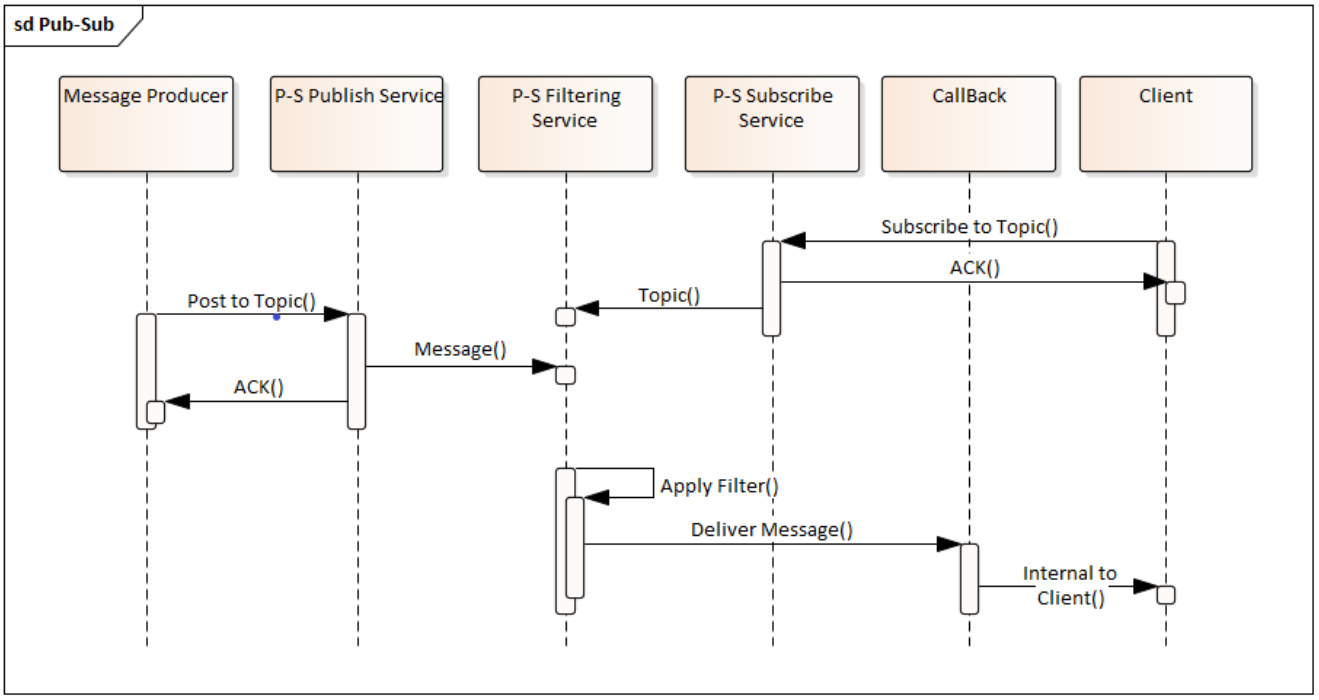
subscribers express interest in one or classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are. (Wikipedia)

The Publish-Subscribe model is distinguished from the request/reply and client/server models by the asynchronous delivery of messages and the ability for a Subscriber to specify an ongoing (persistent) expression of interest.

Alternate Text

The Publish-Subscribe model completely separates message producers and consumers. Potential consumers of messages create filtering criteria which describe the types of messages they wish to receive. They then "subscribe" to a Publish-Subscribe service with this filtering criteria. Producers of messages "publish" those messages to the Publish-Subscribe service along with a set of tags which describe each message. The Publish-Subscribe service evaluates the tags against the filtering criteria of all subscribers. The message is forwarded to all subscribers who's criteria are met.

The "publish" operation follows the Request-Response pattern. The "subscribe" operation follows the Standing Request pattern.




### 4.1.6. Broadcast

Broadcast is the simplest asynchronous pattern. The message producer simply sends the message to everyone. It is left up to the recipients to decide what to do with it.

## 4.2. Notification and Alert

An inherent property of Asynchronus operations is that there is no persistent connection between the message producer and message receiver. Therefore, there must be a way for the message producer to re-establish a connection with the receiver in order to complete the transaction. There are a number of ways this is done.

### 4.2.1. Callback

Callbacks can be viewed as mini-services who's sole purpose is to receive an asychronous response. Information on how to access the callback is provided with the initial request. Message producers (or their agents) use this information deliver responses, typically using the Request-Response pattern.

### 4.2.2. Polling

In polling the requesting entity checks on the status of their request on a recurring basis. Upon completion of the request, the requestor retrieves the result to complete the transaction.

### 4.2.3. Stored response queue

A stored response queue is a service which holds responses to asynchoronous requests. The message producer simply leaves the response in the queue, and it's up to the requestor to retrieve it.

### 4.2.4. Man in the Loop

If all else fails, let the human do it. Many alternatives are available including instand messaging, e-mail, phone calls, even the Postal Service.

## 4.3. Filtering

Filtering allows a message producer to identify the intended recipients of a message.

### 4.3.1. Event (RSS, SNMP)

Event filtering specifies that a notification will be sent if certain conditions occure. For example, if the free space in a mail box drops below 10%.

### 4.3.2. Topic Hierarchy

Publish-Subscribe implementations typically define a set of topics (terms) which can be used to select messages for delivery. In the most basic case a recipient can only subscribe to topics. More capable systems may provide a simple query language to go with the topic vocabulary.

Example: MQTT uses Topic Filters to select messages. A Topic Filter is a path-like hierarchy of concepts. Wildcards are supported to indicate a single path entry or multiple. For example:

1. sport/tennis/player1/score/Wimbledon is a Topic Name
2. sport/+/player1 is a Topic Name with a wild card for only one level
3. sport/tennis/#/ranking is a Topic Name with a wild card for 1 or more levels.

### 4.3.3. Query expression (Standing Query)

Java Messaging Service (JMS) is the foundation for many (most) publish-subscribe services. JMS

supports messaging selection through a query string. The query language is a subset of the SQL92

More capable systems support a full query language for filtering messages. For example, an asynchronous WFS would accept asynchronous requests using the same Filter Encoding language as any other WFS. But the results would be returned asynchronously.

### 4.3.4. Check Point

A check point is a store snapshot of the state of the system as a specific date and time. All changes made after a check point are can

# Chapter 5. OGC's PubSub Implementation Standard

## 5.1. Overview

The OGC has conducted significant work on event-based models and architectures. Figure 4 provides an overview and a timeline of relevant OGC work in the last decade.

The rows show the roadmap of various documents, including official OGC standards (highlighted in bold fonts) and milestones, indicated with their version and OGC reference number and type (Best Practice, Discussion Paper, Implementation Standard, and Engineering Report).

See the abbreviations section for a complete list of abbreviations.

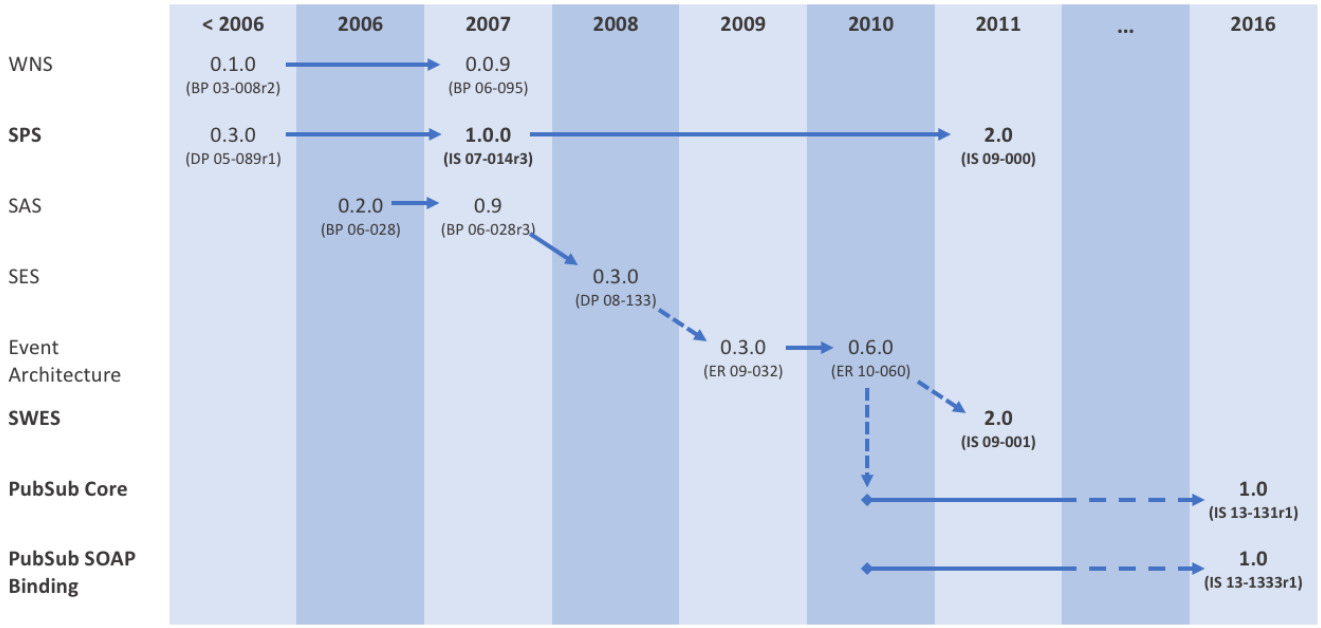



*Figure 4. OGC Publish/Subscribe related work (source: [Rieke2018])*

From 2010 onward, these efforts have been subsumed under the scope of the PubSub SWG and resulted in the adoption of the Publish/Subscribe Interface Standard 1.0 in February 2016.

## 5.2. OGC's PubSub Features

The OGC Publish/Subscribe Interface Standard (in short, PubSub) describes a mechanism to support publish/subscribe requirements across OGC service interfaces, such as Sensor Observation Service (SOS) and Catalogue Service for the Web (CSW), and data types, such as coverages, features, and observations.

PubSub is intended as an overarching model to complement OGC service interfaces with publish/subscribe capabilities, alongside the primarily addressed request/reply model. In fact, the latter is sufficient to meet many use cases, when clients request the data of interest upon need, and may re-issue requests periodically for updates. However, the publish/subscribe model can be better suited to manage urgent, unpredictable pieces of information, such as sporadic events, hazard

warnings and sensor data, resulting in less network traffic and more timely responses.

In the assumption that clients are interested in the information already provided by OGC services, the rationale for PubSub is to avoid impact on the existing service semantics, when possible reusing the specialized mechanisms for accessing and filtering the information provided (*e.g.*, the Filter Encoding Specification [1] for general-purpose filtering). Hence, PubSub provides notification and messaging capabilities as a module to existing services, as well as a standalone, independent OWS capability. Besides, it supports simple notifications (*e.g.* references to data from OWS services) or actual data push.

The basic OGC PubSub workflow, depicted in Figure 5, involves two primary actors: the Publisher of information (XML, binary data, or other content); and the Subscriber, which subscribes to the Publisher to obtain the published information, possibly filtered according to criteria defined upon subscription. Information is contained and transported in messages, which may include additional data, such as for routing or security purposes.

PubSub distinguishes two additional roles, the Sender and the Receiver, respectively the entity that delivers messages and the entity to which messages are delivered, allowing versatile use cases (e.g. a Subscriber entity may subscribe to a Publisher a number of third parties acting as Receivers; or a Publisher may be segregated from the ultimate recipients of its messages by a separate Sender).

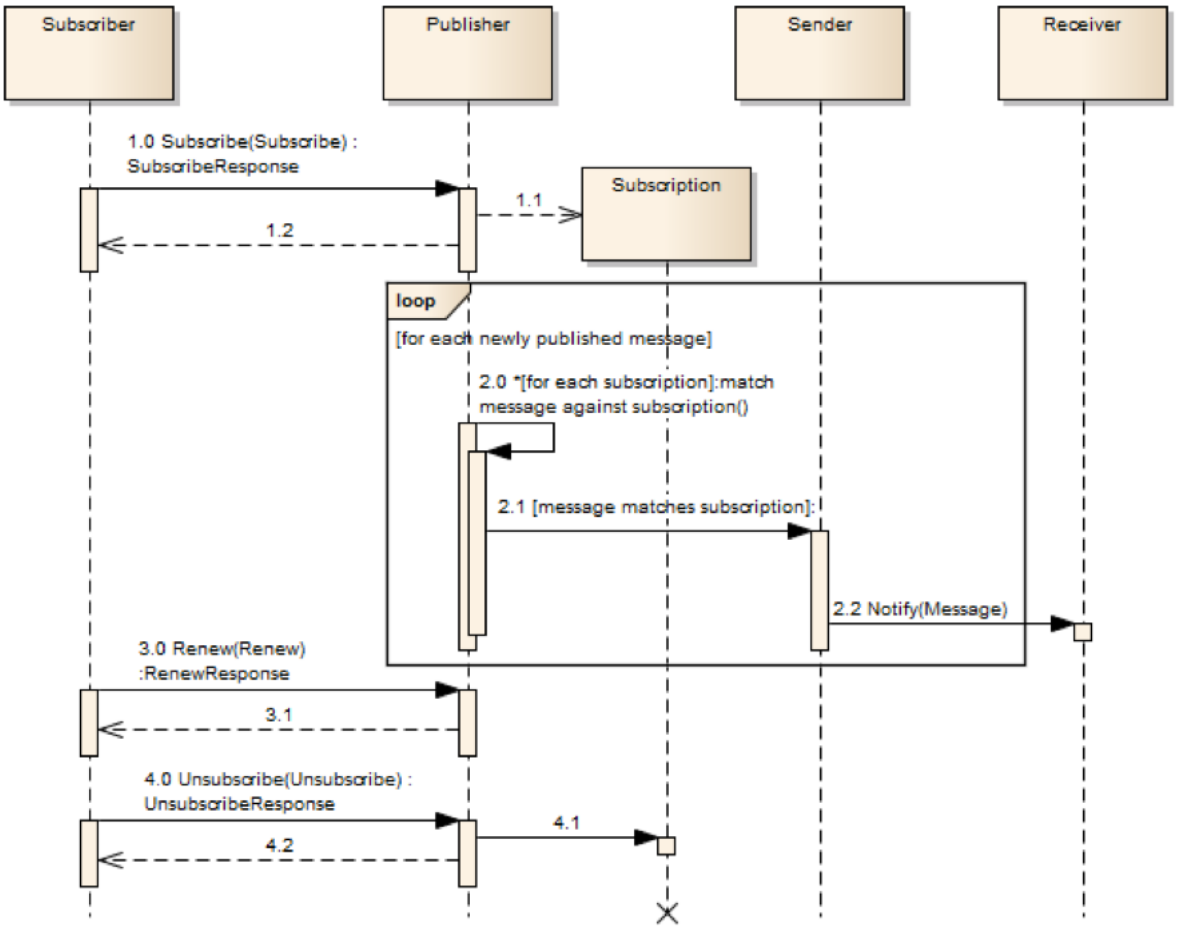



*Figure 5. OGC PubSub basic workflow (source: [PubSubCore2016]).*

In the above figure:

- The Subscriber creates a subscription (with possible filter criteria) on behalf of a Receiver using

the Subscribe operation on a Publisher (1.0).

- The Publisher creates a Subscription (1.1) and returns a SubscribeResponse (1.2).
- The Publisher attempts to match each new message against the filter of each Subscription (2.0).
- If a message matches, the Sender delivers it to the appropriate Receiver via the Notify operation (2.1).
- The Subscriber may utilize the Renew operation (3.0) to extend the lifetime of a Subscription.
- If the Publisher accepts the request, it sets the new termination time on the Subscription and returns a RenewResponse (3.1).
- The Subscriber may at any time request the termination of a Subscription via the Unsubscribe operation (4.0).
- If the Publisher accepts the request, it terminates the subscription (4.1) and returns an UnsubscribeResponse (4.2).

To qualify as a Publisher, an OWS must at least conform to the Basic Publisher conformance class, that is satisfy the requirements for the above basic functionalities. Besides, it must return in its *GetCapabilities* response the three additional Capabilities components represented in Figure 6.

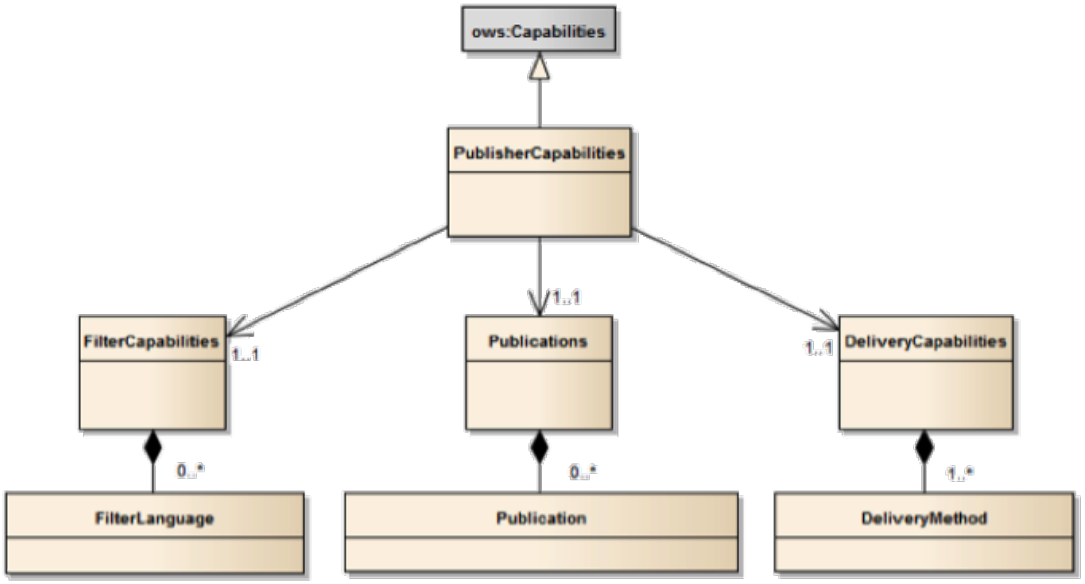



*Figure 6. OGC PubSub additional Capabilities components (source: [PubSubCore2016]).*

The Publications component describes the contents offered by the Publisher, *i.e.*, the sets of messages that Subscribers can subscribe to. The specification is agnostic as to what constitutes a publication, *i.e.*, what events should cause notifications by a Publisher (*i.e.*, its event model). Likewise, PubSub is agnostic as for the encoding of messages.

The *FilterCapabilities* component describes the filtering-related capabilities of a Publisher, *i.e.*, the filter languages it supports for matching messages against subscriptions (*e.g.*, OGC Filter Encoding, XQuery). PubSub is agnostic as for the language to filter messages in subscriptions. A Publisher may support multiple filter languages, to support different Subscribers.

The *DeliveryCapabilities* component describes the methods supported by the PubSub-enabled OWS for delivering messages. The publish/subscribe MEP typically implies push-style message delivery,

however some delivery methods may actually be underpinned by pull-based mechanisms (e.g. polling). Examples of delivery methods include: SOAP and related technologies, such as WS-N (used by the PubSub SOAP Binding), ATOM, PubSubHubbub, OAI-PMH, e-mail, Short Message Service, WebSockets and SSE. The PubSub standard is agnostic as regards delivery methods. A Publisher may offer more than one method of delivery for each Publication, to be chosen by Subscribers.

In addition to the mandatory Basic Publisher conformance class, PubSub defines several other optional conformance classes, introducing additional functionalities, e.g. to pause a Subscription (Pausable Publisher), derive additional publications (Publication Manager), group messages in batches (Message Batching Publisher). Moreover, it defines conformance classes to bind such functionalities to actual technologies. At present, the PubSub specification consists of two parts:

1. a Core document [PubSubCore2016] that abstractly describes the basic mandatory functionalities and several optional extensions, independently of the underlying binding technology;
2. a SOAP binding document [PubSubSOAP2016] that defines the implementation of PubSub in SOAP services, based on the OASIS Web Services Notification (WS-N) set of standards. [WSBN2016]

The scope of the OGC PubSub Standard Working Group also includes a RESTful binding, to realize the PubSub functionality in REST/JSON services. Several communities are proposing additional extensions (e.g. bindings to JMS and MQTT), leveraging on the modular structure of the OGC specifications.

PubSub Core requires that a PubSub-enabled OWS advertise the implemented Conformance Classes in its Capabilities document. Figure 7 shows all the conformance classes currently defined by the PubSub specification.

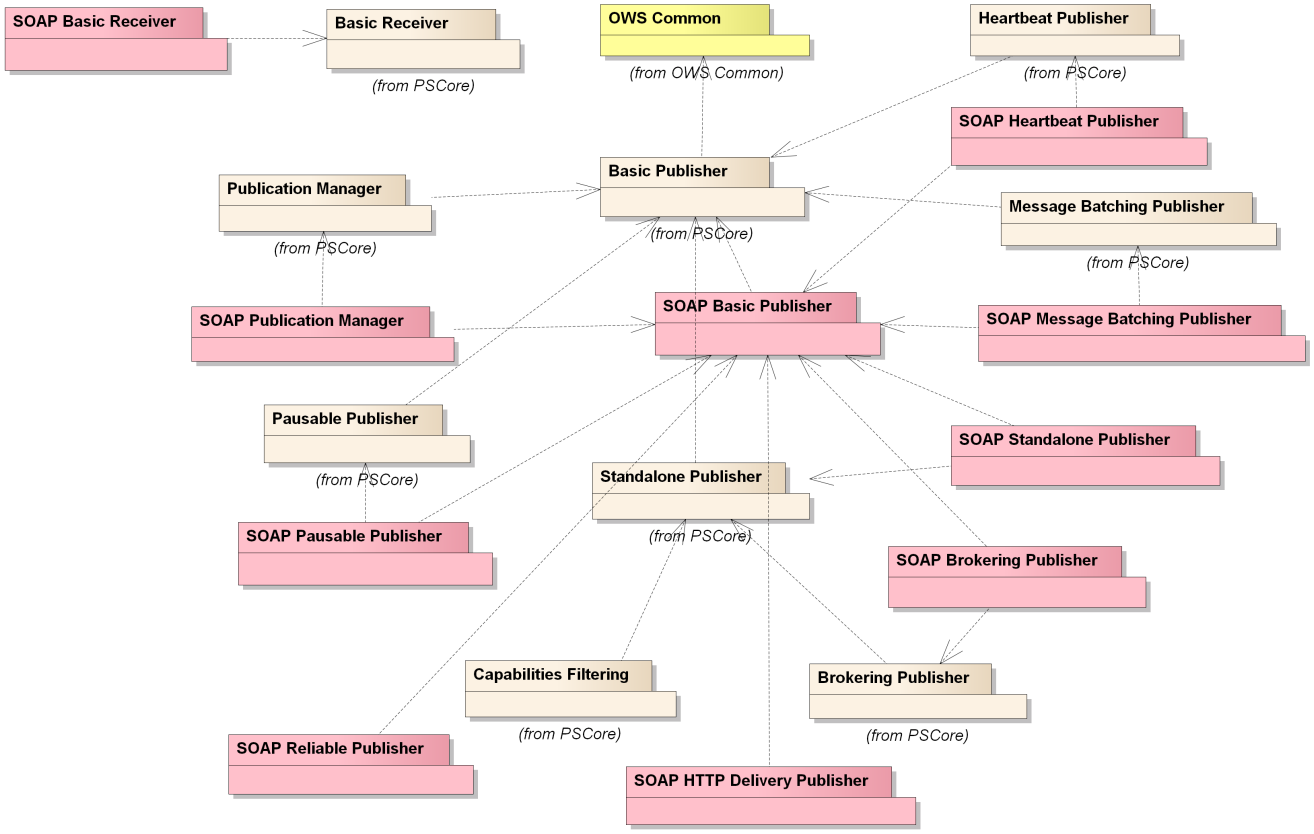



*Figure 7. OGC PubSub Conformance classes (source: [PubSubCore2016]).*

## 5.3. Related Work

Recognizing that the OGC baseline mainly supported synchronous web service request-response capabilities, the 2016 OGC Testbed 12 initiative addressed the means to incorporate forms of asynchronous service interaction, including Publish/Subscribe message patterns, for example in WPS, WCS, WFS, or in application domains such as the Sensor Web and Aviation. [Bigagli2017]

In particular, the RFQ/CFP [2] included a specific Asynchronous Service Interaction subtask, part of a set of subtasks that aimed at enhancing the OGC Baseline, by extending OGC architectural designs through efforts that cross over several individual standards and services and are applied in a much wider scope.

The subtask description in the RFQ/CFP distinguished among three different approaches to handle asynchronous interaction with OGC Web services:

1. WPS façades;
2. Specific extensions to each OGC Web Service with asynchronous request/response capabilities;
3. OGC PubSub.

The document deliverable "A067 Implementing Asynchronous Service Response Engineering Report" (OGC 16-023) elaborates on the above approaches in situations where big chunks of data require asynchronous delivery. The ER focuses on the first and the second approach, with the goal to summarize and compare the results from using a WPS facade and an extension for WFS for asynchronous service responses, as well as to provide recommendations for future activities.

The document deliverable "A074 PubSub/Catalog Engineering Report" (OGC 16-137) [OGC19-137r2] focused on the third approach, OGC PubSub, and exemplified the use of the standard, particularly in conjunction with the Catalog Service interface, investigating the functional requirements of an interoperable, push-based data discovery solution. As underlined in the RFQ/CFP, it is important to provide methods that support notification (push) of new data as opposed to search (pull), given the volume of data typically available in catalogs.

Besides, it introduced a general, basic mechanism for enabling PubSub for the generic OGC Web Service over the existing request/reply OWS's, i.e. usual requests as filters, usual responses as appropriate updates/data pushes, existing semantics and syntax expressiveness. The following chapters summarize such mechanism.

## 5.4. Basic PubSub 1.0 extension for the generic OWS

The PubSub extension for the generic OWS introduced by OGC 16-137 is conceived as a simple way to enable the existing request/reply OWS specifications to Publish/Subscribe, by implementing the OGC Publish/Subscribe Interface Standard 1.0.

An OWS implementing this extension is capable of accepting its usual requests as filters, and of sending notifications about data/metadata updates, based on its existing semantics and syntax expressiveness.

### 5.4.1. Conceptual model

This chapter describes how PubSub 1.0 Core operations, encodings and messages are modeled in terms of the functionalities of the generic OWS. No assumption is made on the capabilities of the target OWS, other than those defined by the OGC Web Services Common Standard. Hence this extension may apply, for example, to WFS, WCS, and other OWS interfaces.

The PubSub specification is agnostic as to what constitutes a change, i.e. an event that should cause a notification by a Publisher (aka its event model). It is only required that a Publisher instance communicate what notifications it will emit by advertising them in the Publication section of its Capabilities document (see below).

In general, a PubSub-OWS may be able to notify about changes to any component of its information set. For example, it may notify about changes to its Capabilities document. The extension introduced in this chapter addresses the most general case, at the expenses of efficiency and semantic accuracy. The precise definition of an event model for the various OWS's is left to the relevant OGC Working Groups.

The basic PubSub-OWS MEP can be generalized as follows (see Figure 8):

1. The OWS client subscribes specifying a request to be used as filter for the notifications;
2. The OWS client obtains the Time-0 response via a standard Request/Reply, with the same request as above;
3. The OWS notifies the client of subsequent updates to the response, according to its existing semantics and syntax.

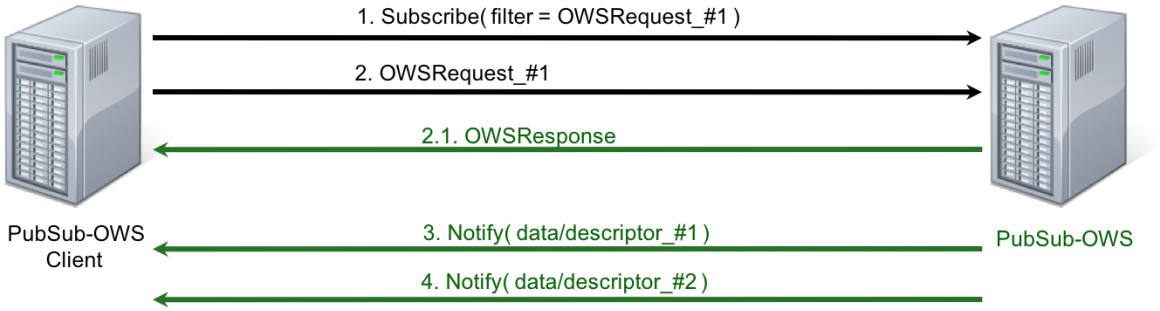



*Figure 8. OWS Publish/Subscribe MEP*

This may be formalized in an "OWS Request/Reply Publisher" Conformance Class that:

- Accepts OWS requests as subscription filters
  - The Publisher may constraint the filter expressions allowed in Subscriptions (e.g. by imposing OpenSearch templates)
- Sends corresponding OWS responses to notify about data/metadata updates

This MEP is a simple way to enable existing OWSs to PubSub, allowing to bind the PubSub 1.0 Core operations, encodings and messages to the standard OWS functionalities, data models, and semantics.

### 5.4.2. Required Capabilities components

PubSub Core requires that the OWS advertise the implemented Conformance Classes in its Capabilities document, namely in the Profile property of the ServiceIdentification section (as of OWS Common 1.1). Besides, it requires that the additional Capabilities components represented in Figure 9 are returned in the GetCapabilities response, but does not specify the specific mechanism for incorporating these additional Capabilities components into the OWS Capabilities document. These extension proposes to include these additional Capabilities components in the ExtendedCapabilities of the OWS, as detailed in the following chapters.

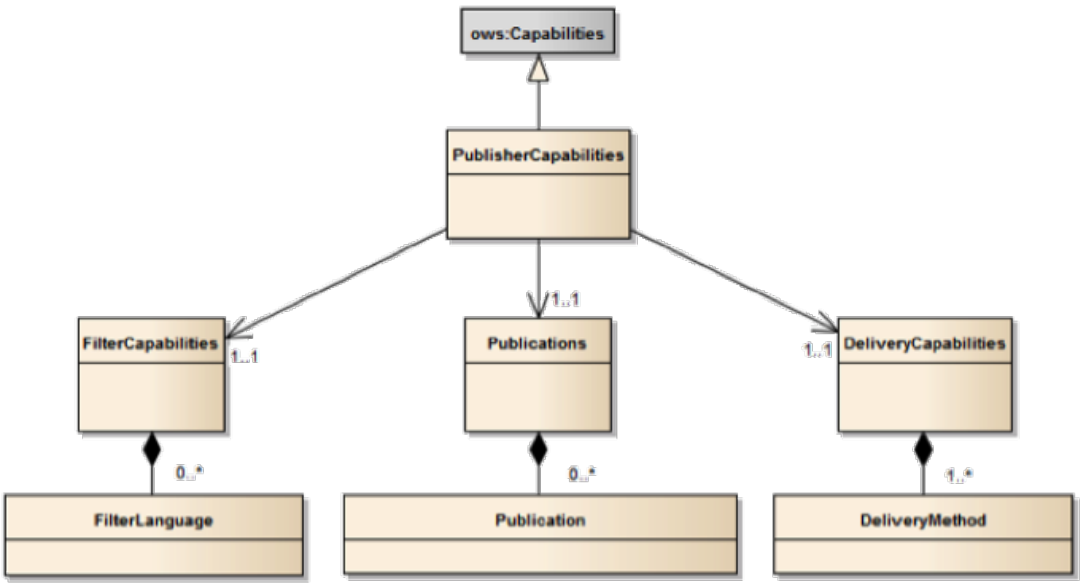



*Figure 9. PubSub Capabilities components*

### FilterCapabilities

The FilterCapabilities section describes the filtering-related capabilities of a PubSub-OWS, i.e. the filter languages it supports for matching events against subscriptions (e.g., OGC Filter Encoding). This allows the pluggability of filter languages.

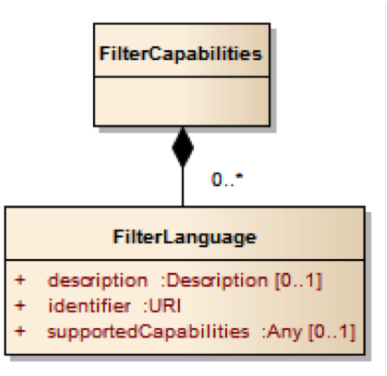



*Figure 10. Filter Capabilities*

The SupportedCapabilities elements allows restricting the acceptable requests, possibly providing templates. The following Capabilities snippet declares that this PubSub-OWS instance (namely, a CSW) accepts as subscription filters GetRecords requests conforming to the specified OpenSearch template. Multiple templates may be introduced, specifying multiple FilterLanguages.

*FilterCapabilities*

```
<FilterCapabilities>
   <FilterLanguage>
      <Abstract>This PubSub-OWS accepts requests as subscription filters, according to
the OpenSearch template specified in SupportedCapabilities.
      </Abstract>
      <Identifier>http://www.opengis.net/spec/pubsub/1.0/conf/ows/request-reply-
publisher</Identifier>
      <SupportedCapabilities>http://tb12.essi-lab.eu/pubsub-
csw/services/opensearch?ct={count?}&amp;st={searchTerms?}&amp;bbox={geo:box?}&amp;ts={
time:start?}&amp;te={time:end?}
      </SupportedCapabilities>
   </FilterLanguage>
</FilterCapabilities>
```

### DeliveryCapabilities

The DeliveryCapabilities section describes the delivery methods supported by the PubSub-OWS, e.g. SOAP, WS-Notification, ATOM, SSE, WebSockets, OAI-PMH. This allows the pluggability of delivery methods.

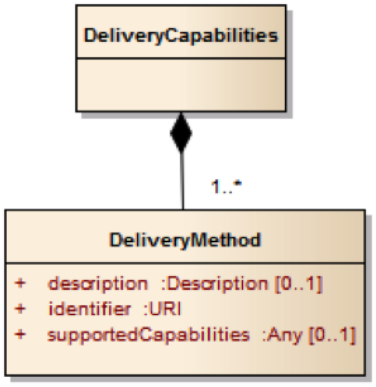



*Figure 11. Delivery Capabilities*

The following Capabilities snippet declares that this PubSub-OWS instance delivers notifications via SSE (see chapter Delivery methods, below).

*DeliveryCapabilities*

```
<DeliveryCapabilities>
   <DeliveryMethod>
      <Abstract>This PubSub-OWS supports notification delivery via SSE.
      </Abstract>
      <Identifier>http://www.w3.org/TR/eventsource/
      </Identifier>
   </DeliveryMethod>
</DeliveryCapabilities>
```

#### Delivery methods

The DeliveryCapabilities section describes the methods supported by the PubSub-OWS for delivering notifications. Publishers may offer more than one method of delivery for each Publication, to be chosen by Subscribers. Publish/Subscribe would imply push-style message delivery, however some methods may actually be pull-based (e.g. polling), under the hood.

Examples include: SOAP and related technologies, such as WS-Notification (used by PSSB), ATOM (polling using the “If-Modified-Since” and “start-index” parameters), PubSubHubbub, OAI-PMH (polling using the “from” parameter), e-mail, SMS, WebSockets, SSE.

Server-Sent Events (SSE) is a pure push-style communication technology based on HTTP and the SSE EventSource API standardized as part of HTML5 by the W3C. A SSE client (e.g. all modern HTML 5.0 browsers) receives automatic updates from a server via HTTP connection, simply setting the following parameters:

- ContentType: "text/event-stream;charset=UTF-8”
- Cache-Control: "no-cache”
- Connection: "keep-alive”

### Publications

The Publications section describes the contents offered by the PubSub-OWS, i.e. the sequences of

notifications that Subscribers can subscribe to.

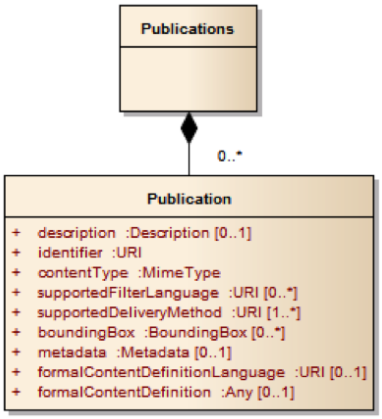



*Figure 12. Publications*

The following Capabilities snippet declares a publication that notifies on all the relevant events for this PubSub-OWS. Notifications can be filtered with the semantics of the requests of this OWS and are delivered via SSE, encoded in JSON (see chapter Notification encoding, below).

*Publications*

```
<Publications>
   <Publication>
      <Abstract>>This publication notifies on all the relevant events for this PubSub-
OWS.
      </Abstract>
      <Identifier>ALL</Identifier>
      <ContentType>application/json</ContentType>

<SupportedFilterLanguage>http://www.opengis.net/spec/pubsub/1.0/conf/ows/request-
reply-publisher</SupportedFilterLanguage>

<SupportedDeliveryMethod>http://www.w3.org/TR/eventsource/</SupportedDeliveryMethod>
   </Publication>
</Publications>
```

#### Notification encoding

For the generic OWS instance, no operation is defined that provides the basic semantics of “insert”, “update”, and “delete” actions on the content managed by the instance.

The most generic mechanism to notify about updates is that the Publisher re-send the whole response element corresponding to the request used as filter in the Subscription. For example, in the case of WFS, if the client subscribes with a wfs:GetFeature request as a filter, the PubSub-WFS should notify about any changes by delivering a standard wfs:FeatureCollection, in response to that request.

By receiving the new response and comparing it with the previous one, a Subscriber can figure out the changes. Future evolutions of this extension may evaluate more efficient and semantically accurate encoding of notifications. A possible option for XML-based content types is XMLdiff (e.g.

XML Patch, RFC 5261), or annotations (XML attributes) to add simple CRUD semantics on top of the existing XSDs.

## 5.5. Support to legacy components

The integration of legacy components in an eventing architecture is desirable in a number of scenarios. However, legacy components may not be instrumented to monitor their state for the purpose of notification, nor to react upon notifications from other components (or they may, but by legacy, non-standard mechanisms).

Implementing the PubSub 1.0 Standard in a legacy component may not be feasible or practical. In some cases, the legacy component can be adapted to the Publish/Subscribe MEP by an additional functional entity that realize the Publish/Subscribe functionalities. Such mediating entity acts as a proxy/adapter, i.e. a middleman between the source and the target of the message exchange, implementing the behavior and/or the interfaces required by the PubSub specification.

This use case has been considered in the phase of requirement analysis for the PubSub 1.0 standard [3] and is supported by the Brokering Publisher Conformance Class of the PubSub 1.0 Standard.

Depending on the intended role of the legacy component, the use case is twofold:

- Proxied Subscribe – a proxy/adapter component subscribes to a Publisher on behalf of the legacy system and acts appropriately upon receiving notifications of interest.

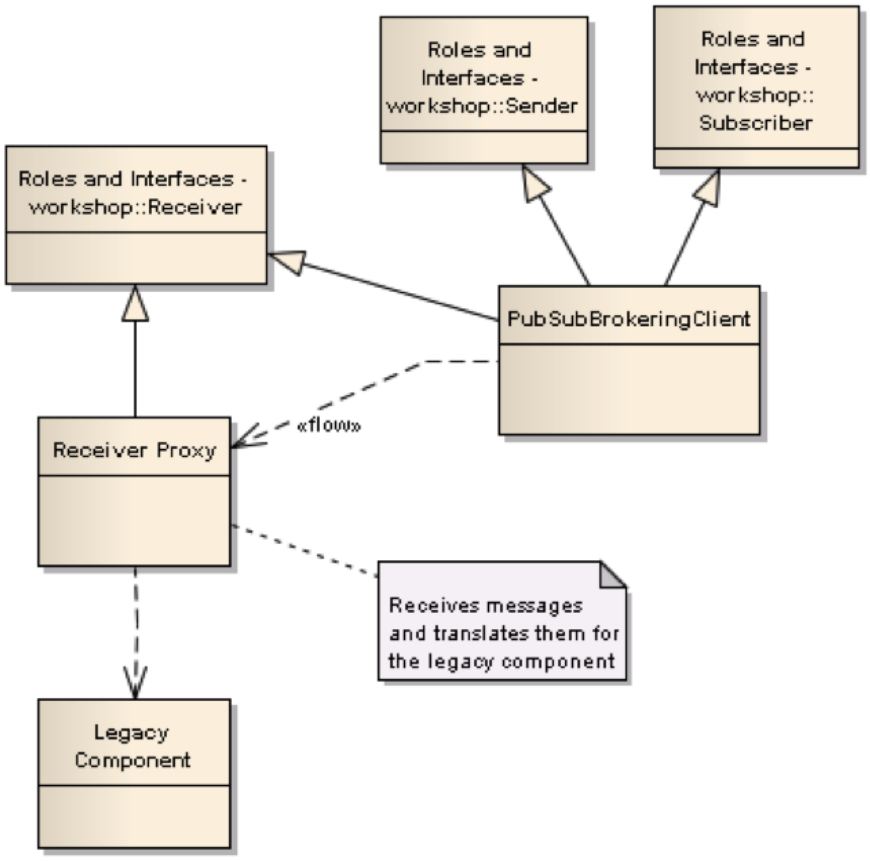



*Figure 13. Proxied subscribe*

- Proxied Publish – a proxy/adapter component monitors the legacy system and generates appropriate notifications upon relevant events (according to a given event model). The

proxy/adapter may act as a full-fledged Publisher, accepting Subscriptions against the sequence of notifications, or just act as a pure Sender, relaying each notification to another Publisher entity (see Figure 14).

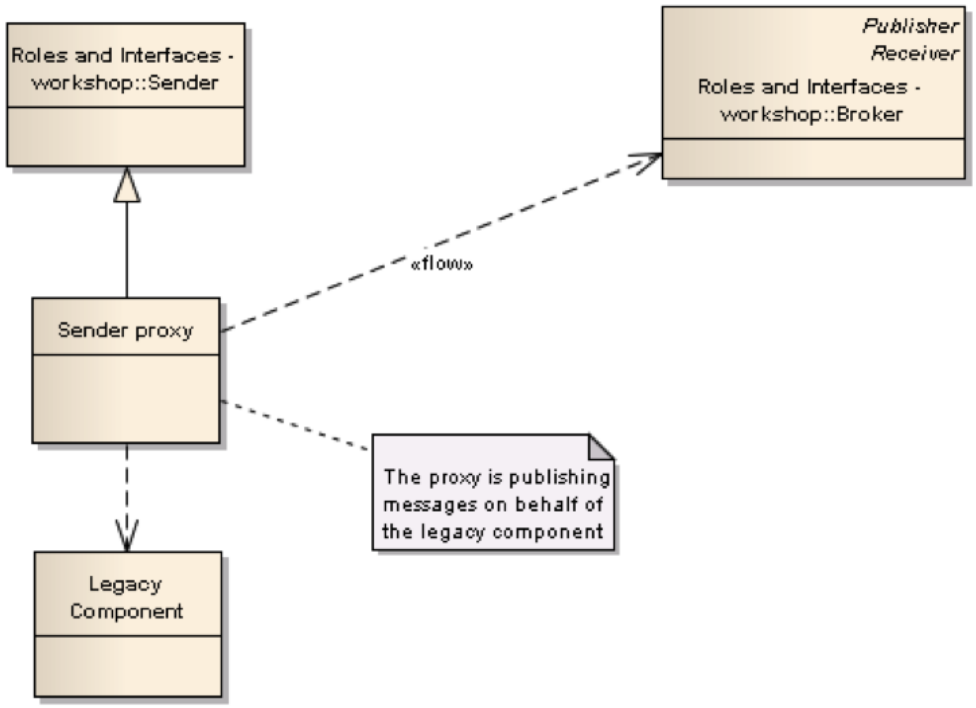



*Figure 14. Proxied publish*

The Brokering Publisher Conformance Class of the PubSub 1.0 Standard supports this use case. In fact, a Brokering Publisher (or, more simply, a broker), is an intermediary between Subscribers and other Publishers which have been previously registered with the broker. The broker is not the original producer of messages, but only acts as a message middleman, re-publishing messages received from other Publishers and decoupling them from their Subscribers. A broker may shuffle or aggregate messages into different publications, may offer publications with different delivery methods than the original ones, or otherwise process the messages (e.g. converting their format). A broker may also provide advanced messaging features, such as load balancing.

In general, a broker is a distinct third party that acts as a communication intermediary between the source and the target of a communication, mediating their interfaces and in some cases adding new behavior. Hence, a broker may conveniently act as a proxy/adapter for one or more legacy components, flexibly implementing any combination of the above twofold use case.

The Brokering Publisher Conformance Class does not mandate any specific behavior to be implemented, in particular as regards the support to Delivery Capabilities, Filtering Capabilities, and Publications of the brokered Publishers. Brokers are free to interact with the brokered Publishers as appropriate for their specific application. Interactions may include subscribing to the offered publications, harvesting the data, decorating the capabilities, or other behavior (future extensions of the Conformance Class may standardize the behavior of Brokering Publishers in specific application scenarios).

Examples of Brokering Publisher applications include the following:

- Publisher Aggregation – a broker subscribes to several Publishers and relays their publications (without modifications) to interested Subscribers, acting like a Proxy to multiple Publishers.

Optionally, the broker may adapt the service interface (binding) of the aggregated Publishers.

- Publication Aggregation – a broker receives messages generated by several Publishers (e.g. dumb sensors) and publishes them to the interested Subscribers as a single publication at a single endpoint, for the sake of simpler connectivity, or improved accountability, or easier management of subscriptions, etc.
- GeoSynchronization (GSS) - GSS is a mediation service that controls transactional access to one or more WFS's (e.g. to moderate updates in crowdsourcing scenarios). A GSS maintains several event channels, including one for changes applied to the WFS content. Clients can subscribe to the channels (possibly specifying a filter) and be notified by the GSS whenever new entries appear. A GSS may be used to monitor insert/update/delete operations performed on one or more WFS's and send appropriate notifications, implementing the PubSub 1.0 Brokering Publisher Conformance Class. Whenever an event (i.e. a Transaction) occurs on a WFS, the GSS will notify Subscribers of that event. In this way WFS's that do not implement the PubSub 1.0 Standard can participate in an eventing architecture. There are plans to extend GSS to other OGC access services, such as WCS.

[1] http://www.opengeospatial.org/standards/filter

[2] http://www.opengeospatial.org/standards/requests/139

[3] See also the Proxied Publish/Subscribe use case (access restricted to OGC Members): https://portal.opengeospatial.org/wiki/PUBSUBswg/PubSubSwgUseCaseBrokeredPubSub

# Chapter 6. PubSub in OGC SensorThings

OGC SensorThings API is a standard for interconnecting the Internet of Things (IoT) devices, data, and applications over the Web. OGC SensorThings API has two main parts: the sensing part that focuses on heterogeneous IoT sensor systems, and the Tasking part focused on IoT actuators and tasking devices.

OGC SensorThings API is following ODATA (Open Data Protocol) for managing the sensing resources. Thus, it has a REST-like API and supports HTTP CRUD operations (GET, POST, PATCH, DELETE) and ODATA query options (select, expand, filter, orderby, top, skip) for data retrieval.

In addition to supporting HTTP, OGC SensorThings API has the extension for supporting MQTT to create and real-time retrieval of sensor Observations.

MQ Telemetry Transport, known as MQTT, is an OASIS standard. MQTT is an extremely lightweight publish-subscribe messaging protocol designed explicitly for resource-constrained IoT devices. The main concepts are topic and publish and subscribe functions. When a new message is published to a topic, anyone subscribing to that topic will receive a notification including that message.

OGC SensorThings API is adopting MQTT protocol and defines a topic structure to create and receive notifications from sensor Observations. The topic structure for OGC SensorThings API MQTT extension is following its HTTP resource path. It means that the topic used for creating Observation through MQTT is identical to the HTTP URL that is used for the POST and creating Observations. Examples of these resource paths are ***v1.0/Observations*** and ***v1.0/Datastreams(id)/Observations***. Similarly, accessing those URLs using HTTP GET would result in retrieving sensor Observations, while subscribing to those as topics means receiving notifications for those Observations in real-time.

Here is a summary of lessons learnt from MQTT adoption is OGC SensorThings API that might be applicable to any RESTful API:

- Each RESTful API has a potential for MQTT binding to receive updates for a resource collection
  - The topic would be the resource GET URL
  - The payload will be the same as the content of HTTP GET
  - Whenever there is a new resource, it will be published to the resource GET URL topic
- For any RESTful API, to create a resource, MQTT can be an option just like HTTP POST
  - The topic will be the same as the POST topic
  - The payload will be the same as the POST payload
  - The service would subscribe to the topics
  - Once it receives the payload, it uses the same process as POST to create the resource

## 6.1. OGC IMIST IOT Pilot

OGC Incident Management Information Sharing Internet of Things Pilot Project (IMIS IoT Pilot) depicts an incident fire, law enforcement, and emergency medical units must deploy, discover and

integrate various sensors and platforms to gain situational awareness. In that pilot, MQTT topics were defined to contain the resource path together with $filter query options. Then the Observation that is published to that topic was filtered by the criteria defined in the topic. In this way, rules can be defined to notify subscribers about specific conditions that happen, such as when the carbon dioxide reading for first responders is higher than a threshold and considered dangerous. Example of the topic would be ***v1.0/Datastreams(id)/Observations?$filter=result gt 100***. IMIS is a pilot implementation and could be a starting point for OGC SensorThings API potential part 3, rules engine. This was a pilot implementation and could be a starting point for OGC SensorThings API potential part 3, rules engine.

# Chapter 7. WMO OpenWIS use of AMQP

The OpenWIS Association, lead by Meteo France, is examining the use of PubSub services to disseminate Meteorological data as part of the World Meteorological Organization (WMO) Information System (WIS) 2 draft technical specification. The OpenWIS Association conducted an initial WIS 2 study on message protocols and found that several use cases were well positioned to use PubSub. This included but was not limted to the current use of WAN for WMO GTS data dissemination, and use of the WAN for NHMs specific users (aeronautical, B2B and general public).

The WIS 2 study also examined a number of message queue protocols the OpenWIS Program could use including: AMQP, MQTT, STOMP, JMS, Kafka, Redis. However, only 2 were found to use ISO standards (AMQP and MQTT). For a long time, the FAA SWIM aprogram has been very successful in standardizing on the mature JMS API for publish-subscribe model. However, an interoperability limitation in the wire-level protocol for JMS being vendor dependent forced the SWIM program to seek other open source alternatives. SESAR/Eurocontrol is now very much interested in AMQP v1.0 publish/subscribe message protocol as a promising new ISO, IEC, and OASIS international standard, that offers SWIM an open publish-subscribe messaging protocol. Given that, the WIS 2 study focused on specific queues that support AMQP.

The first MQ to be analyzed was RabbitMQ. RabbitMQ supports several message queue protocols including: AMQP 0-9-1, AMQP 1.0 (via a plugin), STOMP, MQTT, HTTP (API). In addition, the study found that RabbitMQ has implemented extensions to AMQP.

In support of our WIS 2 study, the following figure (see figure Figure 15) shows the proposed AMQP Architecture

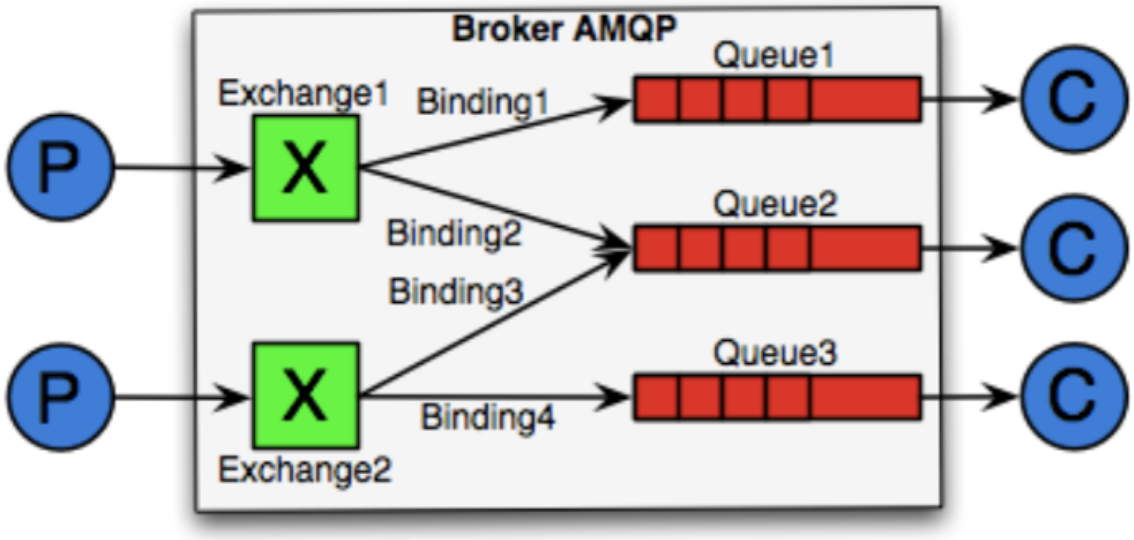



*Figure 15. OWS Publish/Subscribe FIG1*

In our proposed AMQP architecture, all connections are TCP based. There is no UDP. Publish is performed to an exchange, not to a queue. There can be N exchanges. Subscriptions lead to the creation of: 1) a queue (if needed); 2) bindings from exchange to queue. Two basic principles will apply: If there is no binding, there is no routing, and in turn the message is discarded. Also, once messages are consumed, they disappear.

Figure Figure 16 describes one of the proposed AMQP message structure for OpenWIS WIS2

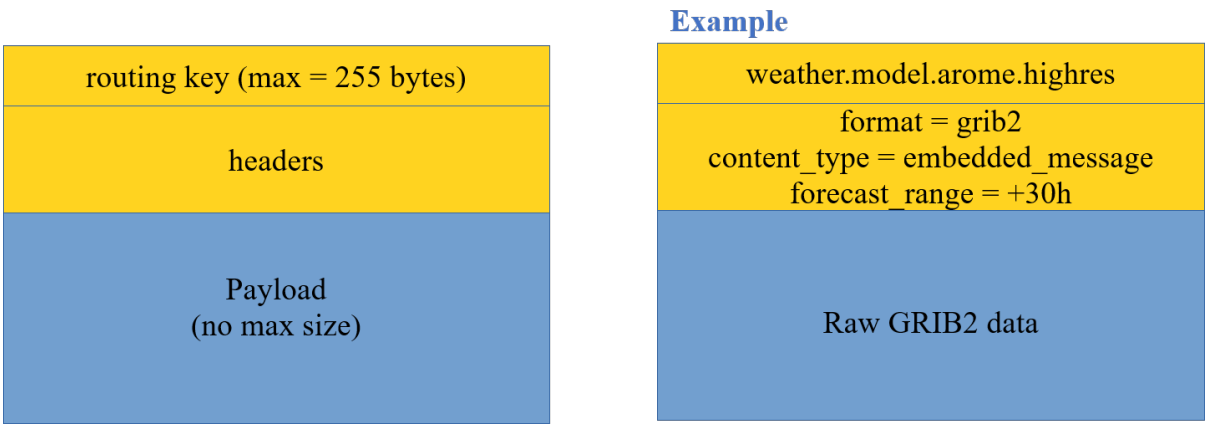



*Figure 16. OWS Publish/Subscribe FIG2*

Four types of exchanges are being considered for OpenWIS, including Direct, Fanout, Topic, and Header exchanges. Figure Figure 17 describes each of these exchanges

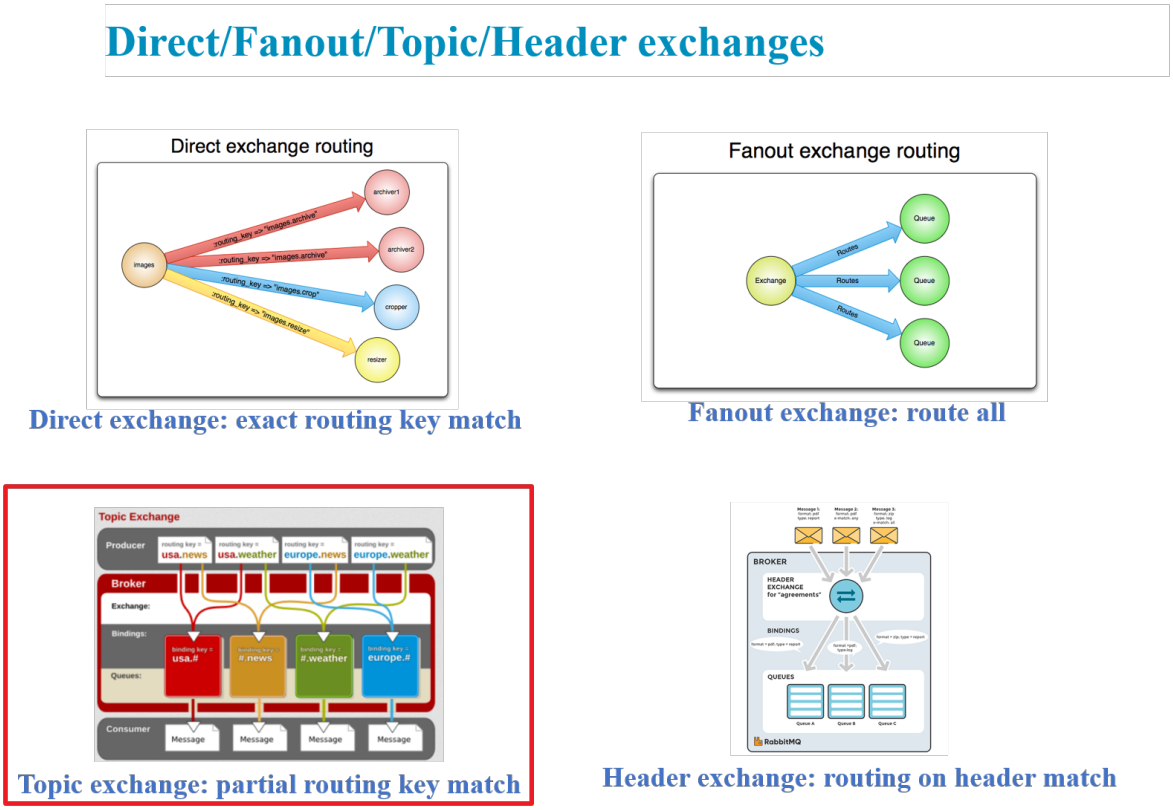



*Figure 17. OWS Publish/Subscribe FIG3*

RabbitMQ tunable settings used during WIS2 study: - Make queues lazy - Store messages on disk (predictable response times) - Make queues persistent - Allows to route messages even if consumer not connected - Tune prefetch - Allows to use all bandwidth even with on high latency WAN - Need to be mindful of big files - Be careful when sending send files bigger than a few Kbs RAM Impact (files loaded to RAM) Transfer times

The proposed message format used during the WIS2 study can be found in figure Figure 18

```
MANDATORY:
        "pubTime"       - YYYYMMDDTHHMMSS.ss - UTC date/timestamp.
        "baseUrl"       - root of the url to download.
        "relPath"       - relative path can be catenated to <base_url>
        "integrity"     - WMO version of v02 sum field, under development.
        {
           "method" : "md5" | "sha512" | "md5name" | "link" | "remove" | "cod" | "random" ,
           "value"  : "base64 encoded checksum value"
        }
OPTIONAL:
        "size"          - the number of bytes being advertised.
        "blocks"        - if the file being advertised is partitioned, then:
        {
            "method"    : "inplace" | "partitioned" , - is the file already in parts?
            "size"      : "9999", - size of the blocks.
            "count"     : "9999", - number of blocks in the file.
            "remainder" : "9999", - the size of the last block.
            "number"    : "9999", - which block is this.
        }
        "rename"        - name to write file locally.
        "topic"         - copy of topic from AMQP header (usually omitted)
        "source"        - the originating entity of the message.
        "from_cluster"  - the originating cluster of a message.
        "to_clusters"   - a destination specification.
        "link"          - value of a symbolic link. (if sum starts with L)
        "atime"         - last access time of a file (optional)
        "mtime"         - last modification time of a file (optional)
        "mode"          - permission bits (optional)
        "content"       - for smaller files, the content may be embedded.
        {
            "encoding" : "utf-8" | "base64"  ,
            "value"    " "encoded file content"
        }
        For "v03.report" topic messages the following addtional
        headers will be present:
        "report"   - status report field documented in `Report Messages`_
        "message"  - status report message documented in `Report Messages`_
        additional user defined name:value pairs are permitted.
```

*Figure 18. OWS Publish/Subscribe FIG4*

A single Node PubSub implementation can be found in figure [clause7_figure5].

The results of the study found: 1. When you compare Reference vs Direct message dissemination methods, large message always need to be disseminated by reference 2. For small messages there are questions, including where or not to incude messages into payload. It appears to be difficult to set a threshold. This leads to a more complicated protocol, and also leads to less predictable message rate. And finally, it leads to potential issues with memory management 3. The conclusion was to not include message into payload.

The next steps in the study include: 1. Agree PubSub tree. It must be TTAAiiCCCC based. It must be data types based, productor based, or a mix. 2. Need to stabilize the message format 3. Conduct a larger scale test

The key for this is to think about how this could be implemented in the frame of WIS/GTS * Need Pub/Sub for “GISCs” * Need Pub/Sub for “Ncs” * Need SLA / Key Performance Indicators * Need Metadata

# Chapter 8. Air Traffic Control Simulators

Kongsberg I-SIM® was originally developed for the Federal Aviation Administration (FAA) as the Air Traffic Control Advanced Research Simulator (ATCARS). I-SIM is a high-fidelity simulation system used for Air Traffic Control Training, Air Space design/analysis, advanced Computer-Human Interface (CHI) and Human-Machine Interface (HMI) development, and to support Unmanned Aviation System (UAS) integration into civil airspace. I-SIM is also used to model high-density airspaces for commercial airspace design.

I-SIM uses InterCOM DDS (Data Distribution Service) as the middleware solution for sharing data between its components. InterCOM DDS is an implementation of the Object Management Group (OMG) Data Distribution Service whose purpose is to abstract away the mechanics of transport and persistence while distributing data among distributed nodes.

DDS is a networking middleware that simplifies complex network programming. It implements a publish–subscribe pattern for sending and receiving data, events, and commands among the nodes. Nodes that produce information (publishers) create "topics" (*e.g.*, telemetry, location) and publish "samples". DDS delivers the samples to subscribers that declare an interest in the topic.

DDS handles transfer chores: message addressing, data marshalling and de-marshalling (so subscribers can be on different platforms from the publisher), delivery, flow control, retries, *etc.* Any node can be a publisher, subscriber, or both simultaneously. The DDS publish-subscribe model virtually eliminates complex network programming for distributed applications.

DDS supports mechanisms that go beyond the basic publish-subscribe model. The key benefit is that applications that use DDS for their communications are decoupled. Little design time needs to be spent on handling their mutual interactions. In particular, the applications never need information about the other participating applications, including their existence or locations. DDS transparently handles message delivery without requiring intervention from user applications, including:

- determining who should receive the messages
- determining where recipients are located
- what happens if messages cannot be delivered

DDS allows the user to specify Quality of Service (QoS) parameters to configure discovery and behaviour mechanisms up-front. By exchanging messages anonymously, DDS simplifies distributed applications and encourages modular well-structured programs. DDS also automatically handles hot-swapping redundant publishers if the primary fails. Subscribers always get the sample with the highest priority whose data is still valid (that is, whose publisher-specified validity period has not expired). DDS automatically switches back to the primary when it recovers, too.

At the core of the product is a Supervisor which generates track data and beacon messages. The I-SIM Track XML message is sent from the I-SIM Supervisor for each current flight once per second and contains flight plan information as well as detailed aircraft state data. Detailed state data is primarily used in the I-SIM Pseudo-Pilot and Instructor applications. Beacon messages are simulated radar contacts and are used to simulate ground radars. The I-SIM Beacon message is sent from the I-SIM Supervisor for each current flight once per radar sweep (or once per second for Automatic Dependent Surveillance–Broadcast (ADS–B) targets) representing the radar and/or ADS-B

target and includes position and transponder data. Transponder data generally includes Mode 3/A and Mode C (altimeter corrected) codes. I-SIM radar processing supports ADS-B, Primary Surveillance Radar (PSR) and Secondary Surveillance Radar (SSR) returns. The information is published on the DDS bus and participants that subscribe to these topics will receive it and decode it.

The controller displays a replica of the En-Route displays used by FAA and both Standard Terminal Automation Replacement System (STARS) terminal control and En-Route Automation Modernization (ERAM) are supported. STARS replaces outdated equipment in Terminal Radar Approach CONtrol (TRACON) and tower facilities. It provides controllers with critical operational information about aircraft positions, flight data, and weather. STARS also provides many new capabilities that allow technical operations personnel to monitor and control system resources.

Messages published by the Supervisor are processed and information is displayed on the screen. The operator can modify the information and resend the message using DDS. The pseudo-pilot receives via DDS requests from the operator using the controller display and will notify the supervisor to change the behaviour of the simulated aircraft.

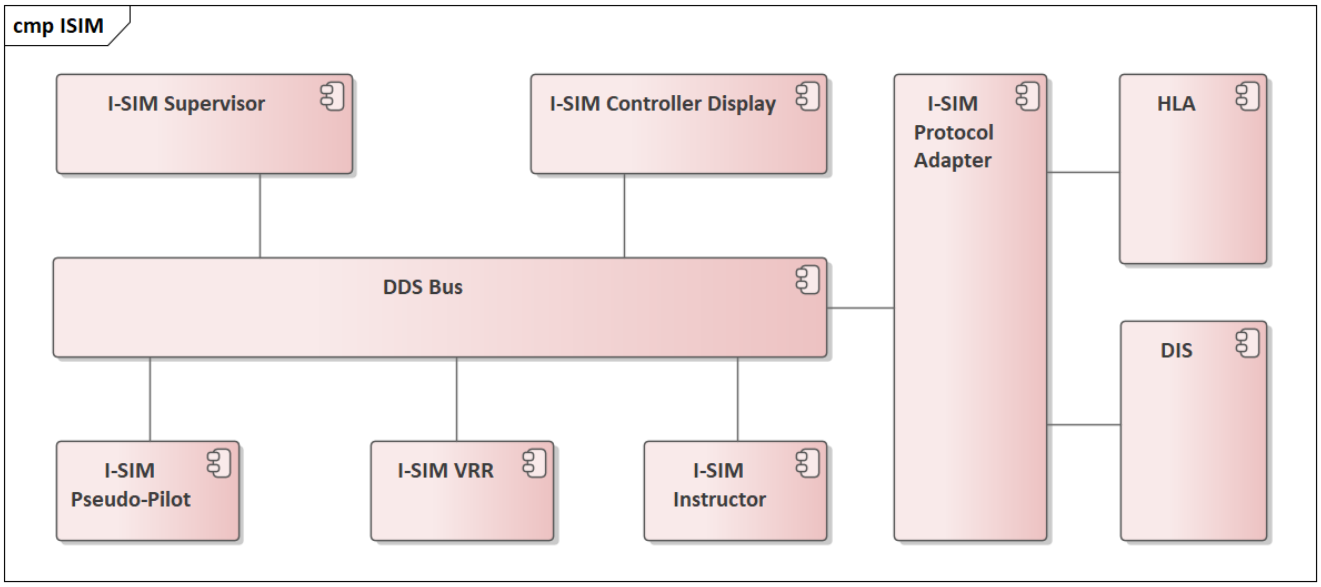



*Figure 19. I-SIM® components diagram*

I-SIM® (Figure 19) includes a Voice Recognition/Response (VRR) module, providing an automated alternative for live pseudo-pilot operators when used in Air Traffic Management (ATM) / Air Traffic Control (ATC) training. I-SIM VRR accepts verbal commands from the air traffic controller and executes them as the aircraft should, providing voice responses to the operator as it does so. The VRR module requires very little voice recognition “training” when being used by a new operator, and provides nearly 98% comprehension of voice commands with most operators out of the box.

I-SIM VRR is a tool for training environments where there is limited space or personnel, and is currently in use as part of the Air Traffic Control Training System on all U.S. Navy aircraft carriers and amphibious assault ships.

Across the voice recognition software and Voice over Internet Protocol (VoIP) industry, the DDS architecture is not considered mature and fast enough to be utilized as a transport layer in such a simulator. In that context, DDS would introduce “voice jitter” and related delays, which would render the audio information contained in the DDS messages unusable thus impacting VRR’s

usefulness as a training system.

For the voice recognition and audio components to work properly, the messages must arrive in the right order/sequence to be understood by trainees. Any delays or incorrect sequencing of messages can cause “voice jitter” or inaudible voice commands which would severely impact training (audio portions of the training exercise would be useless).

Kongsberg Geospatial extended InterCOM (Intercommunication) DDS to implement a new communication messaging process to remove any ‘voice jitter’ experienced by the U.S. Navy. I-SIM includes a protocol adapter allowing DDS messages to be translated to and from other simulation standards such as Distributed Interactive Simulation (DIS) IEEE (Institute of Electrical and Electronics Engineer) Standard 1278 and High-Level Architecture (HLA) IEEE Standard 1516. By using this approach, customers can include their own simulators and provide more efficient training.

# Chapter 9. AsyncAPI description

OGC Web API Standards build on the design patterns described by the OpenAPI Interface Design Language (IDL). The use of an IDL has proven invaluable in the development of OGC Web API Standards. OpenAPI provides a set of common concepts for the design of a Web API and rules for the application of those concepts. Without this common language, the task of coordinating multiple API standardization efforts would be almost impossible.

The OGC is now looking to extend our Request-Response APIs to include event driven (asynchronous) operations. However, OpenAPI does not support event driven operations beyond a simple call-back model. So an IDL for event driven Web APIs is needed.

AsyncAPI is an open source initiative that seeks to improve the current state of Event-Driven Architectures (EDA). Their long-term goal is to make working with EDA's as easy as it is to work with REST APIs. That goes from documentation to code generation, from discovery to event management. Most of the processes used for REST APIs should also be applicable to event-driven/asynchronous APIs. This section provides an introduction to AsyncAPI as an additional tool for use in developing OGC Web API standards.

## 9.1. Concepts

### 9.1.1. Application

An application is any kind of computer program or a group of them. It MUST be a producer, a consumer or both. An application MAY be a microservice, IoT device (sensor), mainframe process, etc. An application MAY be written in any number of different programming languages as long as they support the selected protocol. An application MUST also use a protocol supported by the server in order to connect and exchange messages.

### 9.1.2. Bindings

A "binding" (or "protocol binding") is a mechanism to define protocol-specific information. Therefore, a protocol binding MUST define protocol-specific information only.

### 9.1.3. Channel

A channel is an addressable component, made available by the server, for the organization of messages. Producer applications send messages to channels and consumer applications consume messages from channels. Servers MAY support many channel instances, allowing messages with different content to be addressed to different channels. Depending on the server implementation, the channel MAY be included in the message via protocol-defined headers.

### 9.1.4. Consumer

A consumer is a type of application, connected to a server via a supported protocol, that is consuming messages from channels. A consumer MAY be consuming from multiple channels depending on the server, protocol, and the use-case pattern.

### 9.1.5. Message

A message is the mechanism by which information is exchanged via a channel between servers and applications. A message MUST contain a payload and MAY also contain headers. The headers MAY be subdivided into protocol-defined headers and header properties defined by the application which can act as supporting metadata. The payload contains the data, defined by the application, which MUST be serialized into a format (JSON, XML, Avro, binary, etc.). Since a message is a generic mechanism, it can support multiple interaction patterns such as event, command, request, or response.

### 9.1.6. Producer

A producer is a type of application, connected to a server, that is creating messages and addressing them to channels. A producer MAY be publishing to multiple channels depending on the server, protocol, and use-case pattern.

### 9.1.7. Protocol

A protocol is the mechanism (wireline protocol or API) by which messages are exchanged between the application and the channel. Example protocols include, but are not limited to, AMQP, HTTP, JMS, Kafka, MQTT, STOMP, Mercure, WebSocket.

## 9.2. The AsyncAPI Specification

AsyncAPI can be viewed as an extension of OpenAPI. Many of the objects and structures defined in OpenAPI can also be found in AsynAPI. However, there are some important differences.

### 9.2.1. AsyncAPI Object

The AsyncAPI Object is similar to the OpenAPI Object. This object organizes the top-level components of the AsyncAPI file. Table 1 describes the AsyncAPI object and how it differs from its' OpenAPI counterpart.

*Table 1. AsyncAPI Object*

| Field Name | Type | Description |
|---|---|---|
| asyncapi | AsyncAPI Version String | **Required.** Serves the same function as the openAPI field. The format of the asyncAPI field is defined by the schema. OpenAPI uses the same format, but defines it in the text of the standard. |
| id | Identifier | **Specific to AsyncAPI.** Identifier of the application the AsyncAPI document is defining. |
| info | Info Object | **Required.** Same as OpenAPI. |
| servers | Servers Object | Differs from OpenAPI. In OpenAPI this is an array of Server Objects while in AsyncAPI it is a map of Server Objects. The structure of the Server Objects themselves also differ. |

| Field Name | Type | Description |
|---|---|---|
| channels | Channels Object | **Required**. Channels are similar to OpenAPI Paths in that they identify a resource. The main difference is that a path is used to access the current resource while a channel is used to distribute all instances of that resource over a period of time. |
| components | Components Object | The Components Object serves the same function in AsyncAPI as it does in OpenAPI. The categories of components managed, however, are very different. See the Components Object section for more information. |
| tags | Tags Object | Same as in OpenAPI. |
| externalDocs | External Documentation Object | Same as in OpenAPI. |

It's notable that AsyncAPI does not include Security Requirements in the root file. Instead, security is addressed by the Server Objects.

### 9.2.2. Components Object

The AsyncAPI Components Object provides the same functionality and structure as the Components Object in OpenAPI. The difference is in the types of components that can be included.

Table 2 describes the Components object and how it differs from its' OpenAPI counterpart.

*Table 2. Components Object*

| Field Name | Type | Description |
|---|---|---|
| schemas | Map[`string`, Schema Object OR Reference Object] | An object to hold reusable Schema Objects. Similar to OpenAPI save that they use different versions of JSON Schema. We can expect them to converge on one version in the future. |
| messages | Map[`string`, Message Object OR Reference Object] | **Specific to AsyncAPI** An object to hold reusable Message Objects. |
| securitySchemes | Map[`string`, Security Scheme Object OR Reference Object] | The Server Scheme Objects are nearly identical to those of OpenAPI. They differ only in the valid values for the `type` and `in` fields |

| Field Name | Type | Description |
|---|---|---|
| parameters | Map[`string`, Parameter Object OR Reference Object] | An object to hold reusable Parameter Objects. AsyncAPI Parameter Objects differ significantly from those of OpenAPI. |
| correlationIds | Map[`string`, Correlation ID Object] | **Specific to AsyncAPI** An object to hold reusable Correlation ID Objects. |
| operationTraits | Map[`string`, Operation Trait Object] | **Specific to AsyncAPI** An object to hold reusable Operation Trait Objects. |
| messageTraits | Map[`string`, Message Trait Object] | **Specific to AsyncAPI** An object to hold reusable Message Trait Objects. |
| serverBindings | Map[`string`, Server Binding Object] | **Specific to AsyncAPI** An object to hold reusable Server Binding Objects. |
| channelBindings | Map[`string`, Channel Binding Object] | **Specific to AsyncAPI** An object to hold reusable Channel Binding Objects. |
| operationBindings | Map[`string`, Operation Binding Object] | **Specific to AsyncAPI** An object to hold reusable Operation Binding Objects. |
| messageBindings | Map[`string`, Message Binding Object] | **Specific to AsyncAPI** An object to hold reusable Message Binding Objects. |

### 9.2.3. Server Objects

Servers are described by two objects; the Servers Object (plural) and the Server Object (singular).

A Servers Object provides a map of Server Objects with each Server Object refrenced by an identifier (the map key). This differs from OpebnAPI where the Server Objects are packaged as an array.

Table 3 describes the Server object and how it differs from its' OpenAPI counterpart.

*Table 3. Server Object*

| Field Name | Type | Description |
|---|---|---|
| url | `string` | **REQUIRED**. Same as in OpenAPI. |
| protocol | `string` | **REQUIRED**, **Specific to AsyncAPI**. The protocol this URL supports for connection. Supported protocol include, but are not limited to: `amqp`, `amqps`, `http`, `https`, `jms`, `kafka`, `kafka-secure`, `mqtt`, `secure-mqtt`, `stomp`, `stomps`, `ws`, `wss`, `mercure`. |
| protocolVersion | `string` | **Specific to AsyncAPI**. The version of the protocol used for connection. For instance: AMQP `0.9.1`, HTTP `2.0`, Kafka `1.0.0`, etc. |
| description | `string` | Same as OpenAPI. |
| variables | Map[`string`, Server Variable Object] | Same as OpenAPI except that the AsyncAPI includes an `examples` field in the Server Variable Object. |
| security | [Security Requirement Object] | **Specific to AsyncAPI**. AsyncAPI applies security requirements in the Server Object while OpenAPI applies them on the Path. The Server Scheme Objects are nearly identical, differing only in the valid values for the `type` and `in` fields. |
| bindings | Server Bindings Object | **Specific to AsyncAPI**. A map where the keys describe the name of the protocol and the values describe protocol-specific definitions for the server. |

### 9.2.4. Channel Object

Channels are described by two objects; the Channels Object (plural) and the Channel Object (singular).

A Channels Object provides a map of Channel Objects with each Channel Object refrenced by an identifier (the map key).

Table 4 describes the Channel object.

*Table 4. Channel Object*

| Field Name | Type | Description |
|---|---|---|
| $ref | `string` | Allows for an external definition of this channel item. The referenced structure MUST be in the format of a Channel Item Object. If there are conflicts between the referenced definition and this Channel Item's definition, the behavior is *undefined*. |
| description | `string` | An optional description of this channel item. CommonMark syntax can be used for rich text representation. |

| Field Name | Type | Description |
|---|---|---|
| subscribe | Operation Object | A definition of the SUBSCRIBE operation. |
| publish | Operation Object | A definition of the PUBLISH operation. |
| parameters | Parameters Object | A map of the parameters included in the channel name. It SHOULD be present only when using channels with expressions (as defined by RFC 6570 section 2.2). |
| bindings | Channel Bindings Object | A map where the keys describe the name of the protocol and the values describe protocol-specific definitions for the channel. |

### 9.2.5. Channel Item Object

The Channel Item Object is similar to the Path Item Object in OpenAPI. Both describe the available operations (publish and subscribe vs. the HTTP verbs) and the parameters for those operations. They also differ in two other major ways:

- Channel Item Object includes a `bindings` field. This field is not needed in the Path Items object since the binding is always HTTP.
- Path Items Objects include a `servers` field to identify the target of requests.

Table 5 describes the Channel Item object.

*Table 5. Channel Item Object*

| | | |
|---|---|---|
| **$ref** | `string` | **Allows for an external definition of this channel item. The referenced structure MUST be in the format of a Channel Item Object. If there are conflicts between the referenced definition and this Channel Item's definition, the behavior is *undefined*.** |
| description | `string` | An optional description of this channel item. CommonMark syntax can be used for rich text representation. |
| subscribe | Operation Object | A definition of the SUBSCRIBE operation. |
| publish | Operation Object | A definition of the PUBLISH operation. |
| parameters | Parameters Object | A map of the parameters included in the channel name. It SHOULD be present only when using channels with expressions (as defined by RFC 6570 section 2.2). |
| bindings | Channel Bindings Object | A map where the keys describe the name of the protocol and the values describe protocol-specific definitions for the channel. |

### 9.2.6. Operation Object

The AsyncAPI Operation Object serves in a similar role as the OpenAPI Operation Object, but requires a very different set of data to do so.

Table 6 describes the Operation object and how it differs from its' OpenAPI counterpart.

*Table 6. Operation Object*

| Field Name | Type | Description |
|---|---|---|
| operationId | `string` | Same as in OpenAPI. |
| summary | `string` | Same as in OpenAPI. |
| description | `string` | Same as in OpenAPI |
| tags | [Tag Object] | Similar to OpenAPI. In OpenAPI the tags are simple strings. |
| externalDocs | External Documentation Object | Same as in OpenAPI. |
| bindings | Operation Bindings Object | **Specific to AsyncAPI**. A map where the keys describe the name of the protocol and the values describe protocol-specific definitions for the operation. |
| traits | Operation Trait Object OR Reference Object | **Specific to AsyncAPI**. A list of traits to apply to the operation object. Traits MUST be merged into the operation object using the JSON Merge Patch algorithm in the same order they are defined here. |
| message | Message Object Reference Object | **Specific to AsyncAPI**. A definition of the message that will be published or received on this channel. `oneOf` is allowed here to specify multiple messages, however, **a message MUST be valid only against one of the referenced message objects.** |

The AsyncAPI Operation Object excludes a number of fields which are defined in the OpenAPI Operation Object:

- parameters
- requestBody
- responses
- callBacks
- deprecated
- security
- servers

### 9.2.7. Server Binding Object

The Server Binding Object is specific to AsyncAPI.

This object is a map describing protocol-specific definitions for a server. These objects are specific to AsyncAPI.

Table 7 describes the Server Binding object.

*Table 7. Server Binding Object*

| Field Name | Type | Description |
|---|---|---|
| `http` | HTTP Server Binding | Protocol-specific information for an HTTP server. |
| `ws` | WebSockets Server Binding | Protocol-specific information for a WebSockets server. |
| `kafka` | Kafka Server Binding | Protocol-specific information for a Kafka server. |
| `amqp` | AMQP Server Binding | Protocol-specific information for an AMQP 0-9-1 server. |
| `amqp1` | AMQP 1.0 Server Binding | Protocol-specific information for an AMQP 1.0 server. |
| `mqtt` | MQTT Server Binding | Protocol-specific information for an MQTT server. |
| `mqtt5` | MQTT 5 Server Binding | Protocol-specific information for an MQTT 5 server. |
| `nats` | NATS Server Binding | Protocol-specific information for a NATS server. |
| `jms` | JMS Server Binding | Protocol-specific information for a JMS server. |
| `sns` | SNS Server Binding | Protocol-specific information for an SNS server. |
| `sqs` | SQS Server Binding | Protocol-specific information for an SQS server. |

| Field Name | Type | Description |
|---|---|---|
| `stomp` | STOMP Server Binding | Protocol-specific information for a STOMP server. |
| `redis` | Redis Server Binding | Protocol-specific information for a Redis server. |
| `mercure` | Mercure Server Binding | Protocol-specific information for a Mercure server. |

### 9.2.8. Channel Binding Object

The Channel Binding Object is specific to AsyncAPI.

The Channel Binding Object and Server Binding Object are almost identical. They support the same protocols. However, there may be differences in the protocol details.

### 9.2.9. Message Binding Object

The Message Binding Object is specific to AsyncAPI.

The Message Binding Object and Server Binding Object are almost identical. They support the same protocols. However, there may be differences in the protocol details.

### 9.2.10. Operation Binding Object

The Operation Binding Object is specific to AsyncAPI.

The Operation Binding Object and Server Binding Object are almost identical. They support the same protocols. However, there may be differences in the protocol details.

### 9.2.11. Parameters

Parameters are described by two objects; the *Parameters Object* (plural) and the *Parameter Object* (singular).

A *Parameters Object* provides a map of *Parameter Objects* while each *Parameter Object* referenced by an identifier (the map key).

AsyncAPIs are considerably simpler than those defined in OpenAPI; So, they should be treated as different objects.

Table 8 describes the AsyncAPI *Parameter Object*.

*Table 8. Parameter Object*

| Field Name | Type | Description |
|---|---|---|
| description | `string` | A verbose explanation of the parameter. CommonMark syntax can be used for rich text representation. |
| schema | Schema Object | Definition of the parameter. |
| location | `string` | A runtime expression that specifies the location of the parameter value. Even when a definition for the target field exists, it MUST NOT be used to validate this parameter but, instead, the `schema` property MUST be used. |

### 9.2.12. Message Object

Logically, an OpenAPI Response Object and an AsyncAPI Message Object are expected to be very similar. In practice, the information exchanged may be similar too; however, the information needed to describe that exchange is very different. Thus, the Message Object bears little resemblance to its OpenAPI counterpart.

Table 9 describes the Message object.

*Table 9. Message Object*

| Field Name | Type | Description |
|---|---|---|
| headers | Schema Object Reference Object | Schema definition of the application headers. Schema MUST be of type "object". It **MUST NOT** define the protocol headers. |
| payload | `any` | Definition of the message payload. It can be of any type but defaults to Schema object. |
| correlationId | Correlation ID Object Reference Object | Definition of the correlation ID used for message tracing or matching. |
| schemaFormat | `string` | A string containing the name of the schema format used to define the message payload. If omitted, implementations should parse the payload as a Schema object. Check out the supported schema formats table for more information. Custom values are allowed but their implementation is OPTIONAL. A custom value MUST NOT refer to one of the schema formats listed in the table. |
| contentType | `string` | The content type to use when encoding/decoding a message's payload. The value MUST be a specific media type (e.g. `application/json`). When omitted, the value MUST be the one specified on the defaultContentType field. |
| name | `string` | A machine-friendly name for the message. |

| Field Name | Type | Description |
|---|---|---|
| title | `string` | A human-friendly title for the message. |
| summary | `string` | A short summary of what the message is about. |
| description | `string` | A verbose explanation of the message. CommonMark syntax can be used for rich text representation. |
| tags | Tags Object | A list of tags for API documentation control. Tags can be used for logical grouping of messages. |
| externalDocs | External Documentation Object | Additional external documentation for this message. |
| bindings | Message Bindings Object | A map where the keys describe the name of the protocol and the values describe protocol-specific definitions for the message. |
| examples | Map [`string`, `any`] | An array with examples of valid message objects. |
| traits | Message Trait Object Reference Object | A list of traits to apply to the message object. Traits MUST be merged into the message object using the JSON Merge Patch algorithm in the same order they are defined here. The resulting object MUST be a valid Message Object. |

## 9.3. Resources

The following resources should be explored if you wish to learn more about AsyncAPI, design an asynchronous API, or contribute to the standard:

- AsyncAPI Home
- Getting Started
- OpenAPI Version 2.0
- AsyncAPI GitHub

# Chapter 10. Future Work

The next step the OGC should take is to establish a common approach to asynchronous Web APIs. Ideally, a Pub/Sub extension to OGC API-Common. This would provide a standard API module which could be extended to define Pub/Sub capabilities for existing and developmental OGC Web API standards.

The family of OGC Web API Standards is built on the architecture patterns represented by OpenAPI. AsyncAPI is an extension of OpenAPI. So the first step should be to identify or develop a JSON schema which integrates OpenAPI and AsyncAPI. This API definition language will be used to guide further standards activities.

Once an API definition language has been identified, the next step will be to develop the directions on how that language should be applied to creating Pub/Sub extensions to OGC Web APIs. These directions would be documented as a draft extension to OGC API-Common.

The draft Pub/Sub extension would then be evaluated against existing and planned asynchonous OGC Web API and Web Service standards. Based on this analysis, the draft extension would be refined to maximize its' effectiveness and utility.

The final step would be to publish the refined specification as a new OGC Web API module.

Action: The OGC Web API-Common SWG Charter is currently under review. This work item should be added to the program of work for that SWG.

# Annex A: Revision History

| Date | Release | Editor | Primary clauses modified | Description |
|---|---|---|---|---|
| 2020-06-10 | 1 | Sara Saeedi | all | initial version |
| 2020-09-15 | 1.1 | Sara Saeedi | all | Final edits |

# Annex B: Bibliography

[2] L. Bigagli, M. Rieke. The new OGC Publish/Subscribe Standard - applications in the Sensor Web and the Aviation domain. Open Geospatial Data, Software and Standards, ISSN: 2363-7501, doi: 10.1186/s40965- 017-0030-7, vol. 2, issue no. 1, article 18, Springer, Heidelberg, Germany, December 2017 (electronic 3 August 2017).

[3] M. Rieke, L. Bigagli, S. Herle, S. Jirka, A. Kotsev, T. Liebig, C. Malewski, T. Paschke, C. Stasch. Geospatial IoT—The Need for Event-Driven Architectures in Contemporary Spatial Data Infrastructures. ISPRS International Journal of Geo-Information (IJGI), ISSN 2220-9964, Volume 7, Issue 10, article 385, doi: 10.3390/ijgi7100385, MDPI, Basel, Switzerland, 25 September 2018.

[4] OGC. OGC Publish/Subscribe Interface Standard 1.0 — Core, 1st ed.; Open Geospatial Consortium: Wayland, MA, USA, 2016.

[5] OGC. OGC Publish/Subscribe Interface Standard 1.0 - SOAP Protocol Binding Extension, 1st ed.; Open Geospatial Consortium: Wayland, MA, USA, 2016.

[6] Graham, S.; Hull, D.; Murray, B. Web Services Base Notification 1.3; Technical Report; OASIS: Burlington, MA, USA, 2016.

[7] Bigagli, L.; Vretanos, P.P.A.; Lawrence, M.; Papeschi, F.; Martell, R. Testbed-12 PubSub/Catalog Engineering Report; Technical Report OGC 16-137r2; Open Geospatial Consortium: Wayland, MA, USA, 2017.