Open Geospatial Consortium

Publication Date: 2015-08-19

Approval Date: 2015-06-05

Posted Date: 2015-05-12

Reference number of this document: OGC 15-028

Reference URL for this document: http://www.opengis.net/doc/PER/t11-data-broker-aviation

Category: Public Engineering Report

Editor: Daniel Balog

Testbed-11 Data Broker Specifications Engineering Report

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

Contents

Page

Figures

Page

Tables

Page

Abstract

This document is a deliverable of the OGC Testbed 11 Interoperability initiative. The report's contents cover the summary of the interoperability work regarding the Aviation Data Broker concept. This data broker concept enables the setup of cascading OGC Web Feature Server (WFS) servers to form a data source chain, in which one service is capable of providing information coming from one or more other services. The objectives of this document are to research the feasibility of this concept and to investigate a number of specific Data Broker responsibilities and use cases, such as provenance and lineage, conflation, caching, scalability and flexible management of data sources.

Business Value

Previous OGC Interoperability initiatives have extensively trialed the use of the WFS for the provision of aeronautical data, in which the WFS was typically backed by a database. This resulted in a flat architecture, including one or more WFS data sources providing aeronautical data to applications.

In a realistic deployment scenario, WFS solutions might be applied at more than one level in the data chain, and some data might not always be replicated at higher levels. For instance, WFS data sources could be set up at a national level, from where data might be further centralized at regional levels and made accessible to end-user systems.

This is where the Data Broker concept and the research in Testbed 11 provides its value, i.e. by bringing a solution on how standard WFS components can be cascaded and form a data source chain.

Keywords

ogcdocs, ogc documents, testbed-11, wfs, broker, aggregation, conflation, lineage, provenance, aviation

Testbed-11 Data Broker Specifications Engineering Report

1 Introduction

1.1 Scope

This OGC® Engineering Report (ER) describes the development and testing of a "Data Broker" concept based on OGC technology. The overall goal is to enable the setup of cascading OGC web services to form a data source chain, in which one service is capable of providing information coming from other services. This document specifically focuses on the OGC Web Feature Service (WFS), OGC's primary web service interface standard to request feature data. This document includes the following content:

- An overview of an architecture blueprint explaining which technologies need to be combined to enable the Data Broker concept.
- A discussion on Data Broker aspects like data provenance, conflation, data caching, scalability and flexible management of data sources.
- A description of a prototype implementation, as well as some specific testing with respect to the scalability of the concept.

1.2 Document contributor contact points

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

Name	Organization
Daniel Balog	Luciad
Robin Houtmeyers	Luciad

1.3 Future work

Improvements in this research are desirable for the following topics:

- Prototyping of an implementation of the flexible sources management approach discussed in Section 6.5, using ISO 19119 and OGC Catalog Services – Web (CSW).
- Investigation of the use of semantic mediation in cases where multiple data formats (e.g., AIXM & AFX) need to be merged into a single coherent set of data elements.
- Investigation of the mediation of WFS sources with different coordinate systems. This ER focuses on the use of the WGS 84 single coordinate system, as put forward in [5] and [6].
- Investigation of the generation of more detailed queries to WFS sources that exclude the retrieval of features that have already been cached in the Data Broker, based on their UUID.
- Investigation of merging duplicate data automatically in case features from different WFS sources refer to the same concept, but don't share the same UUID.
- Investigation of the use of WFS-TE (Temporality Extension) to reduce bandwidth usage between the Data Broker and its WFS sources and to improve caching performance for outdated data. The investigation should look at how WFS-TE can be used to get new updates to AIXM features only instead of retrieval of full AIXM features along with their entire history. At the time of writing, the WFS-TE discussion paper is being revised.

1.4 Forward

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium 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.

2 References

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

OGC 06-121r3, *OGC® Web Services Common Standard*

NOTE This OWS Common Standard contains a list of normative references that are also applicable to this Implementation Standard.

ISO 19115:2003 & 19115-1:2014, *Geographic information - Metadata*

ISO 19119:2005, *Geographic information - Services*

ISO 19142:2010 / OGC 09-025r2, *OGC® Web Feature Service 2.0 Interface Standard*

OGC 12-027r3, *OGC® WFS-Temporality Extension Discussion Paper*

OGC 12-028, *Use of GML for aviation data*

AIXM version 5.1, *Feature Identification and Reference*

ICAO Annex 15 (13th Edition), *Aeronautical Information Services*

3 Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard [OGC 06-121r3] shall apply.

4 Conventions

4.1 Abbreviated terms

AFX	Aviation Feature Schema
AIXM	Aeronautical Information eXchange Model
API	Application Programming Interface
CSW	Catalogue Service for the Web
UUID	Universally Unique IDentifier
WFS	Web Feature Service
WFS-TE	Web Feature Service Temporality Extension

4.2 UML notation

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

5 Architecture and design

5.1 Overview

The OGC Web Feature Service (WFS) is OGC's and ISO's primary web service standard to access geographic features in a manner independent of the underlying data store. Previous OGC test beds have extensively trialed the use of the WFS for the provision of aeronautical data, in which the WFS was typically backed by a database. This approach resulted in a flat architecture, including one or more WFS data sources providing aeronautical data to applications.

In a real deployment, WFS solutions might be applied at more than one level in the data chain, and some data might not always be replicated at higher levels. For instance, WFS data sources could be set up at a national level, from where data might be further centralized at regional levels and made accessible to end-user systems. This is where the data broker concept comes into play; its overall goal is to see how WFS components can be cascaded and form a data source chain. Figure 1: Conceptual architecture of the Data Broker shows a conceptual architecture to indicate how data brokers can be used in practice to exchange aeronautical data.

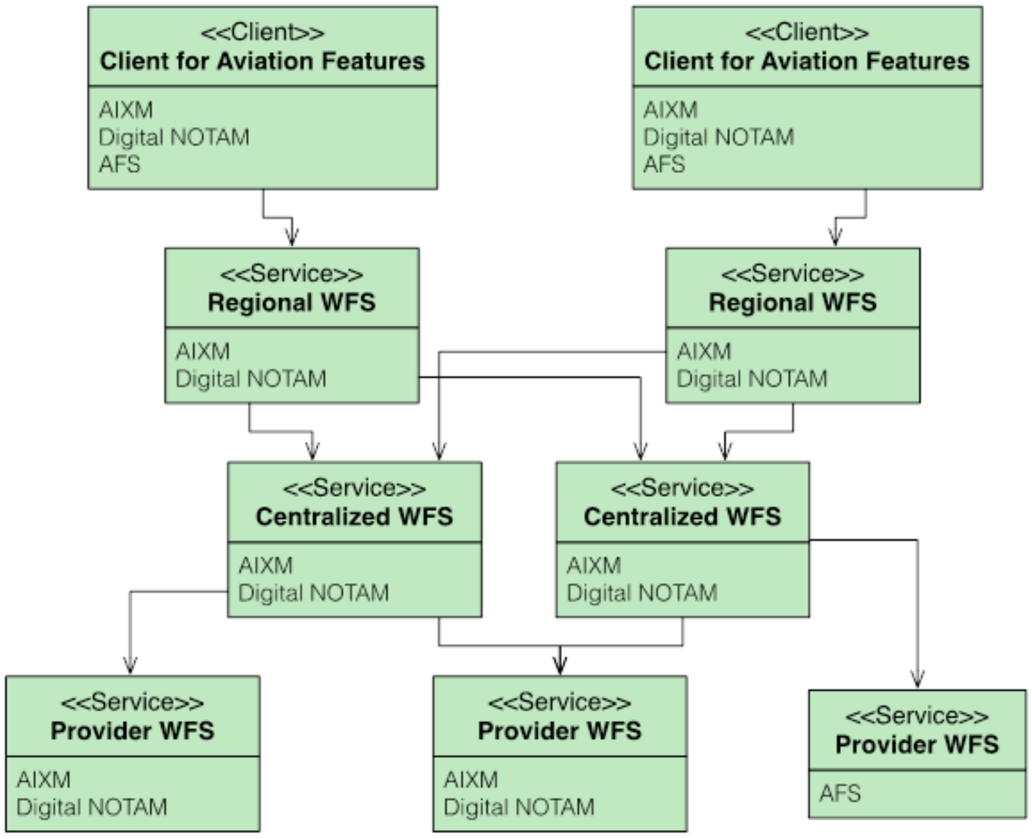

Figure 1: Conceptual architecture of the Data Broker

The following sections will further discuss the architecture and design of the Data Broker. Chapter 6 will then go into more detail about various aspects related to the implementation and operational behavior of the Data Broker.

5.2 Scope & requirements

By definition, a WFS Broker provides access to information supplied by one or more WFS data sources. This is illustrated by the high-level architecture diagram in Figure 2: High-level architecture of the WFS Data Broker (1).

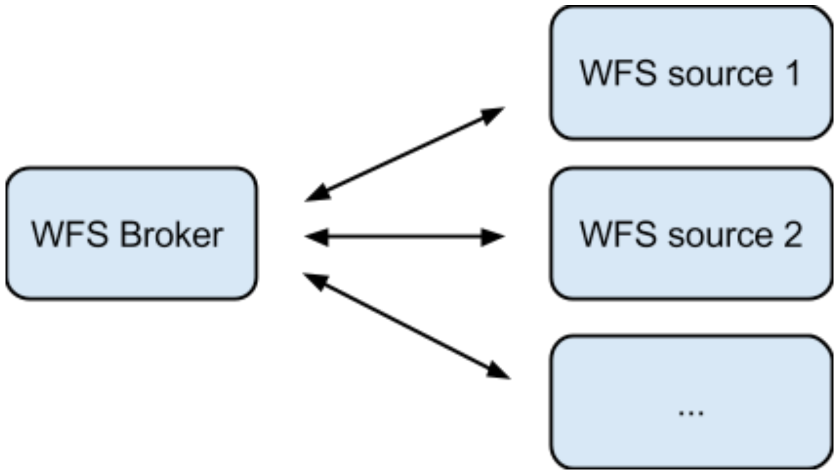

Figure 2: High-level architecture of the WFS Data Broker (1)

A very simple implementation of a WFS Data Broker could just offer a comprehensive list of all available feature types, each backed by one WFS data source. With that approach, the Data Broker would merely act as a pass-through service.

A step further, and also the focus of this Engineering Report, is a WFS Data Broker that is able to merge similar feature types, so that it offers a set of unique feature types to its consumers. In practice, this means that a feature type can be backed by one or more WFS data sources, as illustrated in Figure 3: High-level architecture of the WFS Data Broker (2).

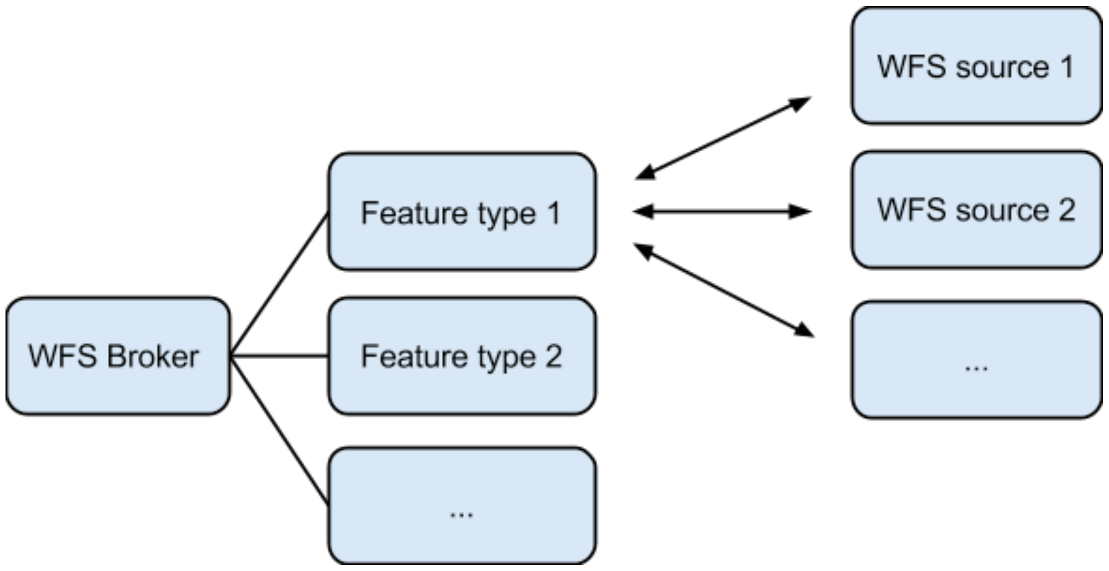

Figure 3: High-level architecture of the WFS Data Broker (2)

One of the tasks of the Broker is to determine the set of offered feature types to be able to populate its capabilities. Figure 4: Capabilities generation in the WFS Data Broker shows a high-level flow of actions to perform this task.

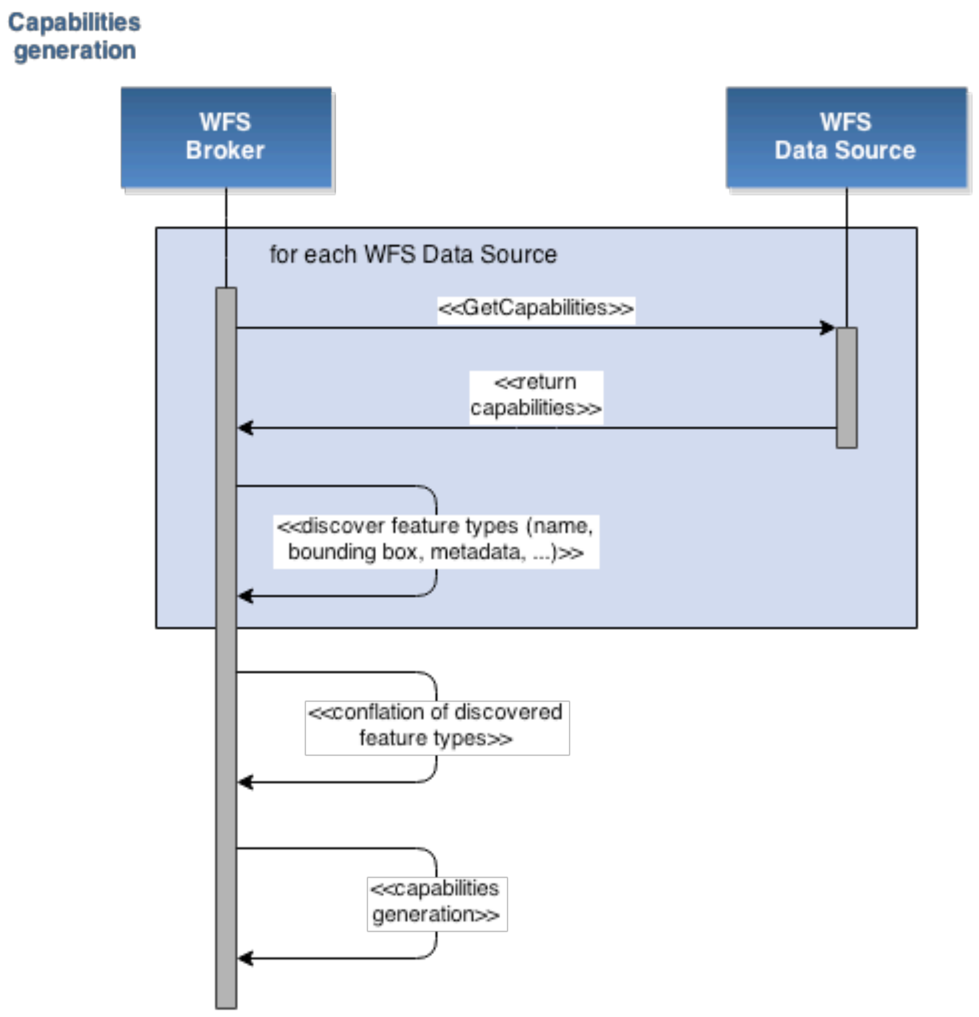
Capabilities generation
WFS Broker
WFS Data Source
for each WFS Data Source
<<GetCapabilities>>
<<return capabilities>>
<<discover feature types (name, bounding box, metadata, ...)>>
<<conflation of discovered feature types>>
<<capabilities generation>>

Figure 4: Capabilities generation in the WFS Data Broker

For each selected WFS data source, the Broker needs to determine the available feature types by means of a GetCapabilities request. Next, the complete list of feature types of all WFS data sources needs to be conflated to end up with a set of uniquely identifiable feature types, each aggregating data from one or more WFS data sources. Finally, the resulting list is used to populate the capabilities of the Data Broker.

This task can be performed during the start-up phase of the Broker. To take into account potential updates in the used WFS data sources, the Broker should regularly perform this task, either using a fixed update rate or on-demand, upon receipt of a GetCapabilities request. If a WFS data source supports the UpdateSequence property in its capabilities, it can / should be used by the Broker to quickly check whether anything has been updated.

Another main task of the Broker is the ability to respond to feature requests. Figure 5: Feature querying in the WFS Data Broker shows a high-level flow of actions to perform this task. For an incoming GetFeature request, the Broker needs to determine the overlapping WFS data sources for the requested feature type. If the request contains a spatial filter, it can be used to determine the overlapping data sources, since a feature type offered by a WFS data source by definition has an associated area of interest. If no spatial filter is available in the request, all WFS data sources linked to the feature type should be taken to account. The next step is to query the WFS data sources. Depending on the filtering capabilities supported by a data source, the Broker can decide to send a simple query (e.g., only consisting of a bounding box filter) and perform the filtering itself; if the data source supports the used filtering capabilities, the Broker can just forward the query. When all data sources are queried, the Broker should send a combined feature response to the client.

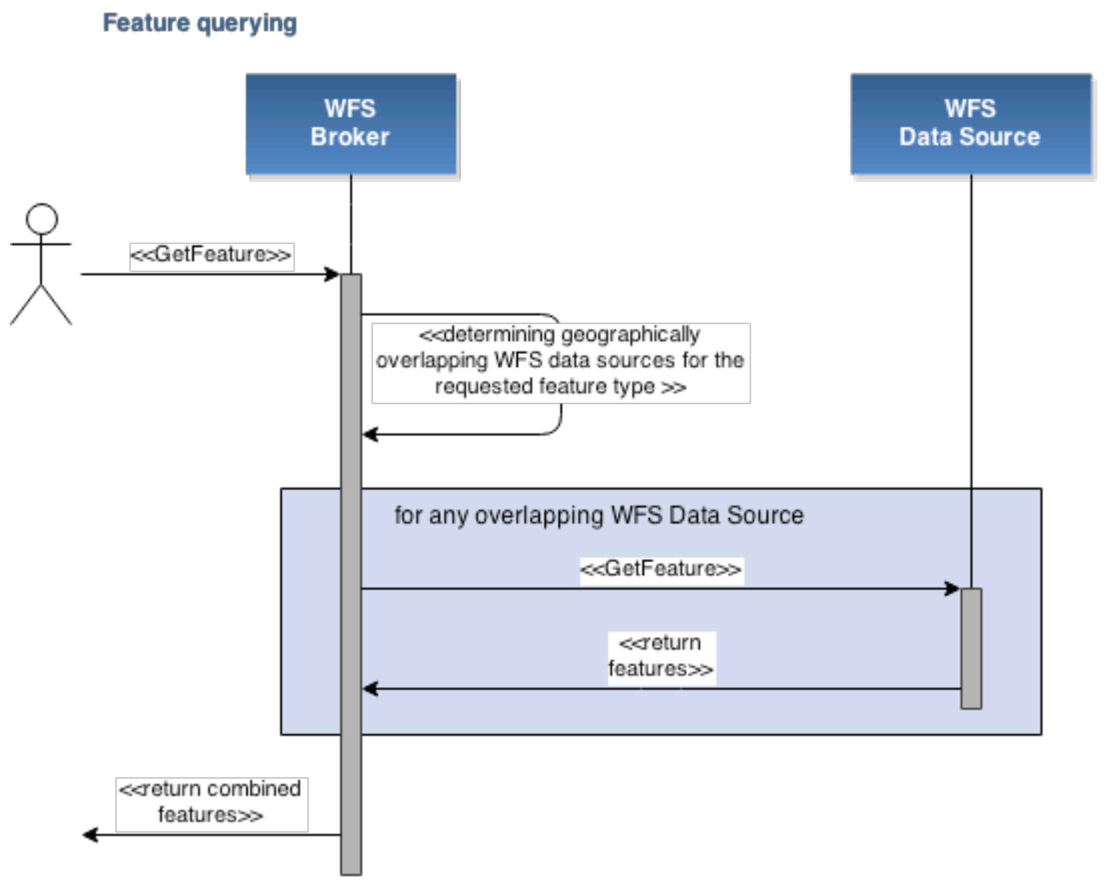
Feature querying
WFS Broker
WFS Data Source
<<GetFeature>>
<<determining geographically overlapping WFS data sources for the requested feature type >>
for any overlapping WFS Data Source
<<GetFeature>>
<<return features>>
<<return combined features>>

Figure 5: Feature querying in the WFS Data Broker

5.3 Broker Component architecture & design

Based on the scope & requirements discussed in the previous section, we can further look at how a Data Broker component can be designed and implemented.

During the initialization phase, the Data Broker needs to determine the set of feature types following the approach outlined in Figure 4: Capabilities generation in the WFS

Data Broker in the previous section. When up & running, we can distinguish two groups of tasks when a feature request is received:

- Querying data from the underlying WFS data sources
- Processing the resulting data and sending it to the client

The querying task needs to start from the aggregated feature type, find out the overlapping individual feature types and query the corresponding WFS data sources. The processing task needs to loop over these features, perform the broker tasks (conflation, provenance / lineage) and send the response.

Figure 6: Component architecture of the Data Broker shows the suggested component architecture to implement these tasks.

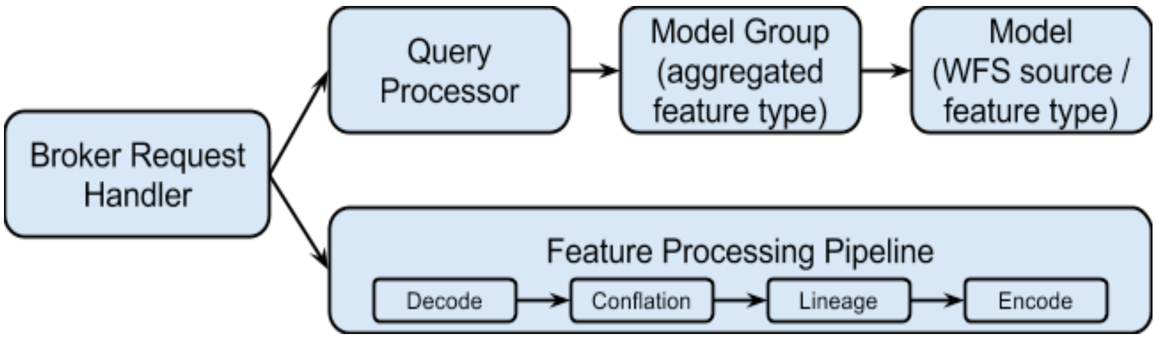
Broker Request Handler
Query Processor
Model Group (aggregated feature type)
Model (WFS source / feature type)
Feature Processing Pipeline
Decode
Conflation
Lineage
Encode

Figure 6: Component architecture of the Data Broker

Each WFS source / feature type combination is abstracted behind a model component, providing an interface to perform queries to the underlying data source. These model components are created during startup, based on the configured WFS data sources and feature types. The resulting models are then grouped in a model group, which is associated with one feature type. This includes a feature type conflation step to merge similar individual feature types to one coherent feature type. At the model and model group levels, caching can be applied if desired. Upfront, a query processor takes care of selecting the right feature types and sending the queries to the underlying WFS data sources.

The resulting responses are then streamed to a Feature Processing Pipeline. In this pipeline, features will be decoded, processed and encoded on-the-fly, feature by feature. This avoids the need to decode a complete response stream upfront.

5.4 Implementation

To support the research regarding the Data Broker concept, Luciad developed an implementation based on the design discussed in the previous section. This implementation is built on top of Luciad’s COTS software product LuciadLightspeed. LuciadLightspeed offers a set of standards-based software components, including an

OGC Web Services Suite equipped with an OGC-compliant WFS service component. One of this component's benefits for the Data Broker task is its open data back-end API, allowing users to easily connect to any type of storage component – such as other OGC web services.

5.4.1 Functional overview

The implemented Data Broker has the following functionality:

- OGC-compliant WFS 1.1.0 & 2.0.0 service interface with support for the following requests: GetCapabilities, DescribeFeatureType and GetFeature. Supported request encodings are HTTP GET and POST.
- Support for connecting with OGC WFS server components with support for version 1.1.0 or 2.0.0 and with support for AIXM 5.1 output.
- Support for OGC Filter 1.1.0 and 2.0.0.
- Support for various automatic & on-the-fly feature type operations useful for the Data Broker:
    - Aggregation of similar feature types from different OGC WFS data sources into one feature type.
    - Conflation of similar features served by different OGC WFS data sources.
    - Adding of provenance by integrating lineage information on the feature type level in the capabilities and on the feature level by adding ISO 19115-based lineage metadata to AIXM 5 features.
- Support for caching to improve the Data Broker's response time.

5.4.2 Deployment characteristics

The Luciad Data Broker implementation is based on Java Servlet technology. To run, the Broker requires a Java servlet container or application server compatible with Java Servlet 2.5 or higher. Apache Tomcat 7 was used by Luciad during Testbed 11. Other than being capable of running a Java Virtual Machine 1.7 (or higher) and an appropriate servlet container / application server, no requirements are posed on the underlying hardware or operating system.

5.5 Usage of OGC standards

The developed Data Broker concept relies on the OGC WFS 2.0 standard. Optionally, an OGC CSW can also be used to set up the data sources for the Data Broker, as will be further discussed in Section 6.5.

5.6 Suggested improvements

This section discussed a number of suggested improvements related to the OGC standards that have been used for the Data Broker.

OGC WFS 2.0:

- *Identification of the update sequence of an individual feature type.* The OGC WFS 2.0 defines an optional property ‘UpdateSequence’ that can be used in the WFS capabilities to identify the update status or revision version of a WFS. This property is a global property, applicable to the entire content served by the WFS. To ease the detection of updates to individual feature types - which is useful in case the Data Broker applies caching - it would interesting to have this property also available on a feature type level in the capabilities.

6 Analysis

Because of the fact that the Data Broker provides information coming from other services, several aspects impact the Broker’s operational behavior. When aggregating data in a service, the consumer needs to be informed about the origin of the data and any possible operations that the data might have undergone during the aggregation process. This aspect is further discussed in Section 6.1. Related to data aggregation, it might be that a feature exists in two data sources, resulting in the need for data conflation in the Data Broker. This is further discussed in Section 6.2. To improve performance, a Data Broker might apply caching techniques to reduce data transfers from its data sources; this is further discussed in Section 6.3. Section 6.4 further investigates scalability, and how to ensure that a Data Broker is capable of working smoothly in a sequential / aggregating setup. Finally, Section 6.5 investigates approaches to flexibly manage the data sources used by a Data Broker.

6.1 Provenance and Lineage

Provenance and lineage is an important aspect when it comes to aggregated data. Lineage not only provides a great deal of information on where the data originated from, it also allows you to estimate the quality and reliability of the data. ISO 19115 describes a model for metadata that features lineage information. ISO 19139 describes the XML encoding for ISO 19115.

The ISO 19115 model describes lineage in its DataQuality element. Lineage is composed of one or more sources, as well as one more process steps. The sources describe the origin of the data, while the process steps describe what processing has been done to the data to get the output result.

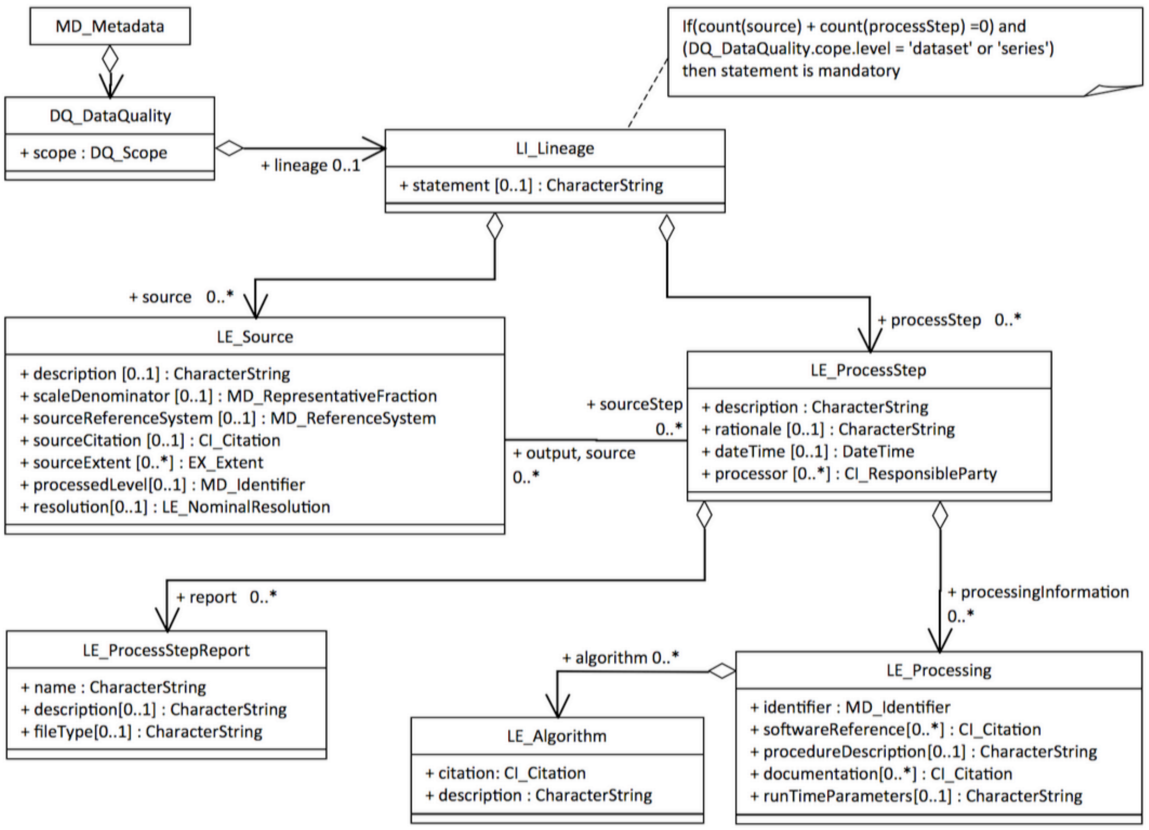
MD_Metadata
DQ_DataQuality
+ scope : DQ_Scope
+ lineage 0..1
LI_Lineage
+ statement [0..1] : CharacterString
If(count(source) + count(processStep) =0) and (DQ_DataQuality.cope.level = 'dataset' or 'series') then statement is mandatory
+ source 0..*
LE_Source
+ description [0..1] : CharacterString
+ scaleDenominator [0..1] : MD_RepresentativeFraction
+ sourceReferenceSystem [0..1] : MD_ReferenceSystem
+ sourceCitation [0..1] : CI_Citation
+ sourceExtent [0..*] : EX_Extent
+ processedLevel[0..1] : MD_Identifier
+ resolution[0..1] : LE_NominalResolution
+ processStep 0..*
+ sourceStep 0..*
+ output, source 0..*
LE_ProcessStep
+ description : CharacterString
+ rationale [0..1] : CharacterString
+ dateTime [0..1] : DateTime
+ processor [0..*] : CI_ResponsibleParty
+ report 0..*
LE_ProcessStepReport
+ name : CharacterString
+ description[0..1] : CharacterString
+ fileType[0..1] : CharacterString
+ processingInformation 0..*
+ algorithm 0..*
LE_Algorithm
+ citation: CI_Citation
+ description : CharacterString
LE_Processing
+ identifier : MD_Identifier
+ softwareReference[0..*] : CI_Citation
+ procedureDescription[0..1] : CharacterString
+ documentation[0..*] : CI_Citation
+ runTimeParameters[0..1] : CharacterString

Figure 7: ISO 19115 Lineage UML model

Since a lineage element in ISO 19115 can have multiple steps, it is possible to chain multiple data-brokers after one-another. This is useful in hierarchical setups.

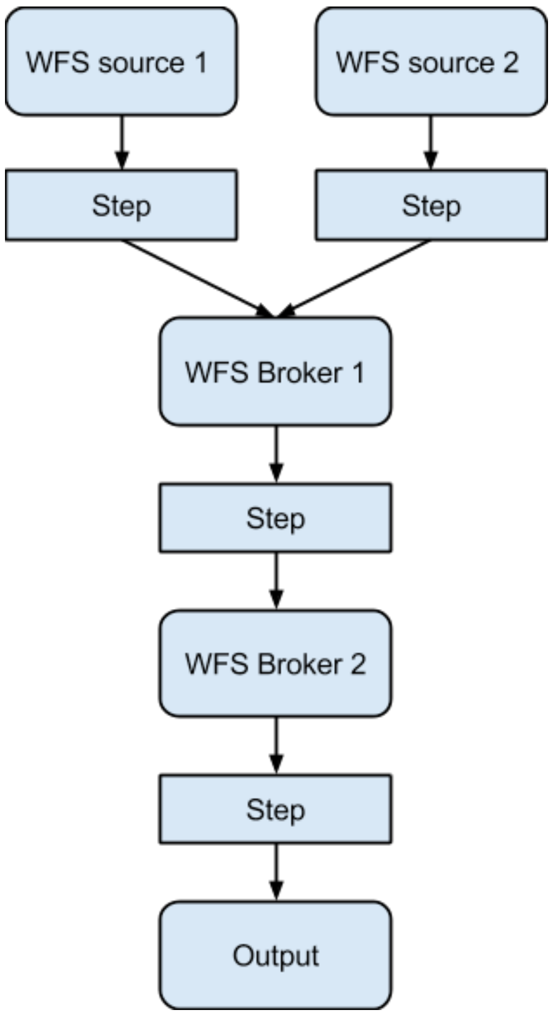

Figure 8: ISO 19115 modeling example for a chained Data Broker setup

An example of XML-encoded metadata lineage for a single source can be seen here:

```
<gmd:MD_Metadata>
   <gmd:dataQualityInfo>
    <gmd:DQ_DataQuality>
     <gmd:lineage>
      <gmd:LI_Lineage>
       <gmd:source>
        <gmd:LI_Source>
         <gmd:description>
          <gco:CharacterString>https://demo-wfs-server/wfs?</gco:CharacterString>
         </gmd:description>
         <gmd:sourceCitation>
          <gmd:CI_Citation>
           <gmd:title>
            <gco:CharacterString>Demo Server WFS Source</gco:CharacterString>
           </gmd:title>
          </gmd:CI_Citation>
         </gmd:sourceCitation>
```

```
                <gmd:sourceStep>
                  <gmd:LI_ProcessStep>
                    <gmd:description>
                      <gco:CharacterString>Aggregated data. No modifications were
made.</gco:CharacterString>
                    </gmd:description>
                    <gmd:dateTime>
                      <gco:DateTime>2015-03-24T09:14:54.277+01:00</gco:DateTime>
                    </gmd:dateTime>
                    <gmd:processor>
                      <gmd:CI_ResponsibleParty>
                        <gmd:organisationName>
                          <gco:CharacterString>Luciad</gco:CharacterString>
                        </gmd:organisationName>
                        <gmd:contactInfo>
                          …………………
                         </gmd:contactInfo>
                      </gmd:CI_ResponsibleParty>
                    </gmd:processor>
                  </gmd:LI_ProcessStep>
                </gmd:sourceStep>
              </gmd:LI_Source>
            </gmd:source>
          </gmd:LI_Lineage>
        </gmd:lineage>
      </gmd:DQ_DataQuality>
    </gmd:dataQualityInfo>
</gmd:MD_Metadata
```

Example lineage XML encoding

When looking at the Data Broker, there are two separate ways to add lineage metadata to the service:

1. Add lineage metadata on a WFS capabilities level
2. Add lineage metadata on a feature level

Both methods have advantages and disadvantages.

6.1.1 Integration at WFS Capabilities Level

On a capabilities level, we have the option of adding ISO 19115 metadata to a feature type, using the MetadataURL field of the FeatureType. This field allows us to link to one or more metadata elements that describe the lineage set for a FeatureType.

While this method is quite fast, and has a low bandwidth requirement, we found that this method is not optimal for the case where the desire is to merge multiple feature layers into a single feature layer. For instance, if there are two WFS sources, each serving AirportHeliport features, it is desirable to configure the Data Broker to offer a single feature layer for AirportHeliport features, instead of two feature layers (i.e., one for each source).

As soon as you merge these feature layers, metadata on a feature layer can no longer provide you with the information needed to see where each feature originates from.

Advantages:

- Metadata available without a GetFeature request
- Low metadata redundancy

Disadvantages:

- No fine-grained lineage
- Does not allow Data Broker to merge feature layers from different sources

6.1.2 Integration at Feature Level

For this report we looked primarily at AIXM 5.1 data. The AIXM 5.1 schema has a “featureMetadata” element for all features. This allows us to encode lineage metadata on a per-feature basis. This allows us to merge WFS feature layers, and group them per feature type, without losing track of which feature belongs to which source.

One important aspect for the performance of a feature-level metadata enrichment is the support of the Data Broker for streaming enrichment. If the Data Broker were to delegate its requests, and process them after the source has fully return a response, the client requesting the data could potentially time-out.

Instead, we recommend enriching while streaming, on a feature per feature basis. As soon as the source WFS service returns a feature, enrich the metadata as needed, and output the result directly to the response of the client. This has the added benefit that it reduces the memory footprint of the Data Broker.

Advantages:

- More fine-grained control over where feature data originates from.

Disadvantage:

- Higher bandwidth requirements
  More processing required in Data Broker GetFeature request

6.1.3 Recommendation

Our general recommendation for data lineage is to encode lineage at a feature level. When many data originators are merged into a single WFS Feature type layer (e.g. Figure 3), there quickly arises the need to be able to distinguish data at a feature level. As long as the Data Broker streams data feature per feature, the observed performance overhead should be relatively small, as demonstrated in section 6.4.

6.2 Conflation

Conflation is the act of merging multiple data sources into a single coherent data source. There are multiple aspects to data conflation, and this section will look over a few of them, and how they affect the design of the Data Broker.

First, conflation affects the data broker on a WFS capabilities level: How are feature layers from WFS services merged into a single WFS service? Secondly, conflation affects the data broker on a feature-level: WFS services with overlapping regions could have duplicate data. Lastly, conflation also affects the caching mechanism mentioned in Section 6.3: If we cache locally in the Data Broker, how do we avoid data duplication in the local cache?

6.2.1.1.1 Conflation at a WFS Capabilities level

We would like to clarify some terminology used in this section. In WFS terms, a "feature type" refers to an item in the capabilities document that logically groups together a set of features. This WFS feature type is identifiable with a unique name, given in the capabilities document of the WFS service.

In AIXM, a "feature type" refers to a specific type of feature. This could be for instance an aixm:RunwayElement, aixm:AirportHeliport, aixm:Airspace, etc… In both definitions, the term "feature" refers to an abstraction of real world phenomena.

Please note that the name of a WFS feature type does not have to match the AIXM feature types it serves. A WFS feature type also doesn't have to limit itself to a single AIXM feature type. Table 1 shows a few examples to demonstrate the flexibility of WFS feature types in combination with AIXM feature types.

Use case	**WFS Feature Type name**	**AIXM Feature Type**
1	aixm:AirportHeliport	aixm:AirportHeliport
2	AirportHeliportType	aixm:AirportHeliport
3	Random_name	aixm:AirportHeliport
4	Airports_and_airspaces	aixm:AirportHeliport, aixm:Airspace

Table 1: Examples of possible WFS feature type names and the type of AIXM features they return

At a capabilities level, the Data Broker needs to be capable of merging feature layers that logically belong in the same category into a single feature layer on the Data Broker. This means that layers serving the same AIXM Feature Type should be merged into a single WFS Feature Type at a capabilities level.

In our examination of several WFS sources serving AIXM 5.1 sources, we found that you can generally not rely on case 1 in Table 1. Often, WFS sources would use one of the first 3 use cases. Use case 4 was generally not seen: A single WFS Feature Type always seemed to map to a single type of AIXM Feature.

If the condition in use case 1 of Table 1 holds for all WFS sources, then the Data Broker can have a single initialization step that reads the capabilities documents of the source WFS services, and matches WFS Feature Type names to serve them as a single feature layer on the Data Broker. While this would ideal performance-wise, we found that since there is no coupling between WFS Feature Type name and AIXM Feature identifier, we cannot rely on this to work in the general case.

If we allow use case 1 to 3 on Table 1, then we can use an alternative to set up the Data Broker. If we know in advance that a single WFS Feature Type will always return a single type of AIXM feature, it is still possible to perform a GetFeature request that returns the first AIXM feature of the WFS Feature Type. This way we will be able to deduce the feature type of the WFS feature layer by looking at only a single feature type.

While this second method is slower as it requires more calls to the source WFS services, it is more robust in cases where slight variations in naming conventions of WFS Feature Types.

Our general recommendation for the Data Broker is to look at the first AIXM feature type returned by a GetFeature request to a WFS service. This should be done for each WFS Feature Type in the capabilities document, and should be done at startup of the service. The return AIXM Feature types should be stored in an in-memory map for each source WFS Service.

Then for each AIXM feature type found in the source WFS services, create a WFS Feature Type that serves that AIXM feature. For convenience, the name of this WFS feature type should be the name of the AIXM feature type. The BBOX should be a union of the BBOX elements of the source WFS services.

Table 2: Example input WFS Sources and their mapping in the Data BrokerTable 3: Resulting Data Broker Capabilities (Given Table 2 as input) show an example of this.

WFS Source	**WFS Feature Type name**	**WFS Feature Type BBOX**	**AIXM Feature Type**	**Resulting Data Broker WFS Feature Type name**

Source A	Airport	[0,0,20,20]	aixm:AirportHeliport	AirportHeliport
Source B	Heliport	[-10,-10,10,10]	aixm:AirportHeliport	AirportHeliport
Source B	Airspaces	[-15,-15,15,15]	aixm:Airspace	Airspace

Table 2: Example input WFS Sources and their mapping in the Data Broker

Data Broker Feature Type Name	**Data Broker Feature Type BBOX**	**AIXM Feature Type**
AirportHeliport	[-10,-10,20,20]	aixm:AirportHeliport
Airspace	[-15,-15,15,15]	aixm:Airspace

Table 3: Resulting Data Broker Capabilities (Given Table 2 as input)

6.2.2 Conflation at a WFS feature level

Conflation on a feature level involves certain assumptions of data described in [2]. The recommendations in this document suggest that every AIXM 5.1 feature has a universally unique identifier (UUID) that is generated by the primary data originator. This means that if overlapping regions of data are offered by several WFS services, the Data Broker can identify duplicate features using the identifier property of the AIXM feature.

For AIXM 5.1 data, we make the distinction between gml:identifier and gml:id. On a feature level, gml:identifier is used to store the UUID (universally unique identifier) of the AIXM feature. On a sub-feature level, each element of a feature contains a gml:id that locally identifies a particular subset of that feature. This gml:id is not globally unique, but it is locally unique to the service it resides in.

6.2.2.1 Feature level identifiers

For feature-level identifiers, we make the assumption that if two WFS sources have AIXM features with the same UUID that they refer to the same feature. If they have a different UUID, then either one of two cases is possible:

1. They refer to separate features
2. They refer to the same features, but contain different TimeSlices created by different sources.

For case 2, the following is described in **Error! Reference source not found.**:

*"It is therefore possible that two or more information sets (list of TimeSlices) exists for the same AIXM feature, in two different systems, with different gml:identifiers values. When data from different sources is merged in a single system, owner of that system might be confronted with the need to identify and merge duplicate data, based on actual properties of the feature, not on the gml:identifier".**Error! Reference source not found.***

The implications of this statement are that any two features in different source WFS services could potentially refer to the same conceptual idea (such as a Runway, or an Airspace), even if they don't share a common UUID. Detecting this use case is quite complex and it is our belief that it should be done as a pre-processing step, before setting up the Data Broker. For this reason, this use case is beyond the scope of this engineering report, and has been added to Section **Error! Reference source not found.** as future work. The rest of this document assumes that UUID's are able to uniquely distinguish features, even on a cross-server level.

Document **Error! Reference source not found.** also mentions the notion that servers might not be fully up-to-date at any time. If two WFS services contain duplicate data, it is possible that the two don't contain exactly the same set of TimeSlices. For instance, if one of the two acts as a "pseudo-primary" information source, then it could potentially be out-of-date.

For this case, we recommend that the Data Broker be pre-configured with WFS Services in such a manner that certain WFS services have priority over others. Giving primary data originators a higher priority than "pseudo-primary" originators lets the Data Broker easily choose which feature to serve in case of duplicate UUID's. The 'pseudo-primary' data source would be dropped in case of a conflict, under the assumption that the primary data source contains more up-to-date data.

Alternatively to this, it is possible perform a property comparison for each conflicting feature. In this method, the Data Broker compares individual TimeSlices on a property-level to find duplicate TimeSlices. After all duplicates are detected, it should be possible to create a single coherent feature with a unique identifier, served by the Data Broker.

The latter method is prone to problems where faulty data has been sent to one of the data sources. For this reason, we recommend using the former option, as it is more consistent in handling problematic time-slices.

6.2.2.2 GML object level identifiers

Apart from UUID's found in AIXM Features, there are also local gml:id elements for every GML object in a feature. This element is mandatory, and should be locally unique to the server. An example of the use of gml:id can be seen in Figure 9.

```
<aixm:Airspace gml:id="uuid.a82b3fc9-4aa4-4e67-8def-aaea1ac595j">
  <gml:identifier
     codeSpace="urn:uuid:">a82b3fc9-4aa4-4e67-8def-
aaea1ac595j</gml:identifier>
  <aixm:timeSlice>
    <aixm:AirspaceTimeSlice gml:id="ID00001">
      <gml:validTime>
        <gml:TimePeriod gml:id="ID00002">
          <gml:beginPosition>2010-06-29T17:31:00</gml:beginPosition>
          <gml:endPosition>2010-06-29T19:00:00</gml:endPosition>
        </gml:TimePeriod>
      </gml:validTime>
      <aixm:interpretation>BASELINE</aixm:interpretation>
      <aixm:sequenceNumber>1</aixm:sequenceNumber>
      <aixm:type>D</aixm:type>
      <aixm:geometryComponent>
        <aixm:AirspaceGeometryComponent gml:id="ID00003">
          <aixm:theAirspaceVolume>
            <aixm:AirspaceVolume gml:id="ID00004">
              <aixm:upperLimit uom="FT">500</aixm:upperLimit>
              <aixm:upperLimitReference>MSL</aixm:upperLimitReference>
              <aixm:lowerLimit uom="FT">GND</aixm:lowerLimit>
              <aixm:lowerLimitReference>MSL</aixm:lowerLimitReference>
              <aixm:horizontalProjection>
              <aixm:Surface gml:id="ID00005">
                <gml:patches>
                  <gml:PolygonPatch>
                    <gml:exterior>
              ...

</aixm:Airspace>
```

Figure 9: Example of the use of local gml:id (Example taken from Error! Reference source not found.**)**

When multiple data sources are merged into one, it is possible that there is a conflict in local gml:id's between servers. This is especially the case if each server has a similar gml:id generation technique.

It is the responsibility of the Data Broker to ensure that while data is being merged from multiple data sources, every gml:id is still unique inside a single GetFeature request to the Data Broker. One relatively easy way to achieve this is by prefixing or suffixing the local gml:id's with a fixed unique string per source WFS service. This fixed string could be any string that is unique to the service, such as the URL to the service.

While none of the WFS services tested in testbed-11 had conflicting gml:id, it is still important to note that the transformation of gml:id tags is not as easy as might seem at first. Care must be taken to prevent XLinks to gml:id data objects to become invalid. Any active XLink pointing to a gml:id element must also be transformed to match the new gml:id.

6.2.3 Conflation at a WFS caching level

Similarly to conflation on a WFS feature type, the act of caching locally in the Data Broker also means that conflation has to happen inside the Data Broker itself. Section 6.3 describes a few caching strategies that involve breaking a single BBOX request into several smaller BBOX requests, to reduce bandwidth costs for requests to the same regions. Since features are not neatly packed into a fixed structure or grid, features can be part of several BBOX regions. In other words: One single feature can appear in multiple BBOX requests. Figure 10: A single feature overlapping multiple BBOX regions shows an example of this.

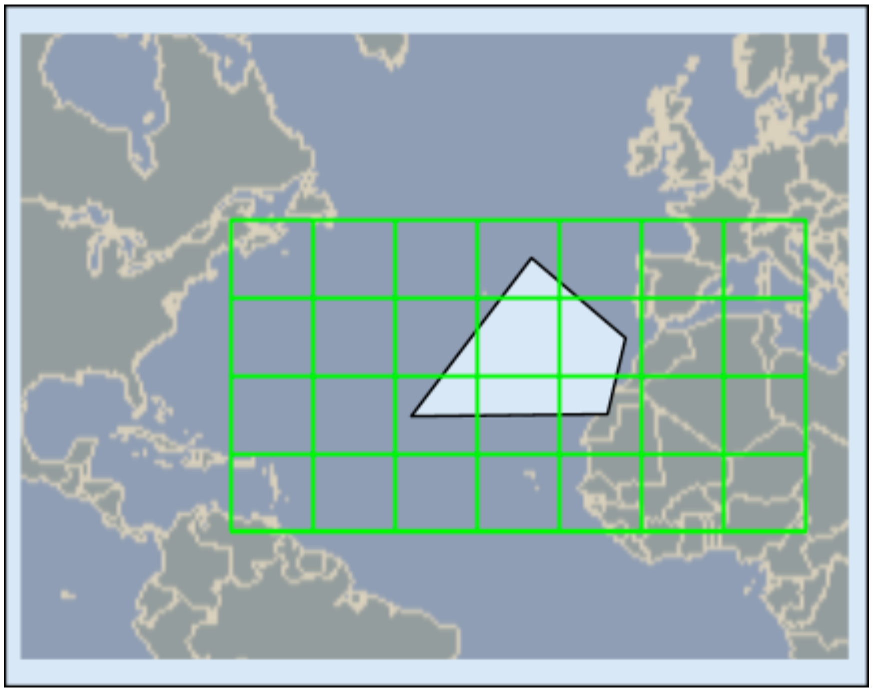

Figure 10: A single feature overlapping multiple BBOX regions

To alleviate this situation, we can define a set of maps and lists to keep track of data that could potentially be duplicated at the Data Broker cache level. We need a total 2 maps and a list to manage all the bookkeeping and reduce memory usage to a minimum, without sacrificing performance. The internal data structure to manage data duplication at a cache level is:

1. A map that maps BBOX tiles to a list of UUIDs
2. A map of UUIDs to BBOX tiles
3. A one-to-one mapping of UUIDS to their data.

See Figure 11: Example use of data structures to conflate local caching for a visual example of these structures.

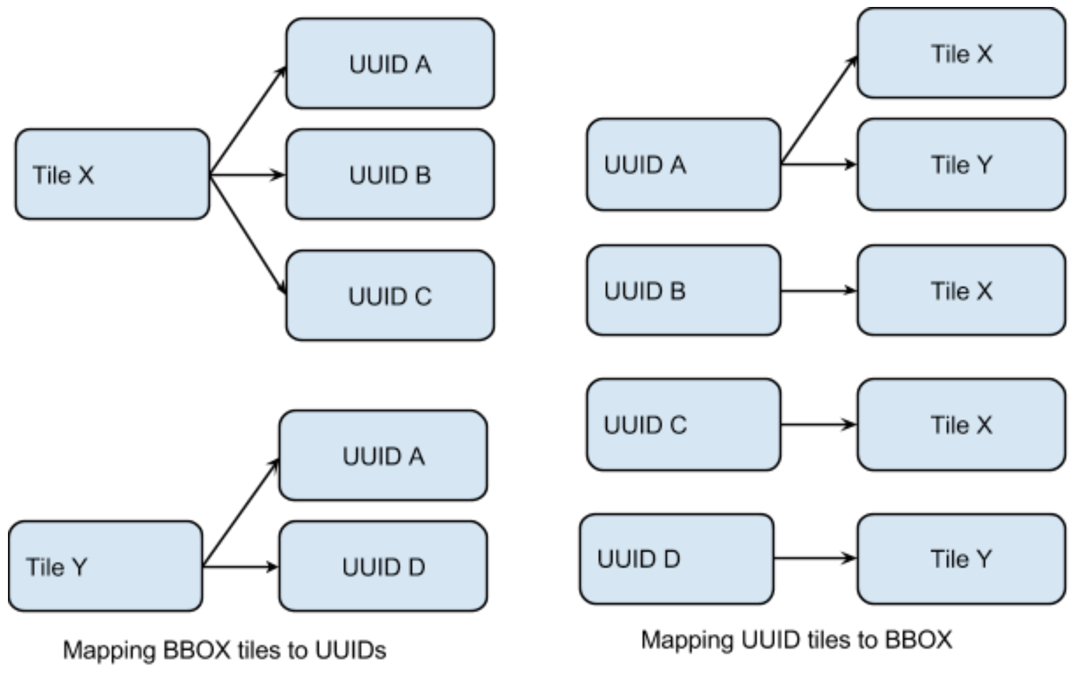

Figure 11: Example use of data structures to conflate local caching

The use of these data structures allows you to quickly and efficiently query what unique features are currently cached, and which BBOX regions they reside in. It also ensured the actual feature data is kept in a separate set, so geometric data is not duplicated.

When building up the cache, each map and set will need to be updated accordingly. If a UUID is already known, then the resulting feature data should not be decoded again. This results in increased performance for large features that overlap many BBOX regions.

When removing old cache entries to free up memory, care should be taken on how feature data is deleted. When a BBOX tile is removed from cache, all UUIDs it pointed to should also be updated. If a UUID for a feature is no longer available in any other tile, its feature data should also be removed from the cache.

One potential area where bandwidth could be reduced would be the generation of filters that exclude certain known UUID's from queries. For instance, if we have the feature data for UUID A, we can create queries to ensure that the source WFS exclude this UUID from the result-set. As mentioned in Section 1.3, we leave this experiment for future work.

6.3 Caching

To improve the performance of feature requests, the Data Broker can rely on caching techniques to reduce the amount of data that needs to be queried from its data sources. Multiple approaches can be used to define a caching strategy, depending on behavioral assumptions and cache control requirements. For the Data Broker, we define the following assumptions and requirements:

Assumptions:

- Features have a unique identifier, to allow the proper identification of cached feature entries.
- Feature queries involve at least a bounding box, to enable the use of a spatial-oriented cache.

Requirements:

- It should be possible to set a maximum size on the cache. This results in a need to be able to track the size of the cache and clear elements from it if the maximum size has been reached.
- It should be possible to detect outdated cache items and get new updates from the Data Broker's data sources.

The maximum size of a cache is a useful feature to limit the resources used by the Data Broker. It is necessary for the Data Broker to be able to estimate the memory it is using to represent cached entries. This is implementation specific, and will often depend on the underlying language used, as well as the way that features are modeled in the application.

For detection of outdated cache entries, there are multiple options available. These options generally require support by the source WFS servers.

- HTTP Conditional GET using etags
- HTTP Conditional GET using last-modified
- HTTP Cache Control Headers using expires, max-age or s-maxage

Regardless of what options are available at the server in terms of HTTP headers, it is vital that the server support spatial filtering on BBOX elements. The idea here is to subdivide queries into a grid of BBOX queries, where each bounding box is cached separately at the Data Broker.

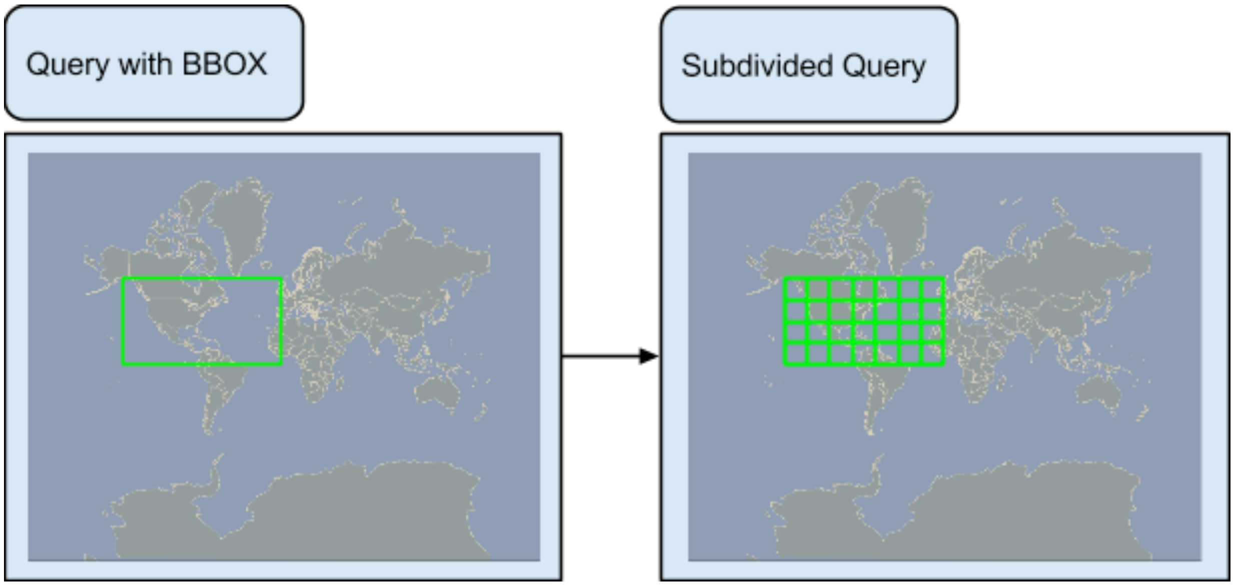

Figure 12: Example of a subdivided query

Figure 12: Example of a subdivided query gives an example of a subdivided query. If a client requests an arbitrary bounding box spatial filter from the data broker, the data broker should split up the request into multiple requests, where each request is fit onto its internal grid representation.

Splitting the problem into a uniform grid format simplifies caching at the broker. This approach also ensures that future requests around the same areas can reduce the amount of bandwidth used between source WFS and Data Broker. Only spatial areas in the uniform grid that are not cached need to be retrieved.

To ensure that the content of the queries gets updated when needed, we require that the source WFS service support some form of HTTP cache header control.

For the case of etags, we recommend that the etag generated by the WFS service takes into account not just the request, but also the spatial filter provided in the request. For other cache control headers, no special functionality needs to be instated.

If the source WFS service supports ResultType=hits, we can replace the uniform grid representation with a more balanced QuadTree structure in a relatively efficient way. A ResultType=hits GetFeature query, will return the amount of features for a given query, without returning any data.

A QuadTree is a data structure that starts off with a bounding box, and subdivides each cell in 4 pieces. This happens for each cell recursively, until a certain stop-condition is reached. The objective for caching is to have a balanced tree that has roughly an equal amount of features per cell. Thus the stop-condition is a certain number of features per cell. This threshold can be configured on the Data Broker.

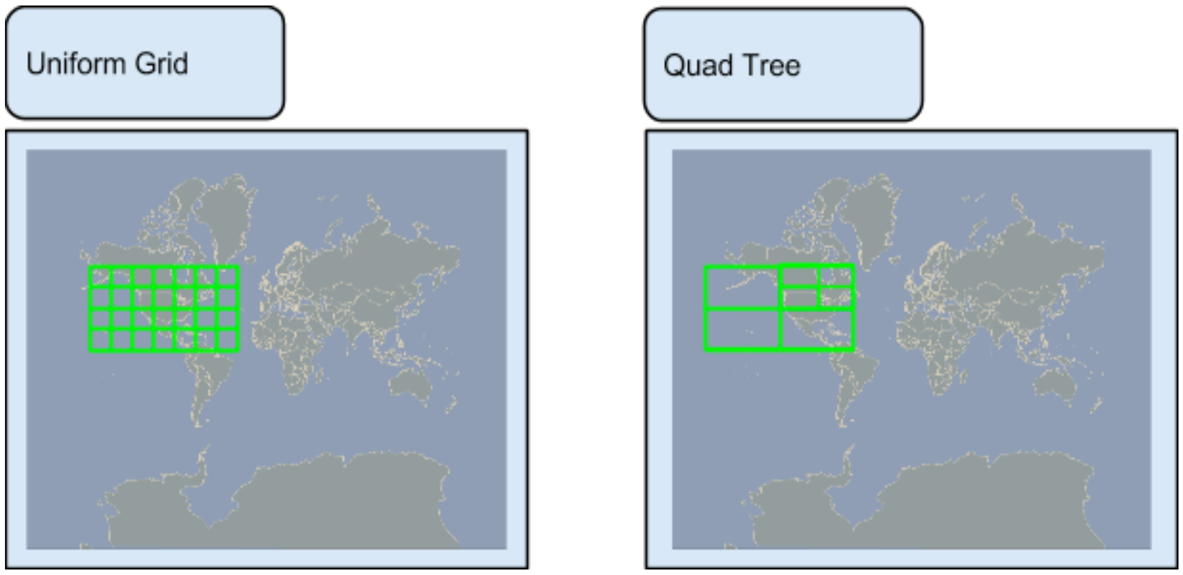

Figure 13: Uniform grids vs QuadTree

To create a QuadTree representation for a feature layer of a source WFS service, the Data Broker will have to perform an initial scan of the target dataset. To do this, we query the service with ResultType=hits find out how many features are present for a certain cell. If this number exceeds our threshold, we subdivide the cell into 4 sub-cells. If no more cells need to be subdivided, the QuadTree is complete.

The advantage of using a QuadTree over a uniform grid is that a QuadTree is more balanced in terms of memory usage per QuadTree cell. It is therefore also more balanced in terms of bandwidth usage per cell retrieval. A uniform grid can potentially be suboptimal for cases where a large dataset is focused on a single area, and there are a few outliers outside of this area. Using a QuadTree alleviates this situation, as it will contain more cells at locations where there is more data.

The downside of using a QuadTree is that it requires a separate setup step. This setup step does have a cost, which depends largely on the performance of the source WFS services. For Data Brokers that are rarely reset or reconfigured, this should not be a big problem.

Querying a WFS service using ResultType=hits should be fairly efficient. While it requires the server to do spatial intersection tests, the bandwidth used to set up this data structure is minimal. No feature data is transferred in the setup-phase. The basic assumption here is that this setup does not have to happen very often, as feature data does not often spatial move around often for aviation data. Most of the updates to data come from Digital NOTAMs, which are update to existing features.

Further optimizations could potentially be done by the use of WFS-TE. When a cached BBOX area is out-of-date, it could potentially be beneficial to only retrieve the latest changes of each of the cached features to save bandwidth. For instance, the baseline will not need to be retrieved again. However, since [4] was being revised at the time of writing, nu further investigation has been done on this topic, and it has been left as future work in Section 1.3.

6.4 Scalability

An important aspect when setting up a data distribution architecture involving one or more WFS Data Broker components is scalability. We can identify a number of factors that introduce a performance cost and thus impact the overall scalability; for each of these factors, we will discuss possible mitigation actions.

- **Network connection overhead**. Each broker essentially introduces an indirection, resulting in an additional network connection in the overall loop between a client and a data source. This additional network connection is inevitable by definition. If performance is a key requirement, make sure that the available network bandwidth in the broker is aligned with the available bandwidth in broker's data sources and the needs of the clients: the weakest link in the connection loop will determine the network throughput so this is an important factor with respect to overall scalability.
- **Provenance and lineage overhead.** Section 6.1 discussed the introduction of lineage metadata to features queried via a broker, to inform a client about the origin of the data. This also comes with a performance cost, since a data enrichment step is needed for this. To make this enrichment step as scalable as possible, it is important that the broker can process features in a streaming way without having to store data responses first. In case of XML data, on which the default WFS exchange format GML is based, this boils down to implementing a data processing pipeline that uses streaming XML parsing and writing. For most of the commonly used object-oriented programming languages have, XML APIs exist with streaming support. Luciad's Java-based WFS Data Broker uses the Streaming API for XML (StAX) for this. The implemented data processing pipeline will buffer the XML of a single feature, decode it into a Java domain model, perform the enrichment step and encode it back to XML.
- **Data aggregation overhead.** In a realistic setup, a WFS Data Broker will often need to contact more than one WFS data source to answer a feature type request. To optimize the throughput in the broker and overall response time, it is recommended to perform these requests in parallel. This also reduces the possible the impact of a slow data source.
- **Conflation overhead.** Relate to the aggregation of data, a WFS Data Broker will need to perform conflation in order to avoid duplicate features in its response. As outlined in Section 6.2, this requires feature to be uniquely identifiable. In case of AIXM data, this is achieved by using the gml:identifier property on the feature level, as recommended in [2]. Via an extra step in the data processing pipeline, this identifier can be used to perform the conflation step. To align this with a parallel execution of data source queries (as recommended in the previous bullet), the broker could maintain a temporary map for each incoming request, tracking all the identifiers of processed features, properly synchronized across all

execution threads. When an identifier is already present, the broker knows that the feature has been encountered and processed earlier.

To test the scalability of the WFS Data Broker in practice, a number of experiments have been performed using Luciad's broker implementation in combination with the AIXM WFS data sources provided in Testbed 11.

Experiment 1: analyzing the impact of adding one or more brokers between a client and WFS data source.

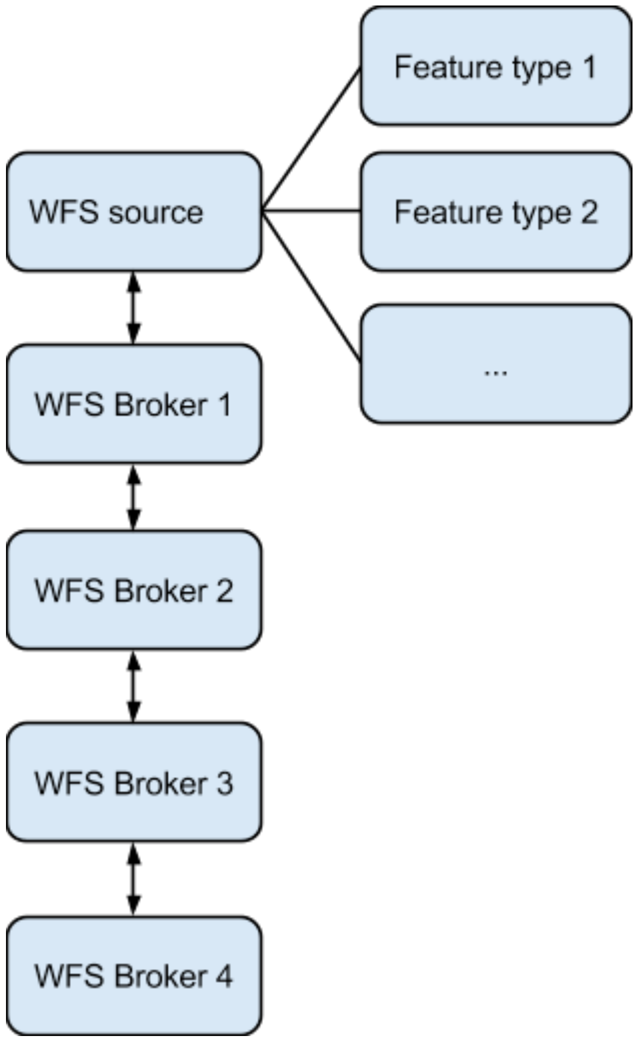

Figure 14: High-level architecture of the scalability experiment to detect the impact of adding brokers between a client and a data source

In this experiment, we identify one WFS source, offering a set of features, and 4 WFS Data Broker components that are sequentially connected with the WFS source. Figure 14: High-level architecture of the scalability experiment to detect the impact of adding brokers between a client and a data source shows the high-level architecture of this experiment.

The goal is to analyze the impact of sequentially adding brokers in a communication pipeline. To test this, we request 200 AirportHeliport features from each of the components and determine the response time. To focus only on the impact added by a broker, we normalize the results by subtracting the average response time of the WFS source, since this is a fixed cost not related to the broker. Table 4: Response times for brokers in a sequential setup and Figure 15: Response times for brokers in a sequential setup show the results of the test.

Component	**Reponse time (in ms)**
Source	0 (normalized)
Broker 1	255.5
Broker 2	556.5
Broker 3	934.7
Broker 4	1376.5

Table 4: Response times for brokers in a sequential setup

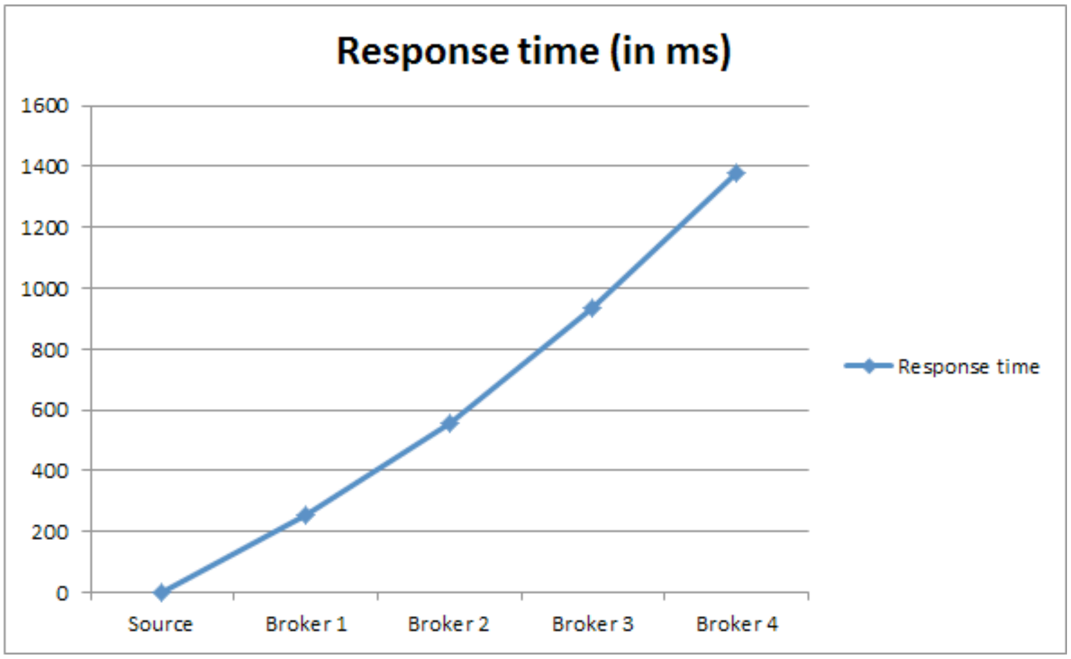

Figure 15: Response times for brokers in a sequential setup

By looking at the results, we can clearly see the cost that is introduced by each broker and the fact that this cost slightly increases for each additional broker. Table 5: Added cost for each broker in a sequential setup shows the cost introduced by each broker compared to the previous component.

Component	**Added cost (in ms)**
Broker 1	255.5
Broker 2	301
Broker 3	378.2
Broker 4	441.8

Table 5: Added cost for each broker in a sequential setup

The fact that this cost increases with each broker level is largely explained by the impact of the data processing pipeline, which enriches a feature with lineage information. This enrichment step adds about 50 lines of XML to each feature; when looking at the output of broker 4, this means that about 200 lines of XML are been added per feature, thus 200 x 200 = 40000 lines of XML in total for the request of 200 features. This obviously impacts the XML decoding and encoding operations in the brokers, and also the network throughput. However, the relative impact of a broker is still small compared to the average response time of the feature request when sent to a WFS data source; this was about 1.5 to 2 seconds for the Aviation WFS providers in Testbed 11.

Experiment 2: testing the impact of feature aggregation in a broker.

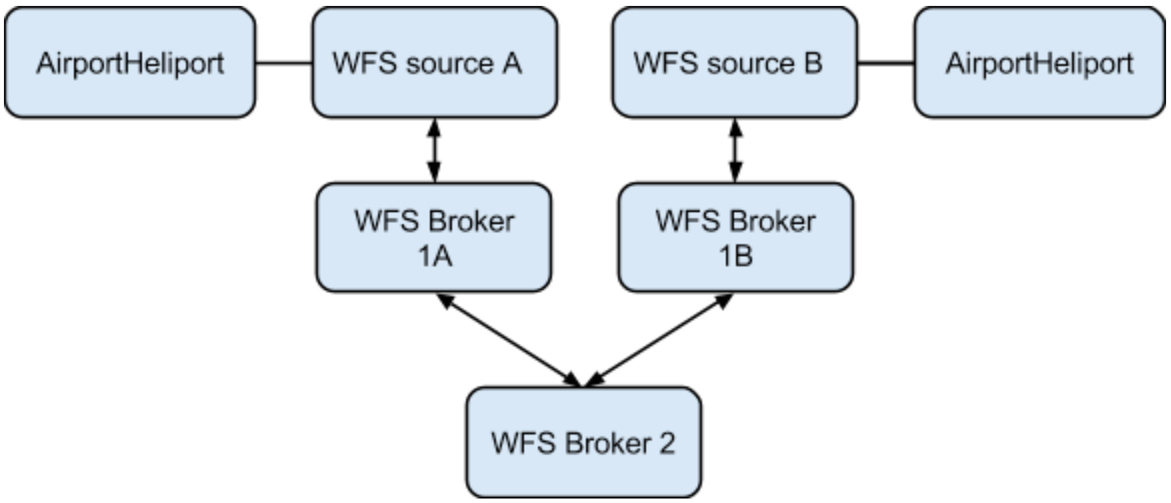

Figure 16: High-level architecture of the scalability experiment to detect the impact of feature aggregation in a broker

In this experiment, we identify two WFS sources, offering a feature type (AirportHeliport), and 3 WFS Data Broker components of which 2 redistribute the WFS sources and of which one aggregates the other 2. Figure 16: High-level architecture of the scalability experiment to detect the impact of feature aggregation in a broker shows the high-level architecture of this experiment.

The goal is to analyze the impact of brokers in case they need to aggregate data from various data sources. To test this, we request 200 AirportHeliport features from the source, 100 features from each of the two redistributing brokers and finally 200 features of the broker that aggregates the other 2 brokers. To focus only on the impact added by the brokers, we again normalize the results by subtracting the average response time of the WFS source. Table 6: Response times for brokers in a parallel / aggregating setup and Figure 17: Response times for brokers in a parallel / aggregating setup show the results of the test.

Component	**Reponse time (in ms)**
Source	0 (normalized)
Broker 1A	240
Broker 1B	240
Broker 2	665.5

Table 6: Response times for brokers in a parallel / aggregating setup

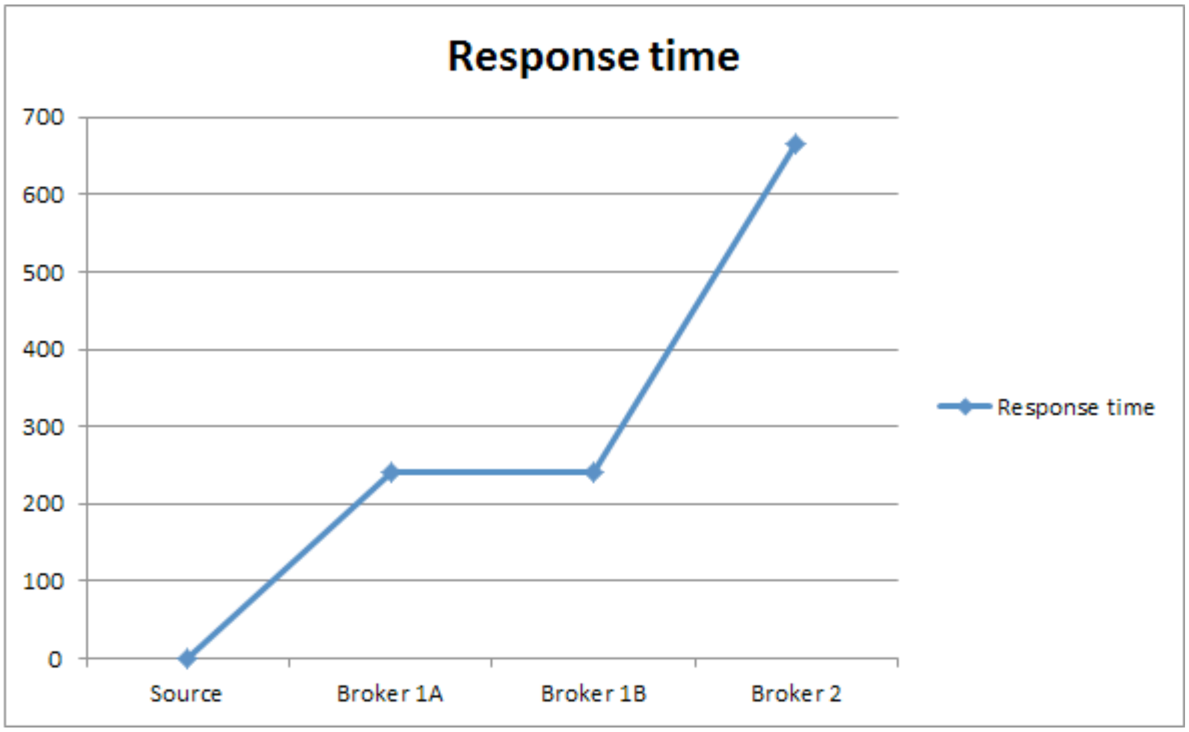

Figure 17: Response times for brokers in a parallel / aggregating setup

Table 7: Added cost for each broker in a parallel / aggregating setup shows the cost introduced by each broker compared to the previous component.

Component	**Added cost (in ms)**
Broker 1A	240
Broker 1B	240
Broker 2	425.5

Table 7: Added cost for each broker in a parallel / aggregating setup

By looking at the results, we can see a cost that is introduced by broker 1A and 1B that is slightly less than introduced by broker 1 in the previous experiment. This is explained by the fact that these queries only involved 100 features instead of 200 features. Next, the cost introduced by broker 2 is higher than introduced by broker 2 in the previous experiment. This is explained by the fact that broker 2 needs to consume data from 2 data sources (broker 1A and 1B) and that broker 2 needs to perform a conflation step in the data processing pipeline. Again, the impact is relatively small compared to requesting 100 features from two WFS sources, which in practice add up to 2.5 - 3 seconds for the Aviation WFS providers in Testbed 11.

6.5 Flexible management of sources

When setting up a WFS Data Broker, one or more WFS data sources need to be identified and configured. In a static architecture, it can be suitable to store a configuration file together with a broker that lists its data sources. In a dynamic architecture, where new data sources are introduced, or data brokers are restructured / reconfigured (e.g., with additional filters for data sources that select certain feature types, define a fixed area of interest, …), it can be more convenient to centralize this configuration and potentially offload it to a separate service.

From a technology point of view, primary candidate standards to implement this are ISO 19119 and OGC CSW:

- ISO 19119 is an international metadata standard for describing service metadata. Included is model to enable users to combine and chain services in ways that are not pre-defined by the service providers. This could serve as a basis to define the configuration of a data distribution architecture involving one or more WFS Data Broker components.
- OGC CSW is an international web service standard to publish and access digital catalogues of metadata for geospatial data, services and related resource information. It is fully aligned with the metadata standards ISO 19115 and ISO 19119, so it is an ideal candidate for a web service to manage the configuration of a WFS Data Broker architecture using ISO 19119.

This approach has not been tested in practice and is therefore listed as potential future work.

7 Conclusions & recommendations

This document researched the concept of a WFS Data Broker, enabling the possibility to cascade WFS components to form a data source chain. A main conclusion is that a WFS Data Broker can be set up in practice, fully aligned with the core OGC WFS standard. Deeper insight and practical recommendations are given to set up an OGC WFS-based data distribution architecture involving WFS Data Broker components

Recommendations:

- The Data Broker concept can be implemented using standard OGC WFS functionality. However, for an optimal setup, several additional recommendations & added features have been put forward in the Data Broker ER. Highlights include:
    - To optimize aggregation of similar AIXM feature types, it is recommended to let a WFS serve a single AIXM feature type (e.g.,

AirportHeliport) per WFS feature type and with as name the AIXM feature type name.

  - To enable proper conflation, it is recommended using a similar identification scheme for features across multiple WFS data sources. Applied to AIXM, this boils down to using a consistent UUID for the feature’s gml:identifier, making sure that the same airport served by the 2 WFS data sources have the same value.

  - To enable the detection of a changed WFS data source, it is recommended for WFS servers to publish an UpdateSequence value in the capabilities (optional in the WFS standard).

- It is important for the Data Broker implementation to stream data on a feature-by-feature basis, rather than on a query-by-query basis. This significantly increases time-to-response for the client, and it reduces overall memory requirements of the data broker. In combination with a feature processing pipeline, the Data Broker can implement all its responsibilities in a streaming way: conflation, caching and provenance/lineage enrichment.

- Since a single query can contain data from multiple sources, it is important to store metadata on a per-feature basis. If you don’t, you lose information on the origin (lineage) of specific features.

- Caching on Data Broker level should be achieved by splitting up requests into fixed BBOX regions called tiles or cells. Depending on source WFS capabilities, this can be done using either uniform grid structure, or a more complex QuadTree.

Annex A: Revision history

Date	Release	Editor	Primary clauses modified	Description
10/02/2015	0.1	Daniel	All	Created outline.
22/03/2015	0.2	Robin	5 & 6	Added introduction and overview. Added small section on caching and suggested improvements.
24/03/2015	0.3	Daniel	6.1 & 6.3	Improved section on lineage and caching.
24/03/2015	0.4	Robin	5	Added content to the Architecture & Design section.
24/04/2015	0.5	Robin	6.4, 6.5, 7, 8	Added section on scalability, flexible management of data sources, conclusions, future work and bibliography.
28/04/2015	0.6	Robin	6.4	Added scalability research experiments & results.

05/05/2015	0.7	Daniel	6.2	Added section on conflation.
08/05/2015	0.75	Robin	5	Improved Architecture & Design section.
11/05/2015	0.8	Daniel	6.2	Improved section on conflation.
12/05/2015	0.9	Daniel	1.4, 5.4, 6.1, 6.3, 7	Improved various sections.
12/05/2015	1.0 RC	Daniel, Robin	All	Final editing to prepare version 1.0's release candidate.
18/06/2015	1.0 RC2	Daniel, Robin	All	Revised based on feedback.

Bibliography

[1] OGC Web Feature Service 2.0 Interface Standard, OGC document 09-025r2

[2] http://www.aixm.aero/gallery/content/public/AIXM51/AIXM_Feature_Identification_and_Reference-1.0.pdf, AIXM 5 Feature Identification and Reference

[3] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html, HTTP/1.1 header field definitions (includes caching-related headers)

[4] http://external.opengis.org/twiki_public/AviationDWG/ProposalWFSTemporalityExtension, WFS-TE (Temporal Extension) Discussion Paper (OGC 12-027r3)

[5] http://external.opengeospatial.org/twiki_public/pub/AviationDWG/GMLGuidelinesForAIXM/12-028_Use_of_GML_for_aviation_data_-_Discussion_Paper_07.doc, Use of GML for aviation data (OGC 12-028)

[6] ICAO Annex 15(13th Edition), Aeronautical Information Services