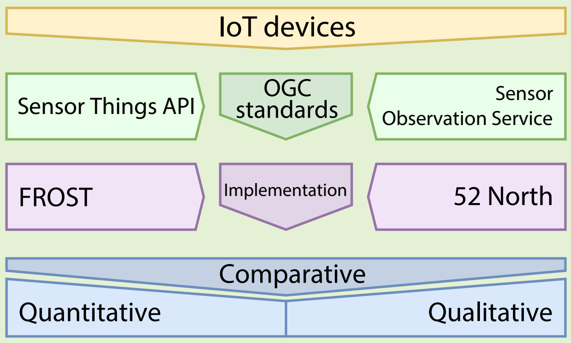
A comparative study in the standardisation of IoT devices using geospatial web standards

Daniel Marsh-Hunn, Sergio Trilles, Alberto González-Pérez, Joaquín Torres-Sospedra and Francisco Ramos

***Abstract*— Although billions of devices are embedded in the World Wide Web through the Internet of Things, there is still a lack of a common, interoperable way to connect them and make them interact seamlessly. IoT has also found its way into the spatial web. Environmental monitoring and sensing platforms connected over the web by wireless sensor networks are now a common way to monitor natural phenomena. This study compares two open Web Standards (*OGC's Sensor Observation Service* and *SensorThings API*) from the geospatial point of view. An IoT platform, called *SEnviro*, is used to integrate and evaluate implementations for each standard and contrast their qualitative and quantitative traits. The results of the study show that the *SensorThings API* proves to be the adequate Web Standard for IoT applications in terms of interoperability. It outperforms the contesting Web Standard in terms of flexibility and scalability, which strongly impacts on developer and user experience.**



***Index Terms*— Internet of Things, interoperability, geospatial standards, sensors**

I. Introduction

The Internet revolution enabled large-scale interconnection between people across the globe. Today, technological advance allows objects to interact over the Internet without the aid of human intervention, creating an Internet of Things (IoT). This concept first emerged in 1999 and has since been subject to constant evolution, redefinition, and expansion. It stepped out of its infancy and is transforming the current Internet into the fully integrated future Internet, connecting billions independent and intelligent devices [1].

Rapidly developing device-to-cloud technologies and the increasing deployment of devices connected to the Internet bring along a new dimension of possibilities and applications in various fields of human activities, but also imply new challenges in making different solutions and heterogeneous data interact seamlessly, enabling a large-scale IoT [2]. Widely defined as ”*a worldwide network of interconnected uniquely addressable objects, based on standard communication protocols*” [3], predictions estimate the IoT will consist of 21 billion connected devices exchanging information over the Internet by 2025 with an economic impact of 1.6 trillion US$ [4]. IoT applications are being developed in significant sectors such as smart business, inventory management, smart home, transportation and logistics, health-care, security and surveillance, and environmental monitoring. This vast number of ”things” can access and acquire data about devices and their environment, independent of human interaction [5].

Environmental and earth monitoring IoT applications have received increased attention in recent decades since they have become a key factor of sustainable growth worldwide. Observing natural phenomena in the field can be challenging due to harsh climatic conditions and difficulty of physical access, resulting in high costs for sensor deployment and maintenance [5]. These challenges have been addressed by technological advances in low power integrated circuits and wireless communications. Modern sensing devices have drastically decreased in size, cost, and power consumption, resulting in the viability of deploying intelligent sensors networks. These Wireless Sensor Networks (WSN) may consist of a large number of nodes with limited processing capability and storage. They can be equipped with several different sensors, capable of observing multiple natural phenomena [1].

Modern web technologies are advancing at a high rate and demand for research in specialised fields is increasing to stay up to date with state-of-the-art technology. IoT solutions are

Manuscript received month day, year; revised month day, yea; accepted month day, yea1. Date of publication month day, yea; date of current version month day, yea. Sergio Trilles has been funded by the postdoctoral Juan de la Cierva fellowship programme of the Spanish Ministry for Science and Innovation (IJC2018-035017-I). The project is funded by the Universitat Jaume I - PINV 2017 (UJI-A2017-14).

D. Marsh-Hunn is with geOps spatial web (e-mail: daniel.marsh-hunn@geops.de)
S. Trilles, A. González-Pérez, and F. Ramos, Universitat Jaume I, Castellón, Spain (e-mail: {strilles,algonzal,jromero}@uji.es)
J. Torres-Sospedra is with UBIK Geospatial Solutions S.L., Castellón, Spain (e-mail: torres@ubikgs.com)

Digital Object Identifier 10.1109/JIOT.2020.XXXXXXX

no exception; researchers are developing new implementations to enhance the quality and efficiency of this kind of systems. A general challenge, in the IoT domain, as well as in specific areas, is to achieve full system interoperability. Globally defined as "*The ability of two or more systems or components to exchange information and to use the information that has been exchanged.*" [6], this is a key concept to make environmental monitoring information accessible to a broader community, and it is a necessary step to take towards open data. Since IoT emerged, a large variety of vendors, researchers, and interested parties have developed IoT solutions in parallel. Although several device types and protocols for IoT are available on the market, only a few interoperate among each other [7]. The importance of interoperability is becoming more evident as the number of IoT solutions grows. While the worldwide Internet relies on standard technologies and protocols like HyperText Transfer Protocol (HTTP), Secure SHell (SSH), etc., these solutions are not well-suited for devices with severe power and data loss constraints. Efforts are currently being made to produce standards compatible with IoT environments [2].

Although the concept of IoT is already established and implementations are mushrooming manifold, it still lacks a widely accepted standard model to enable broad-scale interoperability. Standardisation bodies and alliances are working on defining web standards and protocols, and their adoption requires user and developer consensus through trials and testing. Several interoperable standards are already available, which have minor functionality, specialisation, and structure difference. With IoT also finding its way into environmental monitoring, it is crucial to investigate the potential of existing Web Standards in an interoperability context within this domain. An important actor in interoperability research in the geospatial field is the Open Geospatial Consortium (OGC). This institution provides some initiatives to standardise the IoT environment [8, 9].

This work aims at investigating open web standard solutions for IoT applications. The main contribution of this study lies in producing an in-depth comparison between a selection of web standard solutions from the geospatial domain. The options analysed support geospatial functionalities, useful for data analysis and visualisation [10], and are considered standards recognised by the OGC community. They are compared in terms of performance, semantics, flexibility, and scalability levels. The comparison is structured in terms of qualitative and quantitative aspects. The former analyses how to work with both standards from the point of view of the regulated specifications. For the latter an implementation for each standard was chosen and deployed for comparison in terms of performance. Different computational cost evaluations were performed, contrasting memory usage and time cost. A previous environmental monitoring IoT platform, called *SEnviro for Agriculture*, has been used to deploy these standards [11, 12]. A broker pattern using standard adapters are used to integrate each standard. A further goal of the study is to enhance the interoperability of the already established *SEnviro* platform.

The remainder of this paper is structured as follows. Section II contains the relevant topics to this paper. Section III includes information about the area and context of experimentation used in *SEnviro for Agriculture* [11] project. Section IV sheds light on the methodology used to evaluate the potential of applying open Web Standards. Sections V and VI present the qualitative and quantitative evaluation results of the comparing methods. Next, Section VII addresses a discussion of the results obtained in the previous sections. The paper culminates in Section VIII with conclusions and recommendations for future work.

II. Background

The number of embedded devices within the IoT is increasing drastically, and the IoT producers develop web service protocols only supported by their proprietary IoT devices. It results in closed vertical silos of IoT, each having its complete IoT frameworks, including devices, gateways, services, and applications. An upcoming issue in IoT is that elements in different silos cannot connect, leading to scattered IoT solutions with incompatible, co-existing protocols [8, 13].

An important actor in interoperability research in the geospatial field is the Open Geospatial Consortium (OGC), an organisation of over 260 members from the academic and industrial sectors, as well as governmental agencies. Their primary goal is to find participatory consensus for openly available interfaces and encodings for the Geospatial Web. The OGC provides a set of Geospatial Abstract Specifications for different types of geographical data, upon which the OGC's interoperability standards build on [14].

A. Open & Sensor Web Standards

Geo-scientists have been uploading geographical data for sharing and exploration since the dawn of the World Wide Web (WWW). Machine-to-Machine (M2M) data harvesting has led to community-adopted frameworks, common standards, and enriched metadata, significantly improving observational accuracy, sensor discovery, and configurability [15].

As one of the main actors in the field of sensor web standards, the OGC's Sensor Web Enablement (SWE) builds on machine-readable encoding like Sensor Model Language (SensorML), Observation & Measurement (O&M) and Geography Markup Language (GML) to fully describe the processes used in producing observations and their corresponding sensors [15].

In the scope of the OGC SWE, the organisation released a set of services to facilitate the exchange of observations among SWE enabled nodes and to allow clients and servers to arrange, encode and transfer observations in a semantically enabled way [16]. Our contribution will consider the well-established Sensor Observation Service (SOS) [17] and the more recent SensorThings API [18]. It is important to note the age difference of almost a decade between the two standards. SensorThings API, being the more recent standard, is a lot more focused on IoT devices, since the IoT concept has grown profoundly in importance and popularity. SOS is a more broadly scaled solution for sensor systems in general, but can nonetheless be applied effectively to IoT networks.

1) Sensor Observation Service: The OGC approved SOS in 2007 as an official open standard for handling observations in the WWW. Based on the SWE standard framework, SOS provides a standardised interface using SOAP XML to manage and retrieve metadata and observations from heterogeneous sensor systems and is designed to match the O&M data model (see Figure 1) [17].

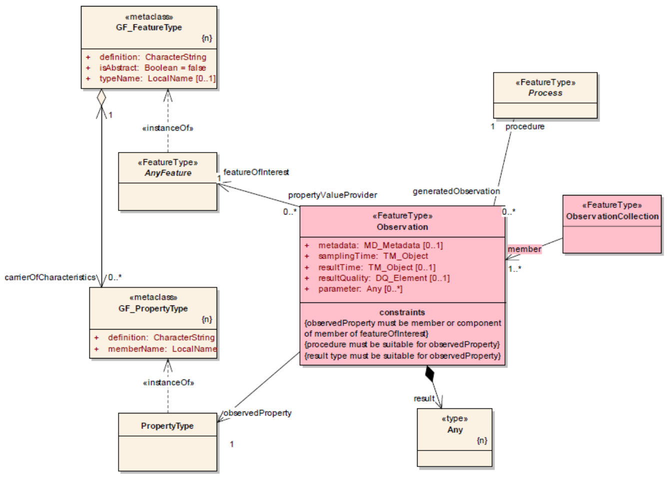

«metaclass»
GF_FeatureType
{n}
+ definition: CharacterString
+ isAbstract: Boolean = false
+ typeName: LocalName [0..1]
1
«instanceOf»
«FeatureType»
AnyFeature
featureOfInterest
1
propertyValueProvider
0..*
«FeatureType»
Process
1
procedure
generatedObservation
0..*
«FeatureType»
Observation
+ metadata: MD_Metadata [0..1]
+ samplingTime: TM_Object
+ resultTime: TM_Object [0..1]
+ resultQuality: DQ_Element [0..1]
+ parameter: Any [0..*]
constraints
{observedProperty must be member or component of member of featureOfInterest}
{procedure must be suitable for observedProperty}
{result type must be suitable for observedProperty}
«FeatureType»
ObservationCollection
member
1..*
carrierOfCharacteristics
0..*
«metaclass»
GF_PropertyType
{n}
+ definition: CharacterString
+ memberName: LocalName
«instanceOf»
PropertyType
observedProperty
1
result
«type»
Any
{n}

Fig. 1. O&M data model extract [17].

The essential components in SOS are the following:

- *Procedure*: produces the measured value of an observation. This can be a single sensor, a sensor platform, or a numerical simulation process.
- *Offering*: logical grouping of observations related to each other belonging to a common service. For example, the relation can be spatial (share the same location), temporal (created in the same time interval) or due to corresponding properties (measure the same phenomena)
- *ObservableProperty*: a procedure can have multiple observed properties, which represent the physical phenomena measured by a sensor (e.g., temperature)
- *FeatureOfInterest*: Features of Interest (FoI) represent identifiable objects on which sensor systems are making observations. These include spatial information to allow the location to be harvested by OGC service registries.
- *Observation*: contains a measurement value for an observed property of an object under observation (FoI). Observation must include the time stamp when the observation was created.

SOS includes a set of operations for retrieving observations and metadata. The three mandatory core operations are:

- *GetCapabilities*: this operation provides access to metadata and details about the service's capabilities. Either an HTTP GET, or POST request is used to retrieve the service Capabilities document, an XML file containing metadata about the service, like unique identifiers, unique groupings of observations (*Offerings*) and physical phenomena measured by the sensors (*ObservedProperties*).
- *DescribeSensor*: the unique identifiers retrieved in the Capabilities document can be used in the *DescribeSensor* operation to request sensor metadata (SensorML) if the procedure with the identifier is present in the service.
- *GetObservation*: this operation provides access to the observation data made by sensors in the service. A request file containing information from the Capabilities document must be sent via HTTP POST to the server, which then returns the requested observations. Details such as the *Offerings* or the *ObservedProperties*, as well as spatial and temporal filters, can be included as query parameters. SOS returns the requested observation data in O&M format.

To make SOS configurable for any sensor observation project, it also provides transactional operations to insert sensors and observations:

- *RegisterSensor*: This operation allows users to register sensors in SOS. An XML file containing the information about the new sensor in SensorML encoding is sent to the service via an HTTP POST request.
- *InsertObservation*: Observation data from sensors is inserted into SOS via HTTP POST, using an XML file following the O&M specification. The file must specify the procedure which produced the measurement, which in turn must be present within the service.

Among the several open-source implementations of SOS, the most established is 52North-SOS. This Java application is developed at 52North GMBH[1]. It includes a variety of extended features, including support for INSPIRE download service and specialised XML encodings (e.g., WaterML 2.0, GroundWaterML 2), code translators for requests in JSON, SOAP, KVP and POX, a REST API and an extensive client interface for service configuration and data exploration. 52North releases new versions of the software every few months, the most recent one (SOS 4.4.4) at the time of creation of this document available since December 6, 2018.

Pradilla et al. [19] developed SOSLite[2], a lightweight SOS implementation using SOAP binding, XML encoding and storing data in a NoSQL database. SOSLite reduces SOS to its operations to a minimum considering the OGC's best practice recommendations for a lightweight SOS profile for in-situ sensors [20] and aiming to adapt SOS to IoT scenarios. The results show an improvement in response times for several SOS operations. In SOSFul[3] [21], the authors developed SOSLite further, proposing a REST API using JSON encoding format which handles core, transactional and enhanced SOS operations via the core HTTP request types (GET, POST, PUT, DELETE). SOSFul and SOSLite are openly available and were both considered for this experiment. They were eventually discarded due to the lack of documentation and recent development activity, with stalled development in both projects since three and four years respectively. The authors

[1]https://52north.org (accessed on 09.09.2020)
[2]https://github.com/Juanvx/SOSLite (Accessed 24.08.2020)
[3]https://github.com/Juanvx/SOSFul (Accessed 24.08.2020)

in [22] extended OGC SOS with the pagination feature to gain efficiency and can be operated by low-cost devices. This extension has been validated and tested with promising results.

2) SensorThings API: The OGC approved the SensorThings API as an official web standard in 2016. It provides an open standards-based and geospatial-enabled framework to store, manage, expose, and use IoT-based sensor observation data over the web. The SensorThings developers boast it furthers the development of premium quality, lightweight services that cover a broader spectrum of applications [18].

The SensorThings API data model is based on the OGC Observation & Measurement (O&M) model. It consists of a set of interrelated entities, depicted in Figure 2. In contrast to SOS, entities are encoded using JSON format.

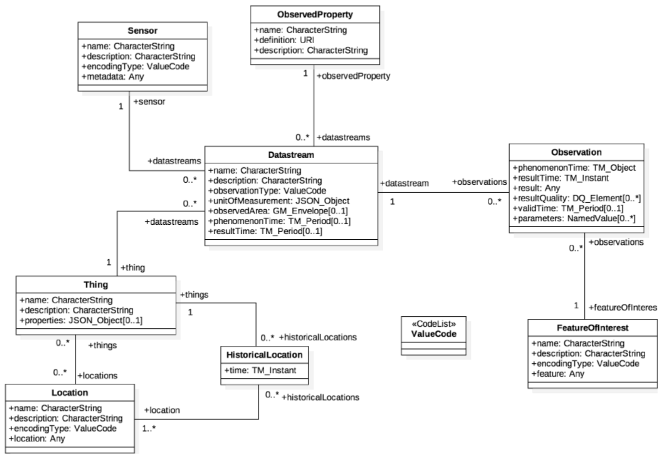


Fig. 2. SensorThings API data model [18].

Brief specifications of the entities as described in the SensorThings API manual in [18] are provided below:

- *Thing*: the Thing entity follows the definition by the International Telecommunication Union (ITU): ”*...with regard to the Internet of Things, a thing is an object of the physical world (physical things) or the information world (virtual things) that is capable of being identified and integrated into communication networks*”.
- *Location*: contains information about the location associated with a corresponding Thing and includes geographical information using GeoJSON encoding.
- *HistoricalLocation*: provides the times of the current and previous Locations of a Thing.
- *ObservedProperty*: specifies the observed phenomenon of measurements.
- *Datastream*: represents the logical grouping of a set of observations and are associated with a single Thing, a single observedProperty and a single Sensor.
- *Sensor*: represents the instrument that observes a property or phenomenon. A Sensor can be associated with multiple datastreams.
- *FeatureOfInterest*: the FoI is the feature being observed. In many cases, the FoI can be identical to the Location of a Thing. In the case of remote sensing, it can be the geographical area or volume being sensed.
- *Observation*: representation of the act of measuring the value of a property at a specified time. Each observation is associated with a single datastream.

SensorThings API data and metadata can be created, read, updated, and deleted with the HTTP protocol (POST, GET, PATCH, DELETE). Each entity has a unique ID and is accessible through the REST API using URLs. The URLs can be chained to access interrelated entities and can be extended using a broad set of query parameters to pinpoint the desired JSON objects.

There are several server implementations of the SensorThings API available as open-source software. The Fraunhofer Open-Source Sensor Things (FROST), a Java application developed by the Fraunhofer IOSB [4] (Institute of Optronics, System Technologies, and Image Exploitation), is considered a well-established, recent SensorThings implementation. FROST-developers are still working on the project to extend its features by the OGC SensorThings API Web Standard.

III. *SEnviro for Agriculture*: A TESTBED IN THE DOMAIN OF SMART FARMING

Like in several other fields, the IoT paradigm shows excellent potential in transforming the agricultural industry by connecting it to the web. Embedded WSN enables new methods to observe and interact with physical objects and promise unprecedented ways to obtain, organise, and consume information [8]. Research projects in IoT and WSN applications for agriculture have been numerous in recent decades [23, 24].

In this work, an IoT application for smart farming is used as a scenario. The application is called *SEnviro for Agriculture* and it is based on the *SEnviro* project [25]. *SEnviro for Agriculture* takes the previously developed environmental monitoring system further and puts it into an agricultural context [11]. The primary objective of *SEnviro for Agriculture* is to design and develop a full system for monitoring crops to improve the production quality and yield. The *SEnviro for Agriculture* monitoring system specialises in observing vineyards. For the sake of simplicity, the remainder of this section refers to *SEnviro for Agriculture* merely as *SEnviro*. Figure 3 shows an overview of the *SEnviro* architecture and components.

At hardware level (represented in the pink section in Figure 3), the *SEnviro* sensorised platform was designed as a smart object, consisting of a similar hardware assembly as the platform presented in [25]. *SEnviro* node components can be categorised into four groups depending on their functions: *core*, *sensors*, *power supply*, and *communication*. *SEnviro* nodes contain sensors for measuring eight meteorological phenomena directly related to plant diseases. These include soil and air temperature, soil and air humidity, atmospheric pressure, rainfall, wind direction, and speed.

The blue section in Figure 3 represents all the elements of *SEnviro* Connect with their corresponding relations. *SEnviro* Connect can be divided into three layers: *data*, *services* and

[4] www.iosb.fraunhofer.de (accessed on 09.09.2020)

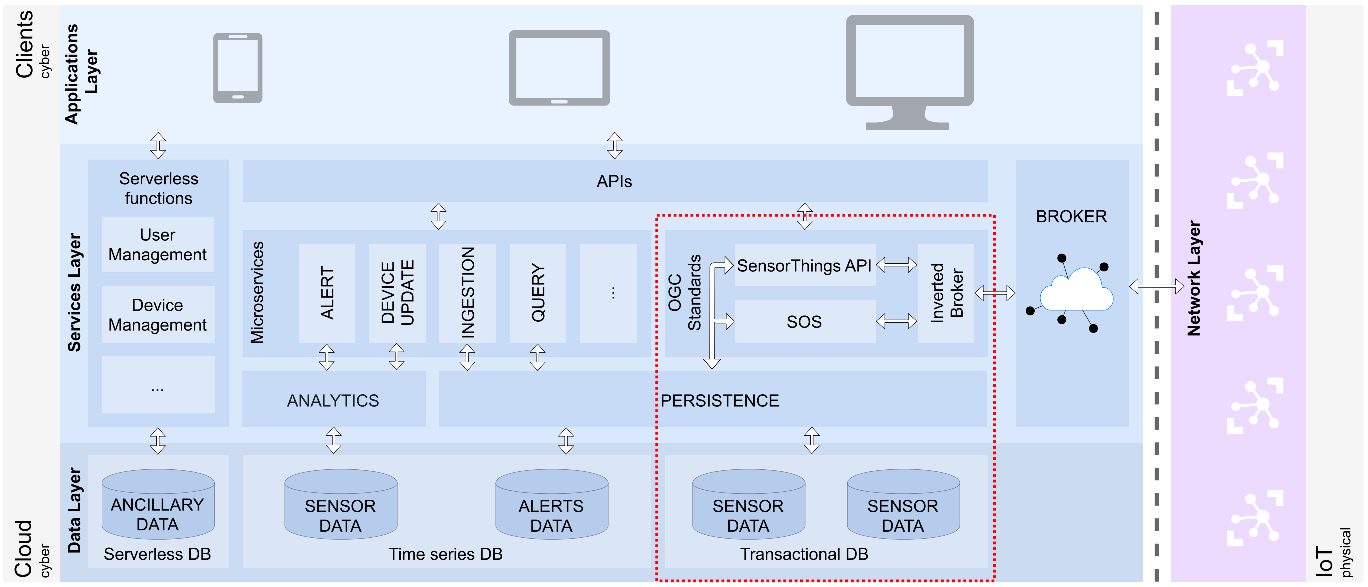


Fig. 3. Schema of *SEnviro* architecture; the purple section represents physical components, the blue section represents software elements. The integration of the Web Standards in *SEnviro* is inside the red box.

applications. The most important part resides in the *services* layer, which can be split into five different components: *broker*, *micro-services* (RESTful API), *persistence module*, *analytics module* and *cloud functions*. The *broker* is used as a bridge to connect *SEnviro* nodes with the software platform. The broker is based on a *RabbitMQ*[5] instance, which supports Message Queuing Telemetry Transport (MQTT) publish-subscribe messaging. All these parts are detailed in [11]. The initial version did not offer an interoperability module. Below we will detail how this module has been added.

SEnviro Connect provides two kinds of analytics: one type focuses on the *SEnviro* node to monitor the node status, such as the battery or last connection, while the other type handles the vineyard use case. The latter bases its analytics on disease models and is supported by alert tasks generated by the *analytics module*. These alert tasks are defined using well-known methods which depend on meteorological phenomena. All the analytics work in real-time, and when a new observation arrives, it is used to calculate each task alert and triggers an alarm for certain types of events.

Five units of the *SEnviro* node have been deployed; four nodes have been installed in vineyard fields in the province of Castello (Spain). The other node was deployed in outdoor environments for testing proposes. The nodes have run continuously and uninterruptedly for 140 days. Each node sent an observation every ten minutes during the vine season 2018. During this time 671,328 observations were collected from the *SEnviro* units. Only 197,887 observations have been used to realise the quantitative performance analysis.

It should be noted that the selected domain is not influential in the comparisons made throughout this study. These could be carried out on another IoT domain, and the outcomes should not vary significantly. The reason for their selection is due to the accessibility of carrying out the evaluation by the authors.

[5]www.rabbitmq.com (accessed on 09.09.2020)

A. Experimental environment

In this experiment 52North-SOS and FROST-Server are integrated the *SEnviro* architecture, representing implementations of *SOS* and *SensorThings API*. The added components are deployed on a main server, centralising all IoT devices and executing all operations autonomously. The same server stores all data and works without computational cost of the IoT devices. Another possible approach is to embed the services in the IoT devices themselves. Since not all current standards provide the necessary support, this approach will be proposed as future work.

To embed the instances into the *SEnviro* architecture, adapter scripts were created to connect standards with the *SEnviro* message broker. The scripts were deployed as stand-alone Docker containers, distributing the incoming observations from *SEnviro* nodes to the corresponding services. In Figure 3 the highlighted section represents the addition to the already established *SEnviro* architecture.

The following sections shed light on the selected web standard applications deployed for this project and the reasons they were chosen. The first section introduces 52North-SOS, based on the OGC SOS. It is followed up by FROST, an implementation of the OGC SensorThings API.

1) OGC SOS: During the selection process of SOS implementations, 52North-SOS was selected due to several reasons. Firstly, 52North-SOS features the SOS test client, a tool for generating and testing sample documents for HTTP requests using several formats including JavaScript Object Notation (JSON). Since JSON objects are structured the same way as Python dictionaries, process automation could be rendered more efficiently in the *SEnviro* integration. Furthermore, 52North-SOS includes tested Docker configuration files, in contrast to other SOS implementations and detailed documentation for all operations Finally, also the fact that 52North-SOS is under ongoing development implies more promising support by the developer community.

52North-SOS runs on an Apache Tomcat[6] web server and stores data in a PostGIS extended PostgreSQL database. The downloadable bundle also includes a user-friendly data exploration tool, the Helgoland Client. The extensive and user-friendly application includes map and diagram visualisation for data. After some trials with the latest 52North-SOS version at the time of the selection process (52North-SOS 4.4.3) and encountering some inconsistencies with the setup in Docker, 52North-SOS 4.4.2 was successfully deployed. The updates in version 4.4.3 were considered irrelevant to the scope of this study. Postgresql 11 and Postgis 2.5 versions were used for both environments.

2) SensorThings API: Of the present SensorThings API implementations, only FROST includes all the features in the OGC compliance test suite and passed it with a full success rate. It includes MQTT extensions for creating and updating data. Furthermore, FROST provides extensive documentation and deployment resources for easy deployment in Docker environments . FROST 1.8 was used to carry out the study, Which was the most recent version at the time of experimentation.

By default, the java-based FROST application launches an Apache Tomcat, but there are also options to configure web server specifications. The application stores all data in a PostGIS extended PostgreSQL database. Fraunhofer IOSB provides several FROST packages, which either comprise HTTP and MQTT operations together or keep them as individual bundles.

B. SEnviro Web Standard Integration

As mentioned, *SEnviro* nodes transmit new values for observed phenomena using a broker. In order to store data in real-time, incoming messages from *SEnviro* must be caught, decoded, processed and posted to the deployed open standard instances. In turn, the deployed standard instances have to be configured for *SEnviro* beforehand in order to store the data correctly. This involved general service configuration and inserting stations and their properties, which was automated using setup scripts and JSON files containing the information for each station.

For the integration of web standards into *SEnviro*, adapters had to be created for each web standard. In the case of both standards, this consisted of connecting to the *SEnviro* broker to intercept messages, decode them, convert them into the right format and post them to the corresponding service via the REST API. Scripts were created in Python for these operations, making use of *Pika*, a Python library to connect to Advanced Message Queuing Protocol (AMQP) -compliant brokers.

[6]http://tomcat.apache.org (accessed on 09.09.2020)
[6]www.github.com/FraunhoferIOSB/FROST-Server (accessed on 09.09.2020)

RabbitMQ supports the AMQP standard and uses *topics* to categorise messages, which can be chained into routing keys. AMQP uses the routing key to intercept messages with specific topics by using * (star) to substitute exactly one word and # (hash) to substitute zero or more words. *SEnviro* routing keys are structured as `current/stationID/phenomenon`. For instance, a *SEnviro* routing key could be:

```
current/2700430019513433334363036/SoilHumidity
```

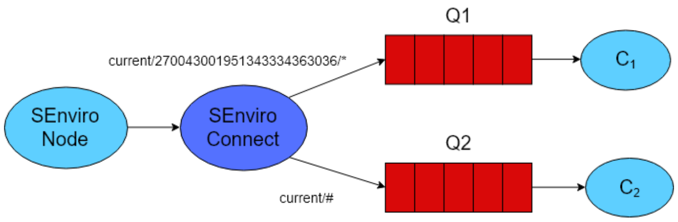


Fig. 4. *SEnviro* message queuing example schema.

In the example in Figure 4, queues *Q1* and *Q2* within *SEnviro* connect intercept messages from *SEnviro* nodes for the message consumers C_1 and C_2. *Q1* queues all messages from station 2700430019513433334363036. *Q2* queues all messages from all stations.

Web standard adapter scripts connect to the *SEnviro* message broker, catch and decode *SEnviro* messages from all deployed *SEnviro* nodes and access the node ID and phenomenon details via the routing key. Using this information, a new message is created and sent to the corresponding web service.

1) SOS Adapter: 52North-SOS supports JSON encoding for inserting observations and a JSON template for this operation is available on the test client of the 52North-SOS interface. This file is loaded into the adapter script and the mandatory information for a successful request inserted. Details about *Procedures* (sensor ID), *Offerings* and *ObservedProperties* are retrieved from the intercepted messages. However, some essential information for a successful *insertObservation* request could not be extracted. Therefore, two workarounds had to be included in the adapter.

Firstly, the unit of measurement in SOS is required in each encoded observation document. This requires including a Python dictionary within the adapter script, matching each phenomenon with the corresponding unit of measurement. Secondly, SOS observation insertions also require the coordinates where the observation was created. Therefore, an external JSON file containing objects with the station ID and the corresponding coordinates as attributes has to be loaded into the script. Once all the information for the observation insertion is complete it is posted to SOS via an HTTP POST request.

2) SensorThings Adapter: Similar to the SOS adapter, reads incoming messages and uses information from the messages to create a JSON object to post to the FROST server with an HTTP POST.

In order to post to the correct datastream, the corresponding datastream ID is required in the target URL. The script does this by first requesting all datastream IDs with their corresponding names via HTTP GET request. Datastream names in the *SEnviro* FROST instance are defined as a combination of the node ID and the measured phenomenon, which are both present in incoming byte messages. The script selects the datastream ID by matching the information from the message with the datastream name. The following example target URL[7] posts observations to the datastream with ID = 1:

```
+serverPath:port/FROST-Server/v1.0/Datastreams(1)
/Observations
```

IV. Comparative Analysis

To shed light on the potential of open standard integration to enhance environmental monitoring application interoperability, a qualitative and a quantitative analysis were performed. This section structures the framework defined to realise the comparative.

In the qualitative analysis the deployment and configuration process for each web service was described and compared. Subsequently, a comparison of available operations for different uses was performed from data producer and data consumer perspectives. On the data producer side insert, update and delete operations were evaluated:, where entities are understood to be: sensors, sensor platforms, and observations. On the consumer side, data and metadata querying and fetching operations were evaluated. Web standard semantics were evaluated based on their encoding formats and data traffic protocols, and they are based on the web standard definition and not the development for each standard.

The quantification of differences in performance between the deployed web services were monitored on various levels and using some well-known tools. These were used to monitor *response time*, *response size*, *CPU* and *memory* usage. All tests have been performed in a virtual machine with four logic CPUs and 8 GB RAM maximum size. The underlying cluster has the following features: 4x Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz, 16,719 MB and Operating System: Ubuntu 16.04.4 LTS. Unlike the qualitative analysis, quantitative results are dependent on the particular web standard implementation and may vary between implementations of the same web standard.

As noted above, the essential feature in terms of performance is the observation retrieval since it is the most used operation to visualise phenomena's behaviour over time. Hence the quantitative comparison was done using this operation.

The following tools were used for performance monitoring:

- *Postman*: Postman[8] is a powerful HTTP Client desktop application for testing web services. Users can create both simple and complex HTTP requests, which return the request status, response times and the size of the returned file.
- *JMeter*: JMeter[9] is a project by the Apache Software Foundation. It is an open-source Java desktop application, designed to measure web applications' and distributed systems' performance and stress test their functional behaviour.
- *cAdvisor*: cAdvisor is provided by Google to monitor Docker containers' behaviour. Apart from a simple user interface showing graph visualisation of the container metrics, *cAdvisor* provides several APIs for accessing container metrics data.

Response times and sizes are measured for observation retrieval operations using Postman monitors. Since Postman monitors run as cloud services, test queries do not depend on the local machine's network connectivity (with low latency) once they have been deployed. Requests for observations are monitored for 24 hours, with two requests per hour, resulting in 48 values per query. Postman monitors periodically recalculate the average response time for each number of requested observations, returning a single average value per request.

To compare the performance for observations retrieval, identical conditions were created for different services. Queries using the same parameters request identical sets of observations with the corresponding REST API of each service. This approach includes queries to obtain sets of 1, 100, 200, 400, 500, 600, 800 and 1,000 observations. The maximum of 1,000 observations was selected due to the FROST default configuration, which sets the maximum number of FROST observations contained in a single response file to 1,000.

The workflow for metrics monitoring relies on measuring the metrics of the individual Docker containers. *cAdvisor* provides the means to access the container metrics data via an API. The selected REST API [10] returns JSON objects containing metrics data. The API was configured to return a single measurement and a container monitoring script was created in Python to send the API request every second once the script is run. Since each service has separate containers for the web applications and databases, CPU and memory values from both the containers are added to show the full amount of resources used by the corresponding standard implementations. The container CPU values are divided by the server CPU usage to reflect how many server resources the containers require in percentage. Memory values are calculated in bytes and then converted to megabytes for data visualisation.

The HTTP requests created in JMeter were configured to run for three minutes launching an HTTP request per second. 52North-SOS and FROST requests for the different quantities of observations were launched simultaneously to the container monitoring script, resulting in approx. 180 values per query.

V. Qualitative evaluation

A. Service setup & configuration

52North-SOS and FROST require distinct setup process and service configuration.

[7]For all examples *serverPath* is located at http://elcano.init.uji.es
[8]www.getpostman.com (accessed on 20.03.2020)
[9]https://jmeter.apache.org/download_jmeter.cgi (accessed on 09.09.2020)
[10]https://github.com/google/cadvisor/blob/master/docs/api_v2.md (accessed on 09.09.2020)

1) 52North-SOS: 52North-SOS includes a Graphical User Interface (GUI) with a broad set of service configuration options. The interfaces enable users to configure most of the service's specifications, including among others the available SOS operations with their bindings and encodings, the data source, the spatial reference system, the services' and data source's timezones, access rights and logging.

For the *SEnviro* configuration of 52North-SOS, transactional security was disabled, enabling the transactional SOS operations for inserting sensors and observations. The *SEnviro* nodes were inserted by posting a preconfigured *InsertSensor* JSON object to the server for each node. The object contains information about the service provider, the node ID, the measured phenomena and the node location. As mentioned in Section III, creating the adapter to divert *SEnviro* observations into SOS required an extra file containing the coordinates of the stations.

2) FROST-SensorThings API: FROST-Server does not include a GUI for service configuration. Service settings are configurable using environment variable or in a XML configuration file. Since *SEnviro* nodes all monitor the same phenomena and are composed of the same sensor constellation, data about observed properties and sensors were inserted in a first step. JSON objects relating to the sensors and observed properties were subsequently used to insert things and datastreams.

B. Register/Update/Delete Things

52North-SOS and FROST handle the insertion of data based on the corresponding SOS or SensorThings API operations in order to remain OGC compliant. In SOS these are the *Transactional Operations*, which are HTTP POST requests to the SOS service URL and include *RegisterSensor* and *InsertObservation*. 52North-SOS extends the transactional capabilities with the *DeleteSensor* and the *UpdateSensorDescription* operations. The SensorThings API supports HTTP request types (GET, POST, PUT, DELETE) for creating, updating and deleting entities. FROST has fully implemented the SensorThings API's sensing functionalities with no significant additions, and therefore this section will refer to the SensorThings API operations directly.

1) 52North-SOS: Before observations can be inserted into a SOS deployment, entities need to be inserted representing the devices generating the observation data and must also include information about the phenomena they measure for a successful observation insertion.

52North-SOS supports several encoding formats, including JSON, which is used in this project, as mentioned in Section II. The 52North-SOS equivalent to the *RegisterSensor* operation is *InsertSensor*. For this operation, an *InsertSensor* request file must be posted to the server. An example JSON object[11] contains the mandatory information for a successful insertion request.

The object must contain a list of details, including unique procedure ID, long procedure name and short name, its offering and its observed properties. For each of these details SML, SWE and GML tags are added to make it ensure its interoperability with other entities of the sensor web. This becomes visible in the large string of XML code in the *procedureDescriptionFormat* property of the JSON object, which is the XML version of the *InsertSensor* operation and is mandatory in the JSON version of the POST request. As a consequence, procedure insertion needs the XML version of the operation, even if the SOS application uses the JSON version of the operation, adding a full step to the workflow and inflating the size of the final JSON object to post to the server to 4,829 bytes. The *InsertSensor* request creates all the necessary entities for the insertion of observation, including its related offering, observed property and feature of interest.

Since SOS 2.0 included some operations to delete or update details of procedures, 52North added these operations as extended operations. *UpdateProcedureDescription* enables the modification of station details, which resembles the InsertSensor protocol. As in the example JSON code above, the file to be posted to the server requires the full XML code as a string value of the corresponding JSON property.

DeleteSensor allows procedures and their affiliated observations to be removed from the service. This requires posting a request file containing the procedure unique ID to the server (Listing 1). The file is comparatively small in size, as demonstrated in the JSON version of the *DeleteSensor* request below. The mandatory SOS offering created with the procedure will remain in the service even when the linked procedure is deleted. *Offering* names act as unique identifiers, which means if a procedure is reinserted it will need a new offering ID.

Listing 1. JSON example of *DeleteSensor* in SOS.

```
{"request": "DeleteSensor",
 "service": "SOS",
 "version": "2.0.0",
 "procedure": "0123456789012345678901 23" }
```

2) FROST-SensorThings API: The SensorThings API data model demands a different approach when inserting data. Every entity within a SensorThings API has its unique ID. It can be referred to by its unique URL for creating further entities, updating their details and properties and also deleting them. This makes data management and system maintenance highly flexible and efficient.

As in SOS, the devices generating the observation data, must be created to enable the storage of observations. The five SensorThings API key components (*Things*, *Datatreams*, *ObservedProperties*, Sensors and Observations) are interrelated, with the *Datastreams* as core entity. JSON objects need to be sent to the server via HTTP POST using the corresponding target URL to create entities. Target URLs are composed of the base URL and */entity*. The example URL below targets the Thing class:

```
+serverPath:port/FROST-Server/v1.0/Things
```

When new entities are created, a unique ID is automatically assigned. If an entity is removed its ID remains stored in the system and cannot be used again. All entities have manda-

[11]An example of 52North-SOS InsertSensor - https://bit.ly/3d2Bzva (accessed on 15.06.2020)

tory properties that must be included when posting to the server. Metadata about the specific Thing can be added in the *properties* field. Furthermore, entities can be extended with their related entities as optional properties when creating them. These extended entities can either be generated within the creating process or they can refer to already existing entities. Extended properties can again be extended, meaning all necessary entities can be created with a single HTTP POST request. This is shown in the following Thing object example (Listing 2).

Listing 2. JSON example of inserting a Thing in SensorThings API.

```
{"name": "01234567890123456789 01234",
 "description": "A SensorThings station",
 "properties": {
  "owner":"Universitat Jaume I",
  "maintainer":"student al374901"},
 "Locations": [{
  "name": "carcagente26_1",
  "description": "Carcagente 26",
  "encodingType": "application/vnd.geo+json",
  "location": {
   "type": "Point",
   "coordinates": [-0.031525, 39.980187]}}],
 "Datastreams":[{
  "name": "AirTemperature-012345678901234567890
  1234",
  "description": "Datastream for recording air
  temperature",
  "observationType": "http://www.senviro.uji.es/",
  "unitOfMeasurement": {
   "name": "Degree Celsius",
   "symbol": "°C",
   "definition":
   "http://www.qudt.org/qudt/owl/1.0.0/unit/
   Instances.html#DegreeCelsius"},
  "ObservedProperty": {
  "name": "Si7021-A20",
  "description": "Monolithic CMOS IC integrating
  humidity and temperature sensor elements, an
  analog-to-digital converter, signal processing,
  calibration data, and an I2C Interface",
  "encodingType": "application/pdf",
  "metadata":
  "https://www.silabs.com/documents/public/data-
  sheets/Si7021-A20.pdf"},
 "Sensor": {
  "@iot.id": 1 }}
```

The example JSON object above (Listing 2) creates a Thing with its mandatory properties (name and description). It also creates its related location and datastream by adding corresponding properties to the object. The embedded datastream creates a related observed property and links to an already registered sensor. Metadata about the owner and maintainer are added in the object of the *properties* key value.

SEnviro nodes were created with no extended properties since the observed properties and sensors were inserted in a previous step of the setup. The size for necessary JSON object to create a *SEnviro* node with its complete set of datastreams is 4,731 bytes.

Properties of any SensorThings API entity can be updated by executing an HTTP Patch request its unique URL with a JSON object containing the properties to be updated and the new values. The example JSON object and the target URL shown below update the name and description properties of a registered sensor with ID=1.

```
+serverPath:port/FROST-Server/v1.0/Sensors(1)
```

Listing 3. JSON example to update a Thing in SensorThings API.

```
{ "name":"SparkfunSoilMoistureSensor",
  "description": "Measures soil moisture" }
```

Entities can be deleted by using its unique URL in an HTTP DELETE request. Deleting Things will remove all their related datastreams including their affiliated observations, but will not remove sensors or observed properties.

C. Insert/Update/Delete observations

In the same way that operations for the management of Things are offered, the two standards define the operations necessary for the treatment of the observations generated by Things.

1) 52North-SOS: After setting up sensors and their properties within SOS, observations can be inserted using the *InsertObservation* operation. This operation is also executed by sending a JSON object containing the necessary details for a successful insertion via HTTP POST to the service URL. The object must include the ID of the procedure of origin, its offering ID, the observed property ID, details about the feature of interest (ID, coordinates, spatial reference system, sampled feature), unit of measurement and the time and value of the observation. The object size for *SEnviro InsertObservation* requests is approximately 1,185 bytes. The example JSON *InsertObservation* (Listing 4) request below shows all the mandatory details required.

Listing 4. JSON example of *InsertObservation* in SOS.

```
{ "request": "InsertObservation",
 "service": "SOS",
 "version": "2.0.0",
 "offering": "offering01234567890123456789 01234",
 "observation": {
  "identifier": {
   "value": "1",
   "codespace": ""},
  "type":
  "http://www.opengis.net/def/observationType/
  OGC-OM/2.0/OM_Measurement",
  "procedure": "01234567890123456789 01234",
  "observedProperty": "AirTemperature",
  "featureOfInterest": {
   "identifier": {
    "value": "featureOfInterest012345678901234567890
    1234",
    "codespace": ""},
    "name": [
     { "value": "01234567890123456789 01234",
       "codespace": "" }],
    "sampledFeature": [
     "parent" ],
    "geometry": {
     "type": "Point",
     "coordinates": [
      -0.073863,
      39.993934 ],
     "crs": {
      "type": "name",
      "properties": {
       "name": "EPSG:4326" }}}},
  "phenomenonTime": "2018-11-30T16:53:43+00:00",
  "resultTime": "2018-11-30T16:53:43+00:00",
  "result": {
   "uom": "°C",
   "value": 84.621094}}}
```

SOS observations can be assigned a unique identifier. When inserting observations, SOS rejects the request if an observation with the same identifier is already present in the database. This detail is an optional property in the *InsertObservation* request, but should always be added to facilitate calling specific observations from the service and to make insert operations idempotent.

52North-SOS has included the *DeleteObservation* operation into the service. This operation enables clients to remove observations from the service by posting a JSON object to the server containing either details about the related entities (e.g., *Procedures*, *Offerings*...) or a temporal filter. Single observations can also be removed by using the identifier mentioned above. Deleting observations does not remove items from the database, but instead sets the attribute deleted (boolean) of the observation items in the PostgreSQL database to from ”F” to ”T”.

2) FROST-SensorThings API: Inserting SensorThings API observations is done by sending the observation JSON object to the corresponding target URL, composed of the Datastream URL and */Observations*. The JSON object must include the result time as a string in *ISO 8601* format and the value of the measurement, as shown below (Listing 5). The approximate size of *SEnviro* observations posted to FROST by the adapter is 65 bytes.

Listing 5. JSON example of *InsertObservation* in SensorThings API.

```
{ "resultTime" :  "2019-01-14T12:35:47.000Z",
 "result" : 0.327 }
```

As showed above, in SensorThing API, each entity can be updated using an HTTP Patch request using a unique URL with a JSON object containing the new observation to be updated. To delete an observation, the HTTP DELETE request has to be used over the observation entity.

D. Retrieving metadata

The values and timestamps of observations hold little value without knowing their origin, their nature and the purpose why they are created. Therefore, it is crucial to obtain not only the observations themselves but also information about the sensors and their locations, the features they are observing and the measured phenomena.

1) 52North-SOS: SOS defines a set of operations to retrieve metadata from various sources within the service. 52North-SOS includes these operations and features a couple of further operations to add functionality. The essential SOS operation for retrieving service information is the *getCapabilities* operation, which provides clients with the complete service metadata about the deployed service, including information about the tightly-coupled data served [17]. The code below shows the 52North-SOS JSON version of the request (Listing 6). An example for 52North-SOS *getCapabilities* response is not presented here, considering the space needed to show the response object (over 1,600 lines).

Listing 6. JSON example of a *GetCapabilities* request in SOS.

```
{ "request": "GetCapabilities",
  "service": "SOS",
  "sections": [
  "ServiceIdentification",
  "ServiceProvider",
  "OperationsMetadata",
  "FilterCapabilities",
  "Contents" ]}
```

Further operations use similar requests to get information about node locations (*getFeatureOfInterest*) and about the relations between measured phenomena, stations and features of interest (*getDataAvailability*). Finally, *DescribeSensor* gets the complete details of a present SOS procedure, including the station owner and maintainer with contact details, the observed phenomena, the SOS offering, the location and the time of registration in the service.

2) FROST-SensorThings API: HTTP Get requests in the SensorThings API are capable of accessing and obtaining all the information of all the present entities by using the extendable URLs to target, select, enrich output data. The most important query operators for metadata queries are *$expand* and *$select*. The *$expand* operator will add related entities to the output of the requested entity provided they have a direct relationship (see Figure 2). The expanded entity can again expand directly related entities, allowing users to dig into the data architecture. This is shown in the following target URL and its corresponding JSON output (Listing 7).

```
+serverPath:port/FROST-Server/
v1.0/Things(1)?$expand=Datastreams($expand=
ObservedProperty)
```

Listing 7. JSON response from an expand query option to display a Thing in SensorThings API.

```
{"name" : "270043001951343334363036",
"description" : "SEnviro monitoring station with ID:
270043001951343334363036",
"Locations@iot.navigationLink" :
"+serverPath:port/FROST-Server/v1.0/
Things(1)/Locations",
"HistoricalLocations@iot.navigationLink" :
"+serverPath:port/FROST-Server/v1.0/
Things(1)/HistoricalLocations",
"Datastreams@iot.navigationLink" :
"+serverPath:port/FROST-Server/v1.0/
Things(1)/Datastreams",
"Datastreams" : [{
 "name" : "Battery-270043001951343334363036",
 "description" : "Datastream for recording battery
 status",
 "observationType" : "http://www.senviro.uji.es/",
 "unitOfMeasurement" : {
  "name" : "Percent",
  "symbol" : "%",
  "definition" : "https://en.wikipedia.org/wiki/
  Percentage" },
 "phenomenonTime" : "2019-01-30T14:27:06.168Z/
 2019-01-30T 14:46:55.060Z",
 "resultTime" : "2019-01-30T14:26:27.000Z/
 2019-01-30T 14:46:14.000Z",
 "ObservedProperty" : {
  "name" : "Battery",
  "definition" : "https://en.wikipedia.org/wiki/
  Electric_battery",
  "description" : "Battery readings in \%",
  "@iot.id" : 9,
  "@iot.selfLink" : "+serverPath:port/
  FROST-Server/v1.0/ ObservedProperties(9)" },
 "@iot.id" : 22,
 "@iot.selfLink" : "+serverPath:port/
 FROST-Server/v1.0/ Datastreams(22)" }],
"MultiDatastreams@iot.navigationLink" : "http://
+serverPath:port/FROST-Server/v1.0/
Things(1)/MultiDatastreams",
"@iot.id" : 1,
"@iot.selfLink" : "+serverPath:port/
FROST-Server/v1.0/ Things(1)" }
```

The Thing with ID=1 is returned with all its main properties and with its expanded datastreams, which in this case is *”Battery-27004300195134333436303 6”*. The expanded datastream is once again expanded to show its related observed property.

In contrast to the *$expand* operator, the *$select* operator allows users to select only certain properties of entities for the JSON output. This can be used to reduce the size of the output files by selecting only specific information of entities.

E. Observation retrieval

As one of the core features in both 52North-SOS and FROST-SensorThings API, the services store observation data over time and make them available in the WWW. Each service has its way to access the stored observation data.

1) 52North-SOS: SOS observations are obtained using the *getObservation* request. A JSON file containing the query parameters is posted to the server (Listing 8), which delivers a response file containing the requested observations. The query can include one or more parameters among *procedure*, *offering*, *observedProperty*, *featureOfInterest*, a *spatialFilter* or a *temporalFilter*.

Listing 8. JSON example of *GetObservation* in SOS.

```
{ "request": "GetObservation",
  "service": "SOS",
  "version": "2.0.0",
  "procedure": "270043001951343334363036",
  "offering": "offering270043001951343334363036",
  "observedProperty": "AirTemperature",
  "featureOfInterest": "featureOfInterest270043001
  951343334363036",
  "spatialFilter": {
    "bbox": {
      "ref": "om:featureOfInterest/
      sams:SF_SpatialSamplingFeature/sams:shape",
      "value": {
        "type": "Polygon",
        "coordinates": [[
        [-0.07902860641479492,39.99347896173187],
        [-0.07259130477905273,39.9903390214231],
        [-0.0714111328125,39.99696396205215],
        [-0.07902860641479492,39.99347896173187]]]}}},
  "temporalFilter": {
    "during": {
      "ref": "om:phenomenonTime",
      "value": [
        "2018-11-29T14:43:00+00:00",
        "2018-12-13T15:32:12+00:00"]}}}
```

Adding query parameters will narrow down the output observations. Spatial filters are provided in GeoJSON encoding, and temporal filters must support ISO8601 format.

52North has added support for the *GetObservationById*, an operation to obtain observation by its unique identifier. A file needs to be posted to the server containing the observation identifier. This returns the observation in the same format as *GetObservation*, but adds the identifier with its value as an attribute. The identifier must be known to the client before invoking the operation. Example request and response objects are presented below (Listings 9 and 10).

Listing 9. JSON example of a *GetObservationById* request in SOS.

```
{ "request": "GetObservationById",
  "service": "SOS",
  "version": "2.0.0",
  "observation": ["2"]}
```

Listing 10. JSON example of a *GetObservationById* response in SOS.

```
{ "type" : "http://www.opengis.net/def/observation
  Type/OGC-OM/2.0/OM_Measurement",
 "identifier" : {
  "codespace" : "http://www.opengis.net/def/nil
  /OGC/0/unknown",
  "value" : "2" },
  "procedure" : "270043001951343334363036",
  "observableProperty" : "Battery",
  "featureOfInterest" : {
   "identifier" : {
    "codespace" : "http://www.opengis.net/def/nil/
    OGC/0/unknown",
    "value" : "featureOfInterest27004300195134333
    4363036" },
   "name" : {
    "codespace" : "http://www.opengis.net/def/nil/
    OGC/0/unknown",
    "value" : "270043001951343334363036" },
    "sampledFeature" : "parent2700430019513433343
    63036",
    "geometry" : {
     "type" : "Point",
     "coordinates" : [
      40.133098, -0.061 ]}},
  "phenomenonTime" : "2018-12-26T13:47:58.000Z",
  "resultTime" : "2018-12-26T13:47:58.000Z",
  "result" : {
   "uom" : "%",
   "value" : 81.726563 }}
```

2) FROST-SensorThings API: Several SensorThings API query operators come in useful to query observations. The *$orderby* operator is used to sort the output JSON objects, which can be extended with suffixes for descending or ascending order (*desc*, *asc*). The number of output objects is specified with the *$top* and the *$skip* operator allows the user to skip a specified number of observations. The *$count* operator returns the number of queried observations as a JSON property at the top of the output file. The above-mentioned operators are used in the query URL below:

```
+serverPath:port/FROST-Server/v1.0/
Datastreams(1)/Observations?$count=true&$skip=500&
$top=50&$select=resultTime,result&$orderby=result
```

The query returns the top 50 observations from the datastream with ID=1, skipping the first 500 values and ordering by result value. The total amount of observations is counted, and the output file only returns the time stamps and the result values of the observations.

SensorThings API also features the *filter* operator. This highly configurable operator is used to make complex queries using a set of over 35 in-built operators and functions. By using the operators and functions, the SensorThings API has extensive possibilities of combining the various query operators and function as filters for pinpointing specific data. The following example URL selects datastreams containing ”SoilHumidity” as a substring in the name property and expands the selected datastreams’ observations that have values lower than 2,500 m^3/m^3.

```
+serverPath:port/FROST-Server/
v1.0/Datastreams?$filter=substringof('SoilHumidity'
,name)&$expand=Observations($filter=result lt 2500)
```

While the maximum number of observations in a single response file is limited in the SensorThings API, the output file always includes a link to the next set of observations until all the observations called with the query parameters have been served. Output objects for unmodified FROST observation requests have the following format (Listing 11).

Listing 11. JSON example of *InsertObservation*

```
{ "@iot.nextLink" :
"+serverPath:port/FROST-Server/
v1.0/Observations?$top=1&$skip=1",
"value" : [{
 "phenomenonTime" : "2018-11-26T13:48:03.946Z",
 "resultTime" : "2018-11-26T13:47:26.000Z",
 "result" : 21.136395,
 "Datastream@iot.navigationLink":
 "+serverPath:port/FROST-Server/v1.0/
 Observations(1)/Datastream",
 "FeatureOfInterest@iot.navigationLink":
 "+serverPath:port/FROST-Server/v1.0/
 Observations(1)/FeatureOfInterest",
 "@iot.id" : 1,
 "@iot.selfLink" :
 "+serverPath:port/FROST-Server/v1.0/
 Observations(1)" }]}
```

VI. Quantitative results

52North-SOS supports all the standard SOS operations, but their data visualisation tool, the Helgoland Client, uses its request method and API to obtain observations. Helgoland is therefore analysed separately to show its potential in the response time analysis. The data retrieved by the Helgoland Client contains only information about the time and value of observations, improving performance. An equivalent FROST-SensorThings API query, limiting result details to observation time and value, was created for comparison with Helgoland (called FROST reduced). The test queries below obtain 1,000 observations.

- **52North-SOS - getObservation**: This query uses the standard SOS *getObservation* request. A getObservation request file containing query parameters for *procedure* (monitoring station), *observed property* and *temporal filter*, is posted to the server via HTTP POST using the service URL. The server then sends a response file with the corresponding observations. Standard SOS uses XML encoding, but 52North-SOS supports JSON format, which is used in this request.
  - URL:

    ```
    +serverPath:port/52n-sos-webapp/service
    ```

  - Post data:

    ```
    {"request": "GetObservation",
     "service": "SOS",
     "version": "2.0.0",
     "procedure": "270043001951343334363036",
     "observedProperty": "Battery",
     "temporalFilter": {
       "during": {
         "ref": "om:phenomenonTime",
         "value": ["2019-01-14T14:14:52.000Z",
                "2019-01-21T11:20:52.000Z"]}}}
    ```

- **52North-SOS Helgoland Client**: 52North-SOS Helgoland Client queries are executed on the client interface. The query parameters are inserted by selecting the *procedure* on a map, *observed properties* (phenomena) from a list. The time parameters are selected in a consecutive step. The API URL is then compiled and sends an HTTP GET request to the server, which returns the requested values and time stamps and displays them in an interactive diagram.

  ```
  +serverPath:port/52n-sos-webapp/
  api/datasets/quantity_9/data?expanded=true&
  format=flot&generalize=false&locale=de&
  timespan=2019-01-14T14:14:52%2B01:00%
  2F2019-01-21T11:20:52%2B01:00
  ```

- **FROST**: FROST observations are obtained with an HTTP GET request. The target URL is extended with the query parameters. This query URL uses *top* to specify the number of obtained observations and *filter* to add time constraints.

  ```
  +serverPath:port/FROST-Server/v1.0/
  Datastreams(18)/Observations?$top=1000&$filter=
  phenomenonTime%20gt%202019-01-14T14:14:52Z%20
  and%20phenomenonTime%20lt%202019-01-21T11:20:52Z
  ```

- **FROST (reduced)**: Here the query URL from above is further extended with the *select* operator, allowing the restriction of output attributes of the obtained observations.

  ```
  +serverPath:port/FROST-Server/v1.0/Datastreams(18)
  /Observations?$top=1000&$select=phenomenonTime,
  result&$filter=phenomenonTime%20gt%202019-01-14T
  14:14:52Z%20and%20phenomenonTime%20lt%20
  2019-01-21T11:20:52Z
  ```

The results for the monitored performance parameters were extracted, stored and analysed. Response times and file sizes were measured using Postman collections, while JMeter and cAdvisor was used to monitor CPU and memory.

A. Client side: response times and sizes

This section contains graph visualisation for response time and size data from the client point of view. Figure 5 shows the sizes of the response files of the different queries by the different services. The output file size for 52North-SOS is the largest, resulting in 1,110 kb at 1,000 observations. Helgoland Client requests return the smallest file sizes, with returned files holding 22.77 kb at 1,000 observations, though the files also contain the shortest amount of metadata. Standard FROST-SensorThings API requests reach up to 557 kb at 1,000 observations, which is reduced to 81.28 when reducing output files to contain only timestamps and values. Figure 5 shows the file sizes growing at a linear rate.
An essential feature in both SOS and SensorThings API is the SWE DataArray format. This feature is used to reduce the size of the observations transmitted over the network for measurement insertion or retrieval. The SWE DataArray contains 1) values metadata: number of values, time and values encoding, and 2) values: sensor data according to the

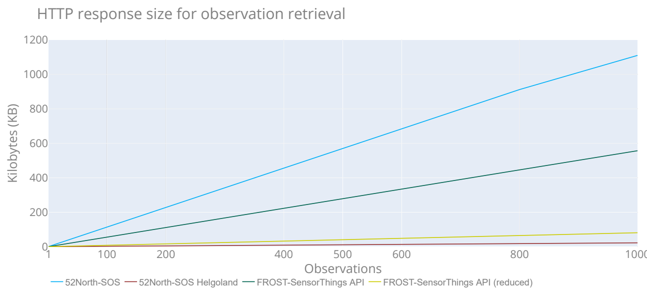


Fig. 5. Response sizes between 52n-SOS, 52N-SOS Helgoland, FROST-SensorThings API and FROST-SensorThings API (reduced).

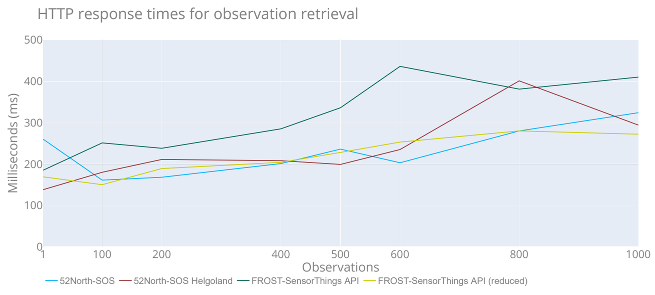


Fig. 6. Response times between 52n-SOS, 52N-SOS Helgoland, FROST-SensorThings API and FROST-SensorThings API (reduced).

value's metadata. In 52North SOS, a request for 100 SEnviro observations reaches from a size of 107 KB in the default format to 8.77 KB using the *getObservation* operation with the *MergeObservationsIntoDataArray* parameter set to true.
Similar to the SWE DataArray in the OGC SOS, SensorThings API also provides the support of DataArray to aggregate multiple Observation entities and reduce the request and response size. SensorThings API mainly uses SWE DataArray in two scenarios: (1) get observation entities in SWE DataArray, and (2) create observation entities. In FROST a request of 100 SEnviro's observations is 48.6 Kb, and with the SWE DataArray extension, we obtain a response of 9.63 KB. In order to request for DataArray, users must include the query option *$resultFormat = dataArray* when requesting Observation entities. FROST response sizes using SWE DataArray are larger because for each observation five different components are added (" *result* "," *resultTime* "," *resultQuality* "," *validTime* "," *parameters* ").
Response times between all implementation variants are displayed in Figure 6. This chart shows the fastest response times for reduced FROST requests at an average speed of 218 ms, closely followed by 52North-SOS and Helgoland Client at 229 ms and 233 ms respectively. FROST requests hold the longest response times with an average of 315 ms. Generic FROST observation requests have the highest response times in most of the requests, peaking at 436 ms at the 600 observation mark. Standard 52North-SOS *getObservation* requests are faster than FROST by an average of 86.12 ms, though they take longest for a single observation, FROST by 75 ms. The Helgoland Client's API shows a similar behaviour to 52North-SOS, but spikes when requesting 800 observations with an average response time of 401 ms. In terms of computational time, all scenarios follow a linear time ($O(n)$).

B. Server side: web service metrics

The observation retrieval HTTP requests used for the response times and sizes analysis for FROST-SensorThings API and 52North-SOS were configured in JMeter and executed in test runs of three minutes with one request per second as done in [26, 27]. Using the cAdvisor API and the automated requests in JMeter, container CPU usage and memory usage were extracted and written into CSV files by the container monitoring script. Figure 7 shows graph visualisations of the CPU metrics output for both services. 52N-SOS Helgoland and FROST-SensorThings API (reduced) are not considered because from the client side they carry the same computational cost as the original standards.

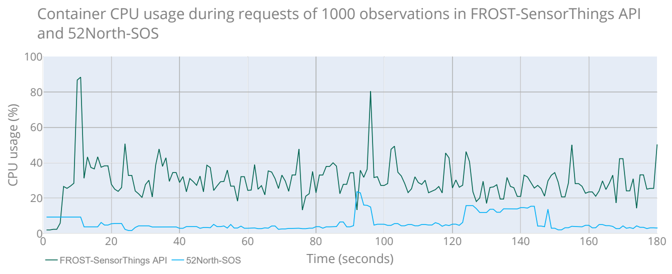


Fig. 7. CPU usage behaviour from FROST-SensorThings API and 52North-SOS during responses of 1000 observations.

Plotting the results for both services in active state (Figure 7) shows FROST's higher CPU usage. FROST-SensorThings API uses an average of 22.58% more processing power than 52North-SOS during observation requests. The difference in the CPU maximum between the two services lies at 86.03%.
Figure 8 shows the average CPU values of both FROST-SensorThings API and 52North-SOS increasing amounts of requested observations. FROST-SensorThings API average CPU usage shows all values for different response sizes between 4.4% and 22.3%. In 52North-SOS, average CPU values lie between 2.6% and 3.1%. Neither of the services show a continuous increase of CPU usage with increasing response sizes in observation requests, instead rising and falling seemingly at random. The colour areas show the standard deviation; CPU usage is more fluctuant in FROST-SensorThings API measurements than in 52North-SOS.
The CPU usage behaviour for FROST-SensorThings API and 52North-SOS, both services show a noticeable activity during constant observation requests. The usage does not change with increasing response sizes up to 1,000 observations. Overall, 52North-SOS shows lower and more stable CPU values.
The graph in Figure 9 shows the RAM behaviour under request activity of 1,000 observations. Both graphs show very little activity during the monitoring run, though the average values are higher than in idle state. In idle state, 52North-SOS uses approx. 2,572MB RAM, while FROST-SensorThings API

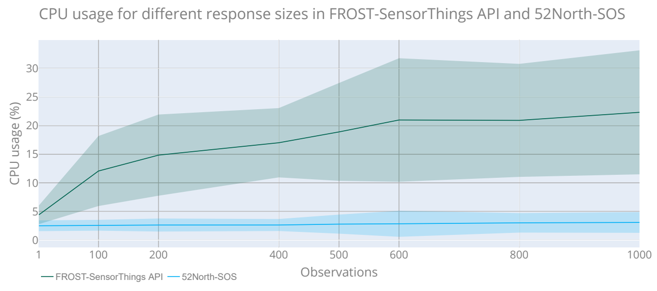


Fig. 8. CPU usage for different response sizes in FROST-SensorThings API and 52North-SOS.

requires 3,116MB. These values include the RAM for the corresponding database containers, which may increase with a growing amount of data. FROST-SensorThings API average memory usage in an active state lies at 3,363MB, resulting in an increase of 247MB. Similarly, 52North-SOS memory usage lies at 2,741.7MB, incrementing RAM usage by 170MB.

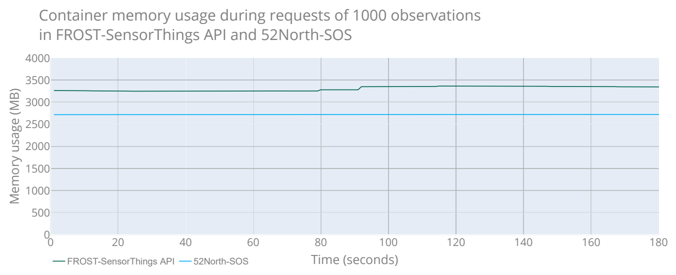


Fig. 9. Memory usage behaviour from FROST-SensorThings API and 52North-SOS during responses of 1000 observations.

When looking at the RAM behaviour during different observation requests, average values do not show significant changes reflecting the response sizes (Figure 10). This testing method was repeated several times and showed slightly different results every time, indicating that the memory is not meaningfully affected by observation requests. In this case, the standard deviation is grouped in both developments.

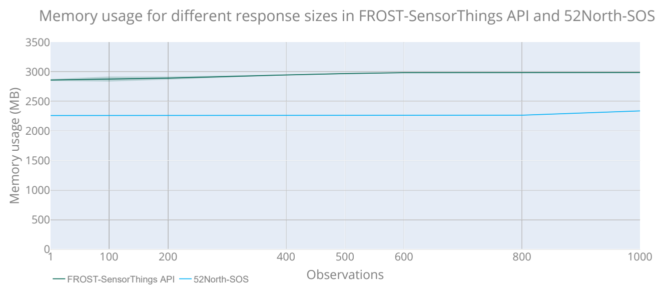


Fig. 10. Memory usage for different response sizes in FROST-SensorThings API and 52North-SOS.

VII. Discussion

When comparing SOS and SensorThings API along with their implementations, one should keep in mind that the more modern SensorThings API is in many ways tailored for the usage with IoT devices (hence its name) and therefore may have an advantage when contrasting it with SOS in terms of IoT adequacy. However, both standards can be applied effectively in an IoT context, and therefore a comparison does highlight the key differences, along with strengths and weaknesses.

52North started developing its SOS implementation in 2010. Over the past nine years, technology has advanced considerably, and 52North has been adding features regularly to stay up-to-date. This becomes evident in the multitude of encoding formats and bindings, the extended operations with added functionality and the extent of configuration options. These extra features enhance the accessibility and configurability of SOS in several ways, contributing to the interoperability of the standard.

A significant addition to 52North-SOS is the support for JSON encoding format. JSON has emerged as a popular encoding format in recent years due to its simple syntax and easy serialisation to JavaScript variables, making it more compatible with modern web applications. Further arguments that favour the usage of JSON versus XML are that it has little overhead and that less bandwidth is required to transmit messages [28]. 52North-SOS supports JSON format for all the SOS core and transactional operations and most of the enhanced SOS operations, but XML messages are still encoded inside the JSON.

All encoding types for 52North-SOS operations including the JSON version are HTTP POST requests. The objects posted to the SOS server for transactional operations have different sizes but are generally larger than data insertions into FROST-SensorThings API. Requesting observations also requires posting an object containing the query parameters, while in FROST-SensorThings API this can be done with a GET request to the target URL using the API's query extensions. Request output is also more abundant in SOS for the most part than it is in SensorThings API GET requests, leading to general inflation of data traffic. A list of the different operations, their request type input and output sizes are shown in Table I.

	52North-SOS			**FROST**		
Operation	**Request type**	**Size In**	**Size Out**	**Request type**	**Size In**	**Size Out**
Insert single device	POST	4829	202	POST	4731	223
Request device information	POST	671	15986	GET	-	10103
Insert single observation	POST	1168	88	POST	105	65
Request single observation	POST	105	1361	GET	-	616
Delete device	POST	105	230	DELETE	-	230
Update single device property	POST	5733	226	PATCH	41	148

TABLE I. 52North-SOS and FROST-SensorThings request types and approximated input and output object sizes in bytes for *SEnviro*.

52North-SOS data can be accessed, visualised and maintained using an extensive client interface. Particularly the integration of the Helgoland client gives users an effective, easy-to-use and resource efficient data visualisation tool, saving users some effort in developing their interfaces. However, *SEnviro* already has its stand-alone web interface for data visualisation and analysis, therefore the Helgoland Client is not necessary. Moreover, the 52North-SOS service interface may confuse some users. The interface has so many options, settings and features that non-expert users may easily be overwhelmed by the vastness of its possibilities. SWE standards like SOS are as complex as needed, aiming to include support tasks ranging from the management of in-situ stations to the control of satellites. 52North-SOS is designed to support a vast spectrum of tasks, many of which are not necessary for IoT environmental monitoring applications like *SEnviro*, where devices run with limited resources.

In terms of performance 52North-SOS *getObservation* requests performs better than SensorThings API GET requests. This came as a surprise since the data traffic in SOS is heavier given the number of objects transferred and the object's sizes (see Table I). Furthermore, 52North's Helgoland client uses an API that improves response speed even further by reducing output information.

The SensorThings API (and FROST) surpasses 52North-SOS by a large margin in terms of flexibility and scalability. While 52North-SOS transactional operations can be configured to some extent, the input data must follow strict formats and semantics in order for requests to be successful. Once inserted, many of the service properties are difficult or impossible to update. In contrast, the SensorThings API data model and API makes inserting data extremely flexible [29]. Since multiple types of related entities can be inserted within the same request, clients can construct the JSON objects to be inserted in a "building block" fashion, assembling related entities as a single object. Additionally, the entities can be inserted separately and consequently linked with HTTP PATCH requests. This also allows any property of any entity within the service (apart from the unique ID) to be updated in an easy, developer-friendly way.

SensorThings API also provides a multitude of scaling possibilities for data output. 52North-SOS queries can be configured using up to five query parameters (e.g., spatial, temporal, properties) to narrow down output observations, which always include the full observation object and cannot be modified. SensorThings API queries can consist of an almost unlimited amount of query operators, which can be used to query any property of any entity within the service. The operators can be chained within the target URL, so no object needs to be posted to the server.

The fact that FROST does not include a user interface is irrelevant in a project like *SEnviro*, since a custom client interface has already been developed. Furthermore, Fraunhofer IOSB is developing FROST-Client and FROST-Dashboard client applications that provide user interfaces connecting to FROST-Server. Nonetheless, the SensorThings API's flexibility makes it easy to connect custom client interfaces to the service. SensorThings' usage of frequently used data standards like ISO8601 for dates and GeoJSON for spatial data combined with JSON encoding format facilitates spatial web development in countless ways, demonstrating the Web Standard's superiority over other standards and ensuring its interoperability.

In terms of response times, FROST-SensorThings API did not excel 52North-SOS, instead showing longer response times for getting observations with the minimum number of query operators. When applying query filters to reduce the size of data requested the response times are reduced to similar response times as SOS *getObservation* requests.

The results from the CPU and RAM usage show 52North-SOS as being less demanding than FROST-SensorThings API. This did come as a surprise since the FROST-SensorThings API functionalities are more focused on the essential data exposure using the REST API and does not include the array of different features 52North-SOS has (e.g., binding and encoding formats, client interface, configurable settings etc.). To fully understand the reason for 52North-SOS' superior performance, further investigations are necessary, analysing the core mechanisms of the services in detail. It was eventually concluded that 52North-SOS has been developed for a longer period of time than FROST-SensorThings API and therefore performance was optimised.

Unmodified response sizes from observation requests are smaller in FROST-SensorThings API. This promises reduced data traffic when requesting observations, for example in web applications for visualising large amounts of observations. As mentioned above, to make a tangible quantitative comparison requests were limited to 1,000 observations within the performance analysis. Attempts to retrieve larger amounts of observations were made though. On the one hand, 52North-SOS would frequently freeze or crash when requested amounts of data were too large. FROST-SensorThings API, on the other hand, can handle any amount of requested observations due to the approach of limiting file output but providing web links to the still pending data.

FROST-SensorThings API and 52North-SOS CPU and memory usage do not increase proportionally to the number of observation requests. However, this may change when requesting larger data sets.

Although the performance analysis does not favour FROST-SensorThings API over 52North-SOS, it was nonetheless concluded that FROST's advantages in the qualitative analysis sufficiently outweigh 52North-SOS. Finally, it is important to note that the performance results do not only derive from the web standards themselves, but rather from their implementations. When looking past FROST-SensorThings API and 52North-SOS at the mere web standards, the SensorThings API's data model and operations comply more fully with the IoT concept and with the requirements of current environmental monitoring and Smart Farming applications.

VIII. Conclusions

Investigating SOS it was concluded to be outdated many of aspects, lacking support for modern web technology trends, such as the use of JSON, RESTful binding and MQTT [30]. 52North-SOS has compensated many of these shortages with a multitude of features and has the capabilities to integrate with a large range of device types and can be applied to a wide spectrum of use cases. However, many of these features are workarounds for the outdated SOS data model and operations [29]. What 52North-SOS adds in functionality it lacks in flexibility and scalability, which has a strong impact on developer experience. Furthermore, the myriad of configurations and settings in the client interface render the software overwhelming. Since *SEnviro* has it is own client interface the extensive front-end features of SOS are not relevant to *SEnviro*, but may be useful in projects in need of an effective data visualisation tool such as the 52North-SOS Helgoland Client.

The SensorThings API proves to be an excellent choice for interoperability enhancement for *SEnviro* and environmental monitoring applications in the IoT field. FROST-SensorThings API implements the complete SensorThings API data model and functionalities of the standard as a back-end server instance, making it suitable for the integration into *SEnviro*. The API is flexible, scalable and follows modern web development trends. It focuses on the essential functionalities required in an IoT environment. Interoperability is guaranteed by using up-to-date technologies. Data is stored in compact JSON encoding and can be easily inserted, updated and removed via HTTP requests. Stored entities can be accessed by HTTP GET requests and data output can be customised by large variety of query parameters. A good developer experience is ensured by making the service flexible and scalable. While showing higher resource consumption and response times than 52North-SOS, performance issues can easily be overcome by using URL extensions to select only the required data, maintaining a low overhead.

One of the main limitations of the work can be seen in the selection of the chosen web standard implementations. Although a semantic comparison should not vary, since it is based on the regulations of the standard itself, the performance obtained in terms of time and CPU/memory resources of each option can be strongly linked to the developed software itself, therefore producing very different results.

An important point to note is that both implementations have been deployed following the default configuration parameters. For example, in FROST, only primary and foreign keys have indexes on them. It implies that if the database grows, a significant decrease in performance may occur. So it is advisable to identify which queries are used most frequently and add the appropriate indexes. In our study (SEnviro's use case), a large number of observations are not stored. It consisted of five nodes and operated during a vine season (4 months). In total, the experiments were realised using 197,887 observations. In case of expanding the number of vine seasons, we would apply these optimisations as future work.

This study not only answered questions about the researched topic, but also revealed some further issues and possible future work. Firstly, more web standards are bound to be released in the coming years and should be investigated. The SensorThings API shows great potential as it stands, but also needs to be further tested, especially considering the OGC's release of the SensorThings API's Tasking capabilities in January 2019.

A feature that strongly favours SensorThings API over other standards is its data publish/subscribe support via MQTT, avoiding larger data traffic via the HTTP protocol. FROST includes this service, making it possible for devices to publish directly to the FROST-server. This feature was not experimented on in this research because of the configuration of *SEnviro* Nodes and *SEnviro* Connect. Including the FROST MQTT support would include modifying the existing *SEnviro* architecture, which is outside the scope of this project but should be considered in future work.

Finally, another experimentation would be to carry out this approach from the Things' side. For this, a central server would not be used, and each Thing would have the ability to execute operations autonomously. The services would be embedded in each device. New implementations of the standards should be developed capable of being performed in environments with restrictive features in terms of computational cost, memory, connectivity and energy.

References

[1] J. Gubbi, R. Buyya, S. Marusic, *et al.*, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, 2013.

[2] R. Sutaria and R. Govindachari, "Making sense of Interoperability: Protocols and Standardization Initiatives in IoT," *The 2nd ComNeT-IoT workshop in the 14th International Conference on Distributed Computing and Networking (ICDCN 2013)*, 2013.

[3] A. Bassi and G. Horn, "Internet of Things in 2020: A Roadmap for the Future," *European Commission: Information Society and Media*, vol. 22, pp. 97–114, 2008.

[4] R. Alur, E. Berger, A. W. Drobnis, *et al.*, "Systems computing challenges in the internet of things," *arXiv preprint arXiv:1604.02980*, 2016.

[5] M. T. Lazarescu, "Design of a WSN Platform for Long-Term Environmental Monitoring for IoT Applications," *IEEE Journal on Emerging and Selected topics in Circuits and Systems, Vol. 3, NO. 1*, 2013.

[6] IEEE, "IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries," *IEEE Std 610*, pp. 1–217, Jan. 1991.

[7] B. Weinberg, "The Internet of Things and Open Source (Extended Abstract)," in *Interoperability and Open-Source Solutions for the Internet of Things*, ser. SoftCOM 2014, 2014.

[8] C. Granell, A. Kamilaris, A. Kotsev, *et al.*, "Internet of things," in *Manual of Digital Earth*, Springer, 2020, pp. 387–423.

[9] A. Kotsev, K. Schleidt, S. Liang, *et al.*, "Extending inspire to the internet of things through sensorthings api," *Geosciences*, vol. 8, no. 6, p. 221, 2018.

[10] S. Trilles, O. Belmonte, L. Diaz, *et al.*, "Mobile access to sensor networks by using gis standards and restful services," *IEEE Sensors Journal*, vol. 14, no. 12, pp. 4143–4153, 2014.

[11] S. T. Oliver, A. González-Pérez, and J. H. Guijarro, "An iot proposal for monitoring vineyards called senviro for agriculture," in *Proceedings of the 8th International Conference on the Internet of Things*, ser. IOT '18, Santa Barbara, California: ACM, 2018, 20:1–20:4.

[12] S. Trilles, A. González-Pérez, and J. Huerta, "An iot platform based on microservices and serverless paradigms for smart farming purposes," *Sensors*, vol. 20, no. 8, p. 2418, 2020.

[13] C.-Y. Huang and C.-H. Wu, "A Web Service Protocol Realizing Interoperable Internet of Things Tasking Capability," *Sensors*, 2016.

[14] C. Reed, "Data integration and interoperability: Iso/ogc standards for geo-information," *Development directions*, Feb. 2004.

[15] J. Fredericks and M. Botts, "Promoting the capture of sensor data provenance: a role-based approach to enable data quality assessment, sensor management and interoperability," *Open Geospatial Data, Software and Standards*, vol. 3, no. 1, 2018.

[16] Open Geospatial Consortium, *opengeospatial.org*, http://www.opengeospatial.org, Accessed: 2019-01-14, 2019.

[17] ——, *Sensor Observation Service*, https://www.opengeospatial.org/standards/sos, Accessed: 2019-01-14, 2007.

[18] ——, *OGC SensorThings API - Sensing*, http://www.opengeospatial.org/standards/sensorthings, Accessed: 2019-01-14, 2016.

[19] J. Pradilla, M. Esteve, and C. Palau, "SOSLite: Lightweight Sensor Observation Service (SOS)," *IEEE LATIN AMERICA TRANSACTIONS*, vol. 13, 2015.

[20] Open Geospatial Consortium, *Best practice for sensor web enablement lightweight sos profile for stationary in-situ sensors*, Open Geospatial Consortium, 2014.

[21] J. Pradilla, M. Esteve, and C. Palau, "SOSFul: Sensor Observation Service (SOS) for Internet of Things (IoT)," *IEEE LATIN AMERICA TRANSACTIONS*, vol. 16, 2018.

[22] A. Samourkasidis and I. N. Athanasiadis, "A sensor observation service extension for internet of things," in *International Workshop on Interoperability and Open-Source Solutions*, Springer, 2016, pp. 56–71.

[23] S. Trilles, J. Torres-Sospedra, Ó. Belmonte, *et al.*, "Development of an open sensorized platform in a smart agriculture context: A vineyard support system for monitoring mildew disease," *Sustainable Computing: Informatics and Systems*, 2019.

[24] A. Kamilaris, F. Gaoy, F. X. Prenafeta-Boldú, *et al.*, "Agri-IoT: A Semantic Framework for Internet of Things-enabled Smart Farming Applications," *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 2016.

[25] S. Trilles, A. Lujan, O. Belmonte, *et al.*, "Senviro: A sensorized platform proposal using open hardware and open standards," *Sensors*, vol. 15, no. 3, pp. 5555–5582, 2015.

[26] M. A. da Cruz, J. J. Rodrigues, A. K. Sangaiah, *et al.*, "Performance evaluation of iot middleware," *Journal of Network and Computer Applications*, vol. 109, pp. 53–65, 2018.

[27] J. d. C. Silva, P. H. Pereira, L. L. de Souza, *et al.*, "Performance evaluation of iot network management platforms," in *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, IEEE, 2018, pp. 259–265.

[28] A. Tamayo, C. Granell, and J. Huerta, "Using SWE Standards for Ubiquitous Environmental Sensing: A Performance Analysis," *Sensors 2012*, 2012.

[29] J. A. B. S. Teixeira, "Using sensorthings api to enable a multi-platform iot environment," 2018.

[30] H. Van Der Schaaf, J. Moßgraber, S. Grellet, *et al.*, "An environmental sensor data suite using the ogc sensorthings api," in *International Symposium on Environmental Software Systems*, Springer, 2020, pp. 228–241.

Daniel Marsh-Hunn currently works as a spatial web developer at geOps - Open Source Spatial Web. He received an Erasmus Mundus Scholarship for the Erasmus Mundus Master in Geospatial Technologies at Jaume I University, from which he graduated in March 2019. Moreover he holds a BSc. in Environmental Systems Sciences with emphasis on geography. His main interests lie in all things spatial, be it GIS, spatial web or geo-enabled IoT applications.

Sergio Trilles has a PhD in Integration of Geospatial Information from the Jaume I University in 2015 and he is currently a post-doctoral fellow at University Jaime I, holding a Juan de la Cierva-Incorporación fellowship. His research lines are centre on geospatial fields such as the Internet of Things (sensors), interoperability, geoprocessing or web mapping. He is author of more than fifty journal and conference peer-reviewed publications.

Alberto González is currently pursuing his Ph.D. in Computer Science thanks to an FPU grant from the Spanish Ministry of Science, Innovation and Universities. Previously, he received his master's Degree in Intelligent Systems in 2018 and a bachelor's degree in Computer Science in 2017. He has experience developing web applications, mobile applications and web services, also working with embedded hardware. He aims to expand his developer toolbox adding data scientist skills during his Ph.D. studies, concretely geospatial analysis skills.

Joaquín Torres-Sospedra received his PhD about Ensembles of Neural Networks and Machine Learning from Universitat Jaume in 2011 I. Since January 2020 he is the Scientific Coordinator of UBIK Geospatial Solutions. He has authored more than 120 articles in journals and conferences. His current research interests include indoor positioning solutions based on Wi-Fi & BLE, Machine Learning and Evaluation. He is the chair of the IPIN International Standards Committee and IPIN off-site Competition.

Francisco Ramos is associate professor in the Department of Computer Languages and Systems at UJI (Spain). He got his Ph.D. with honours from this University in 2008. His research interests are in the areas of mobile interaction, computer graphics, visualization and GIS. He is CTO and co-founder of Emotional Apps, an enterprise which combines technology and emotions by means of creating innovative mobile apps.