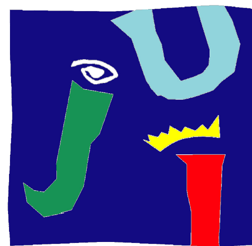
A thesis submitted for the Master of Science degree in Geospatial Technologies

Interoperability Enhancement of IoT devices Using Open Web Standards in a Smart Farming Use Case

Composed by Daniel Marsh-Hunn

Supervised by Ph.D Sergio Trilles

Co-supervised by Prof. Joaquín Huerta & Prof. Roberto Henriques



Universitat Jaume I

04.02.2019

Abstract

Since its first appearance the Internet of Things has been subject to constant evolution, development and change. Now it has stepped out of its infancy with billions of devices embedded in the world wide web. However, IoT providers mostly define their own data formats and protocols and there is still a lack of a common standard that connects these devices in an interoperable manner. There are several organisations dedicated to developing common standards for IoT devices and research is focusing on defining an effective standard to be used by embedded devices. Unsurprisingly, IoT has also found its way into the spatial web and into 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 three open Web Standards in the use case of *SEnviro for Agriculture*, a full stack IoT for monitoring vineyards. The interoperability potential of the OGC's *Sensor Observation Service* and *SensorThings API* are evaluated by integrating Web Standard implementations for each standard and contrasting their qualitative and quantitative traits. In a further step the Mozilla Corporation's *Web Thing API* was implemented and evaluated in an environmental monitoring and Smart Farming context. The results of the study show that the SensorThings API proves to be the most adequate Web Standard for SEnviro and IoT applications for environmental monitoring and Smart Farming in terms of interoperability. It outperforms the contesting Web Standards in terms of flexibility and scalability, which strongly impacts on developer and user experience.

Acknowledgements

I would like express my gratitude to all the people involved in the process of this study. First I would like to thank my supervisor Ph.D. Sergi Trilles for his support. Sergi's advice, feedback, knowledge and dedication proved to be crucial for the success of this thesis. I also want to thank Alberto Gonzales for taking the time to aid me in technical challenges. My thanks also go to my co-supervisors Prof. Joaquin Huerta and Prof. Roberto Henriques for providing valuable feedback. I further wish to express my appreciation for all dear friends who aided me with their advice, their motivation and their good spirit. My special thanks go to my fellow students, with whom I shared this precious life experience. Lastly I want to thank my dear life partner Franziska for her encouragement, her patience, her council and her love.

Contents

1. Introduction **1**
1.1. Context 1
1.2. Problem Definition 3
1.3. Motivation and goals 3
1.4. Structure 4

2. Background **5**
2.1. Internet of Things 5
2.2. Sensing hardware 6
2.3. Wireless Sensor Networks 7
2.4. Machine-to-Machine Communication & MQTT 9
2.5. Interoperability 10
2.6. Open & Sensor Web Standards 11
2.6.1. Sensor Observation Service 12
2.6.1.1. SOS data model 12
2.6.1.2. SOS operations 14
2.6.1.3. SOS implementations 14
2.6.2. SensorThings API 15
2.6.2.1. SensorThings data model 15
2.6.2.2. SensorThings operations 17
2.6.2.3. SensorThings implementations 17
2.6.3. Web Thing API 18
2.6.3.1. Web Thing data model 19

3. SEnviro for Agriculture **20**
3.1. Smart Farming 20
3.2. SEnviro components 22
3.2.1. SEnviro nodes 22
3.2.2. SEnviro Connect 24

4. Methodology **28**
4.1. Experimental Environment 29
4.2. Deployed Web Standard Implementations 30
4.2.1. 52North-SOS 30
4.2.2. FROST-Server 33

4.3. SEnviro Web Standard Integration 33
4.3.1. SOS Adapter 35
4.3.2. SensorThings Adapter 35
4.3.3. Web Thing Adapter 36
4.4. Comparative Analysis 37
4.4.1. Performed operations 37
4.4.1.1. Data management 37
4.4.1.2. Accessing metadata 37
4.4.1.3. Observation retrieval 38
4.4.2. Qualitative Analysis 40
4.4.3. Quantitative Analysis 41
4.4.3.1. Response times and sizes 41
4.4.3.2. Web service metrics 42

5. Web Standard Comparison 45
5.1. Web Thing API evaluation 45
5.2. 52North-SOS & FROST qualitative evaluation 47
5.2.1. Service deployment & configuration 47
5.2.1.1. 52NorthSOS setup 47
5.2.1.2. FROST setup 48
5.2.2. Data management 48
5.2.2.1. 52North-SOS transactional operations 48
5.2.2.2. FROST *Create-Update-Delete* requests 53
5.2.3. Retrieving metadata 55
5.2.3.1. 52North-SOS metadata operations 56
5.2.3.2. FROST metadata queries 56
5.2.4. Observation queries 58
5.2.4.1. 52North-SOS queries 58
5.2.4.2. FROST queries 60
5.3. 52North-SOS & FROST performance evaluation 64
5.3.1. Response times and sizes 65
5.3.2. Container metrics 68
5.4. Discussion 72

Conclusion & Future Work 78

Bibliography 82

Appendix 93

List of Figures

2.1. WSN components overview (Mainetti et al., 2011) 8
2.2. O&M data model extract (Open Geospatial Consortium, 2007) 13
2.3. SensorThings data model (Open Geospatial Consortium, 2016) 16
2.4. Mozilla - Project Things 18

3.1. Schema of SEnviro architecture; the pink section represents physical components, the blue section represents software elements 22
3.2. SEnviro node core and power supply assembly 23
3.3. Fully assembled and deployed SEnviro node. 24
3.4. A screen-shot of the SEnviro Connect view to visualise observations and alerts. 26
3.5. A screen-shot of the SEnviro Connect view to manage SEnviro nodes. 27

4.1. Schematic view of Web Standards SEnviro integration 29
4.2. 52North SOS Client interface 30
4.3. Helgoland Map View 31
4.4. Helgoland Diagram View 32
4.5. SEnviro message queuing example schema 34
4.6. Postman Response Time Monitoring Interface 42
4.7. JMeter Performance Monitoring interface 43
4.8. cAdvisor interface 44

5.1. Elcano specifications 64
5.2. Average Postman response sizes 66
5.3. Average Postman response times 67
5.4. Above: Plots from 52North-SOS and FROST container CPU usage in idle state and during requests of 1000 observations; Below: CPU metric behaviour from both services during requests of 1000 observations 69
5.5. Average CPU usage for different response sizes in FROST and 52North-SOS 70
5.6. Memory metrics for 52North-SOS and FROST requests of 1000 observations 71
5.7. Memory usage for different request sizes in FROST and 52North-SOS 71

List of Tables

4.1. FROST OGC Compliance Testing Status 33

5.1. Built-in operators for the SensorThings Filter operator 61
5.2. Built-in functions for the SensorThings Filter operator 62
5.3. Average HTTP response sizes in Kilobytes [kb] for observation retrieval from Postman 65
5.4. Average HTTP Response Times in Milliseconds [ms] for observation retrieval from Postman 67
5.5. 52North-SOS and FROST request types and approximated input and output object sizes in bytes for SEnviro 74

1. Introduction

The Internet revolution enabled the 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*. The concept of the Internet of Things (IoT) first emerged in 1999 and has since been subject to constant evolution, redefinition and expansion. It has stepped out of its infancy and is transforming the current Internet into fully integrated future Internet, connecting billions independent, intelligent devices (Gubbi et al., 2013).

1.1. Context

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 disparate solutions and heterogeneous data sources and formats interact seamlessly, enabling a large-scale IoT (Sutaria and Govindachari, 2013). Widely defined as "*a world-wide network of interconnected uniquely addressable objects, based on standard communication protocols*" (Bassi and Horn, 2008), predictions estimate the IoT will consist of 50 billion connected devices exchanging information over the Internet by 2020 with an economic impact of 2 trillion US$ (Weinberg, 2014). IoT applications are developing in major sectors such as smart business, inventory management, smart homes, transportation and logistics, health-care, security and surveillance and environmental monitoring. This vast number of "things" is able to access and acquire data about devices and their environment, independent of human interaction (Lazarescu, 2013).

IoT environmental and earth monitoring applications have received increased attention in the recent decades, since they have become a key factor of sustainable growth world-wide. Among outdoor environment monitoring, observing open nature phenomena can be challenging due to harsh climatic conditions and difficulty of physical access to the field, resulting in high costs for sensor deployment and maintenance (Lazarescu, 2013). These challenges have been addressed by technological advances in low power integrated circuits and wireless communication. Modern sensing devices have drastically decreased in size, in cost and in power consumption, resulting in the viability of deploying networks of intelligent sensors. These Wireless Sensor Networks (WSN) may consist of a large number of nodes with lim-

ited processing capability and storage, and can be equipped with several different sensors, capable of observing multiple natural phenomena (Gubbi et al., 2013).

The advance in technology for environmental monitoring has also extended into agriculture and farming (Kamilaris et al., 2016). Highly accurate embedded environmental sensors have paved the way for *precision agriculture*, which applies more efficient irrigation, targeted use of fertilisers and pesticides and more precise use of fodder and antibiotics for livestock. This enhanced form of farming potentially leads increased productivity, greater yields and a reduced environmental footprint. Taking precision farming even further in an IoT context, the concept of *Smart Farming* has emerged for decision-support for farmers. Smart Farming focuses on real-time data gathering, processing and analysis and on the automation of farming procedures for an overall improvement of farming operations and management (Kamilaris et al., 2016).

Modern web technology is advancing at a tremendous rate and demand for research in specialised fields is increasing in order to stay up to date with state-of-the-art technology. Web-based solutions for environmental and agricultural monitoring are no exception and researchers are designing and developing new implementations to enhance precision and efficiency of monitoring systems. A challenge, in the IoT domain in general as well as in specialised domains, is to achieve 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.*" (IEEE, 1991), this is a key concept to make environmental monitoring information accessible to a larger community and is a necessary step to take towards open data. Since IoT emerged a large variety of vendors, researchers and interested parties have been developing IoT solutions in parallel and although several device types and protocols for IoT are available on the market, only few interoperate among each other (Weinberg, 2014). With the growing number of IoT devices the importance of interoperability is becoming more evident. While the world-wide Internet relies on standard technologies and protocols like HTTP, SSH etc., these solutions are not designed for devices with severe power and data loss constraints. Efforts are currently being made to produce well defined open standards-backed protocols compatible with IoT environments (Sutaria and Govindachari, 2013).

1.2. Problem Definition

Although the concept of IoT is already established and implementations are mushrooming manifold, it still lacks a widely accepted standard model to enable large 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. Currently there are several proposed open standard solutions already available, each differing in slightly in functionality, specialisation and structure, but having interoperability as a primary goal. With IoT also having found its way into environmental monitoring, Smart Farming and the Spatial Web, it is crucial to investigate the potential of existing Web Standards in an interoperability context within these domain. This includes testing and comparing different standards in order to find the most efficient Web Standard solution. This study will investigate openly available Web Standards to answer the following research questions:

"How can Open Web Standards increase interoperability of Smart Farming IoT solutions?"

"Which Web Standard is the most adequate to enhance interoperability in Smart Farming IoT solutions?"

1.3. Motivation and goals

In 2015 a prototype for a sensorised platform WSN using open hardware and open standards was developed in the scope of the *SEnviro* project (Trilles et al., 2015) at Universitat Jaume I in Castellón de la Plana, Spain. Three years later *SEnviro for Agriculture* (Trilles et al., 2018) was launched, furthering the development of a new prototype based on the SEnviro platform in an agricultural context to monitor environmental parameters in vineyards. While it is planned to include open standards into the project, this task is still pending and will be the focus in this master thesis.

This master thesis aims at investigating potential open Web Standard solutions for environmental monitoring applications in the IoT. The main contribution of this study lies in producing an in-depth, contrasting juxtaposition of a selection of openly available, up-to-date web standard solutions. Open source implementations of these standards are deployed and adapted to the needs of SEnviro for Agriculture, then they are compared on performance, on semantic, on flexibility and on scalability levels. The results help determine the most suitable candidate for SEnviro for Agriculture and potentially for other environmental monitoring and agricultural

IoT applications. A further goal of the study is to enhance the interoperability of the already established *SEnviro for Agriculture* web platform.

1.4. Structure

The remainder of this document consists of the following chapters:

2. **Background**: This chapter contains a full literature review covering topics relevant to this paper. The chapter will provide insight into previous and state-of- the-art research in WSN, open standards, communication protocols and interoperability.

3. **SEnviro for Agriculture**: The document continues with a detailed summary of the SEnviro for Agriculture project. This includes information about the area and context of experimentation, the included hardware and software architecture and the applied web technologies.

4. **Methodology**: The Methodology section sheds light on the methodology used to evaluate the potential of applying open Web Standards in SEnviro. It focuses first on the selection of open-source Web Standard implementations, followed up by their deployment in the experimental environment. The chapter then describes the methods of comparing the deployed instances used in the study.

5. **Web Standard Comparison**: This chapter presents the results of the comparing methods. The deployed open standard entities are compared on multiple levels. Performance is assessed quantitatively, Web Standard semantics will be contrasted and aptitude for SEnviro and Smart Farming projects is determined.

The terminal section **Conclusion & Future Work** provides a summary of the project results and specifies on future work in this field of study.

2. Background

This chapter aims to provide an overview of the topics included in this research, investigates the key concepts and explores work relevant to this study. Furthermore, it will focus on several technologies crucial to the realisation of this research. While many of these technologies can be applied in several application fields, this document will primarily refer to their application in an environmental and agricultural monitoring context, since the study was applied to a Smart Farming use case.

Like several other application fields, environmental and agricultural monitoring have evolved with the technological revolution of the recent decades. While in the past environmental monitoring depended on the manual collection of field data and laboratory analyses, smarter, smaller and inexpensive sensors and wireless communication technology now enable continuous monitoring and real-time applications.

The following sections provide insight into state-of-the-art technologies and key concepts included in this study's work.

2.1. Internet of Things

The term *Internet of Things* was first introduced in supply chain management by Kevin Ashton in 1999, but in the past two decades the definition evolved to include a wide range of fields like health care, utilities, transport etc. (Gubbi et al., 2013). Typically, it is described as a framework of interconnected, uniquely identifiable devices within the Internet. In the IoT concept every physical thing is reflected in the information world by its unique virtual identity, enabling it to interconnect and communicate with each other (Huang and Wu, 2016).

Unique identifiers are key to any kind of communication and transaction among devices, which in IoT environments are represented by IP addresses in a global network of devices. Furthermore, Uniform Resource Identifiers (URI) are considered as the primary identifiers in the Physical Web, which consists of devices within the IoT which are directly accessed and managed by web technologies. Key enabling technologies for communication among IoT devices are Radio Frequency Identification (RFID), Bluetooth Low-Energy beacons (BLE) and optical tags such as quick-response codes (QR-Codes). These technologies enable power-efficient and remote device identification (Want et al., 2015).

The two main capabilities of the IoT are Sensing and Tasking. The sensing capability enables users to monitor device's statuses (e.g. On-Off status, battery status, etc.) and environmental properties of their surroundings (e.g. temperature, humidity, location etc.). The tasking capability enables users and other devices to control devices remotely, making the device execute tasks or adjusting its properties. The integration of these capabilities forms the base of various automatic and efficient IoT applications (Huang and Wu, 2016).

2.2. Sensing hardware

Sensor nodes in WSN consist of four essential components: one or multiple sensing units, a processor, a transceiver and a power supply (Trilles et al., 2015). Research in environmental sensing has furthered the development of these components, resulting in prolonged node operation time and WSN lifetime (Moiş et al., 2018).

Sensors are devices able to detect changes in the environment, measuring physical phenomena (e.g. temperature, humidity etc.) transforming them into electric signals (Spann et al., 2018). Rapid development in sensing technologies is one of the primary drivers for the IoT. Decreasing cost, higher accuracy, greater power efficiency and smaller size of sensing devices has led to a large variety of sensor solutions and smart devices for several application fields on the market (Swan, 2012). Since Smart Farming focuses widely on monitoring plant health, sensors typically include hardware for measuring soil and air temperature and humidity and optical sensing units for monitoring plant leaf colour and light intensity.

Environmental monitoring mostly requires humidity and temperature measurement, resulting in a high demand for affordable, easy-to-use integrated humidity and temperature sensors (Lee and Lee, 2005). Examples for low-cost air temperature and humidity sensors are the DHT11 and the DHT22, which are small in size (up to 15.5x25x7.7 mm), in many cases sufficiently accurate (humidity accuracy 2-5%, temperature accuracy up to ±0.5ºC) and costs under 10 US$ on the market.

For effective monitoring in the field, sensors are complemented by microcontroller boards. Microcontrollers contain all the necessary components which allow it to operate standalone, such as the microprocessor (CPU), memory, interface controllers, timers and one or multiple input-output pins (Gridling and Weiss, 2007). Several microcontroller boards are now available on the market, including specific features and functionalities depending on their field of usage. Among the most popular boards are the Arduino Uno and the Raspberry Pi 2 (Spann et al., 2018).

2.3. Wireless Sensor Networks

The advances in monitoring hardware go hand in hand with more effective ways for devices to connect and communicate among each other. While environmental monitoring was traditionally based on field sampling, on laboratory analysis or on off-line sensors and field loggers requiring manual data downloading, Web technology now enables real-time monitoring with sensor networks (Kotamäki et al., 2009).

Environmental monitoring forms the backbone of Smart Farming, and the bulk of monitoring applications rely on WSN due to their flexibility, scalability and reduced installation and maintenance costs (Moiş et al., 2018). Originally mostly based on closed or proprietary systems with limited communication to the outside world, modern WSNs use the Internet Protocol to connect WSN nodes to the IoT (Mainetti et al., 2011).

WSN include generic nodes and gateway nodes. Generic (multi-purpose) nodes are equipped with devices capable of taking measurements from the environment. Gateway nodes receive data from generic nodes and transmit them to a central station, which requires the nodes to have greater processing power and power supply. As shown in *Figure 2.1*, key to enabling WSN applications to connect to sensor technologies can be classified into three component groups: system, communication protocols and services (Yick et al., 2008).

Every node is a self-organising, self-managing system. WSN platforms are designed to support a a variety of sensors, radio components, processors and storage. The system software, such as the operating system, has to be designed to support the varying sensor platforms. Further core system features must include automatic management, longevity optimization and distributed programming. Organisations like the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) develop technology standards for IoT addressing these issues, with standards like IEEE 802.15.4, ZigBee, WirelessHART, ISA100.11, IETF 6LoWPAN, IEEE 802.15.3 and Wibree being among the most frequently used. These standards typically include dedicated communication protocols and aim at increasing IoT interoperability. WSN nodes use communication protocols to communicate with one another, which need to be energy-efficient and reliable in transporting data packages. Network services include node localisation (e.g. GPS, RIPS etc.), synchronisation, coverage and data compression and aggregation (Yick et al., 2008).

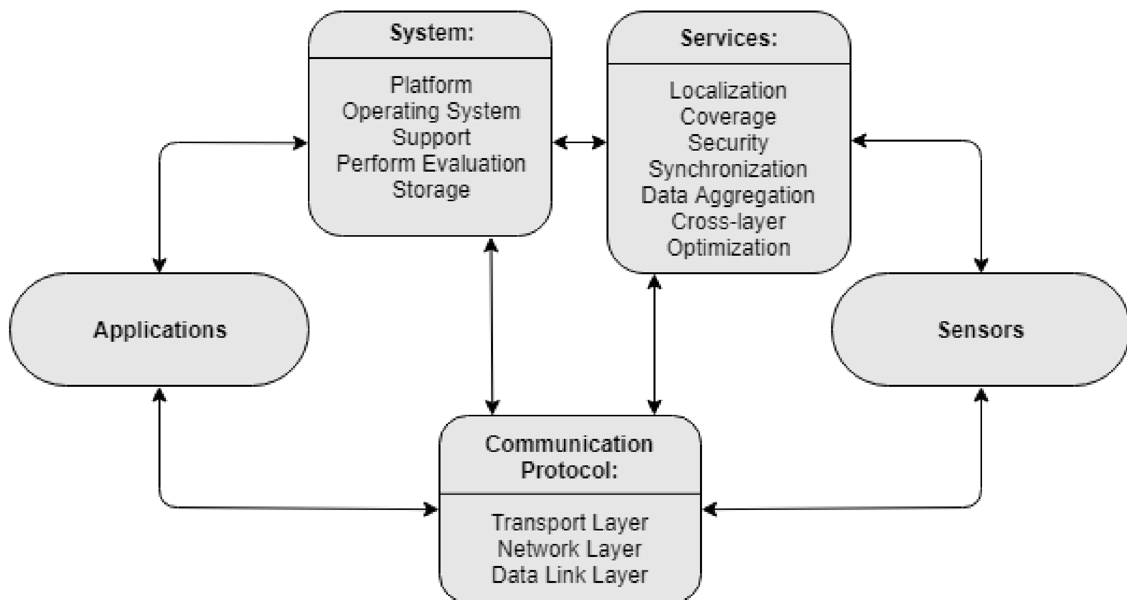


Figure 2.1.: WSN components overview (Mainetti et al., 2011)

2.4. Machine-to-Machine Communication & MQTT

Machine-to-Machine (M2M) communication is essential in the IoT, enabling intelligent devices communicate and make decisions without human interaction. M2M applications impose new limitations on communication solutions due to the type of data traffic generated. M2M data traffic typically transfers small data packages, but they are large in number and involve a many connected devices. Even though the messages are usually short, M2M services need to be highly reliable and able to handle delays. Further challenges are potential computing and power constraints, which depend on device hardware (Latvakoski et al., 2014).

Devices in the IoT are exposed on the World Wide web using the lingua franca, namely the Hyper Text Transfer Protocol (HTTP), and their accessibility typically builds on Representational State Transfer (REST) principles. While HTTP was designed for client-server communication and transferred data packages are generally large, embedded devices usually can't handle high-level protocols and require lightweight binary protocols and bidirectional communications (Collina et al., 2012).

The websocket protocol, standardised by the Internet Engineering Task Force (IETF) in 2011, allows bidirectional communications and several M2M messaging protocols are now available. Research in REST-like applications in constrained environments have produced protocols like the Constrained Application Protocol (CoAP), which enables resource constrained applications to use HTTP interaction methods (GET, POST, PUT, DELETE). Furthermore, standards like 6lowpan (IPv6 Low Power wireless Area Networks) can handle Maximum Transmission Unit (MTU) sizes and compress IPv6 headers from 60 bytes to 7 bytes. Message protocols can be *broker-based*, such as Data Distribution Service (DDS), Advanced Message Queuing Protocol (AMQP) and STOMP, or *broker-less*, such as ZeroMQ. Finally, the Message Queuing Telemetry Transport (MQTT) protocol enables M2M applications to exchange lightweight messages in publish/subscribe patterns (Latvakoski et al., 2014). MQTT messages are binary based, meaning the control elements are binary bytes instead of being text strings. Byte messages have considerably smaller payloads than e.g. JSON strings and therefore ensure messages remain lightweight.

The MQTT protocol implements various levels of Quality of Service and boasts high speed and power efficiency. It is based on the concept *topics* to which clients can publish updates or subscribe for receiving updates from other clients. With rapidly increasing popularity, MQTT libraries now support major IoT development platforms (e.g. Arduino), programming languages and for iOS and Android mobile platforms. The MQTT community claims that a publish/subscribe protocol is the

key component to build the future IoT (Collina et al., 2012).

2.5. Interoperability

The number of embedded devices within the IoT is increasing drastically and many IoT producers develop distinct web service protocols, only supported by their proprietary IoT devices. This results in closed *IoT silos*, each having its complete IoT frameworks, including devices, gateways, services and applications. An upcoming issue in IoT is that elements in different IoT silos cannot connect, leading to scattered IoT solutions with incompatible, co-existing protocols (Huang and Wu, 2016).

Interoperability in the rapidly growing IoT ecosystem is a challenging yet crucial aspect of successful IoT. Highly interoperable protocols must address several challenges. Firstly, the translation solution must not impose any pre-configuration of applications. Furthermore, it should be scalable, capable of running hundreds of connections within cloud environments. There are also security and verifiability requirements to uphold. Finally, the interoperability solution must be capable of verbose reporting and supporting Quality of Service parameters (Derhamy et al., 2017).

Approaches to address IoT interoperability have been made by numerous research projects in industry and academia. Solutions include the development of Middleware (e.g. Starlink, INDISS, uMiddle) and protocol proxies (e.g. MuleSoft, IBM MQ, Artix), yet they shift the interoperability dilemma rather than solving it. Currently, it is improbable that a single IoT communication protocol prevails as a globally accepted solution, yet there have been efforts to do so as proposed in Derhamy et al. (2017).

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 concensus 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 (Reed, 2004).

2.6. Open & Sensor Web Standards

Geo-scientists have been uploading geographical data for sharing and exploration since the dawn of the World Wide Web. M2M data harvesting has led to community-adopted frameworks, common standards and enriched metadata, significantly improving observational accuracy, sensor discovery and configurability (Fredericks and Botts, 2018). Sensor systems contribute to a large part of geospatial data and generally include in-situ sensors, moving sensor platforms or networks of static sensors (Open Geospatial Consortium, 2007).

The Open Geospatial Consortium (OGC) is one of the main actors in the field of Sensor Web Standards. The OGC Sensor Web Enablement (SWE) supports the ability to fully describe the processes used in producing observations and their corresponding sensors. Furthermore, it includes machine-actionable access to web-accessible observations and sensor tasking capabilities. SWE builds on machine-readable encoding like Sensor Model Language (SensorML), Observation & Measurement (O&M) and Geography Markup Language (GML)(Fredericks and Botts, 2018).

OGC Web Standards are a tempting choice for Smart Farming applications, since this organisation specialises in publicly available standards development to *geo-enable* the web. In the scope of the OGC's SWE, the organisation released a set of services to facilitate the exchange of sensor data between SWE enabled nodes and to allow clients and servers to arrange, encode and transfer sensor data in a semantically enabled way (Open Geospatial Consortium, 2019). This document will consider the well- established *Sensor Observation Service* (see *2.6.1 Sensor Observation Service* and the more recent *SensorThings API* (see *2.6.2 SensorThings API*).

Although the OGC offers the most complete open standards for sensor observation and boast most features and functionality, there are also non-OGC solutions for environmental monitoring currently available. Some more recent solutions address challenges more related to the IoT paradigm, designed for connecting devices to the World Wide Web in a light-weight and flexible, machine-readable format. This study provides a closer look into the *Web Thing API* (see *2.6.3 Web Thing API*), a proposed data model and API by the Mozilla Corporation and intends to investigate its potential in environmental monitoring applications like SEnviro.

2.6.1. Sensor Observation Service

The Sensor Observation Service (SOS) was approved by the OGC in 2007 as an official open standard for handling sensor data in the World Wide Web. SOS provides a standardised interface using SOAP XML binding for managing and retrieving metadata and observations from heterogeneous sensor systems. It is part of the OGC Sensor Web Enablement framework of standards and leverages the SWE data models and XML encodings for *Observation & Measurement* (O&M), *Transducer Model Language* (TransducerML) and *Sensor Model Language* (SensorML). The SOS development approach aims to describe sensing devices, sensor systems and observations in a way that supports all types of sensors and the requirements of users of sensor data. SOS is therefore designed to match the O&M data model (see *Figure 2.2*) (Open Geospatial Consortium, 2007).

2.6.1.1. SOS data model

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 and belong to a common service. For example, the relation can be spatial (share the same location or platform), temporal (created in the same time interval) or due to corresponding properties (e.g. measure the same phenomena)
- *ObservableProperty*: A procedure can have multiple observed properties, which represent the physical phenomena measured by a sensor (e.g. temperature, humidity etc.)
- *FeatureOfInterest*: Features of interest (FoI) represent identifiable objects on which sensor systems are making observations. These should 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.

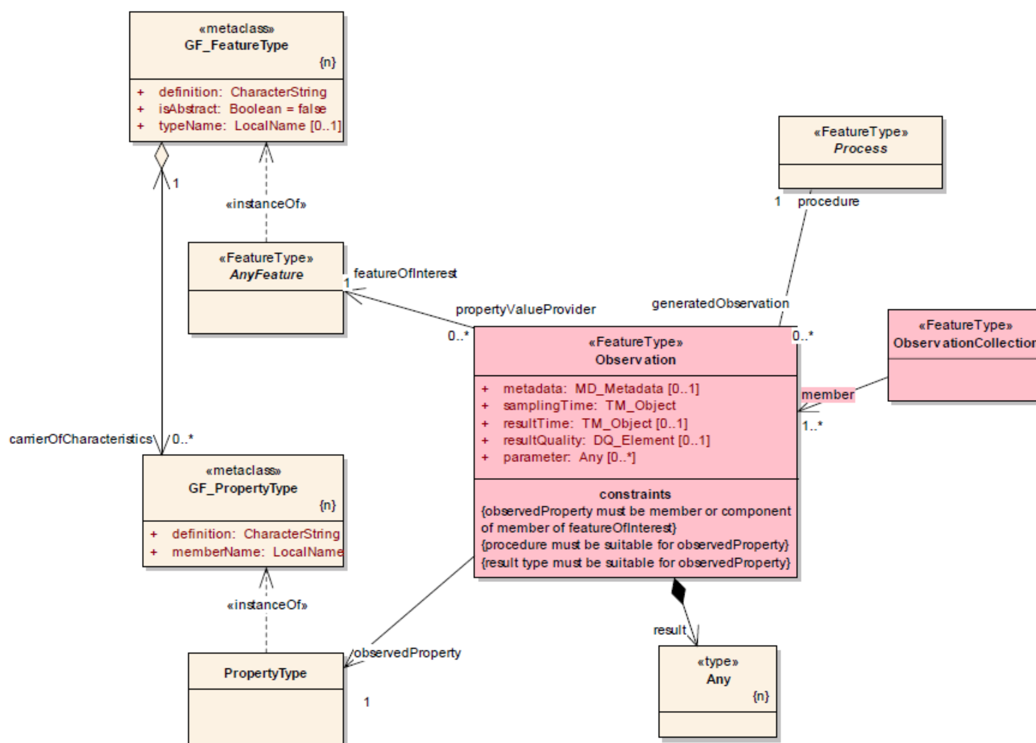


Figure 2.2.: O&M data model extract (Open Geospatial Consortium, 2007)

2.6.1.2. SOS operations

SOS includes a set of operations for retrieving sensor data 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 in SensorML format 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 type of 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 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.

2.6.1.3. SOS implementations

There are currently several open source implementations of SOS. Among the most developed, openly available SOS-implementations are:

- *istSOS*: SOS server implementation written in python, developed at the Insti-

tute for Earth Sciences at SUPSI[1] (University of Applied Sciences and Arts of Southern Switzerland). It a user-friendly interface for service configuration, a REST API and data exploration. The latest istSOS2 release (istSOS-2.3.1) was in February 2016. istSOS3 is in development, but still in its early stages.

- *52North-SOS*: This Java implementation of SOS is developed at 52North GMBH[2], based in Münster, Germany. 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 latest one (SOS 4.4.4) available since December 6, 2018.

2.6.2. SensorThings API

In 2016 the OGC approved the SensorThings API as an official Web Standard. It provides an open standard-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, light-weight services that cover a broader spectrum of applications. This lowers the risks, time and cost across IoT device life cycles and enhances compatibility in device-to-device and device-to-application connections (Open Geospatial Consortium, 2016).

2.6.2.1. SensorThings data model

The SensorThings API data model is based on the OGC Observation & Measurement model. It consists of a set of interrelated entities, depicted in *Figure 2.3*. In contrast to SOS, entities are encoded using JSON format.

Brief specifications of the entities as described in the SensorThings API manual in Open Geospatial Consortium (2016) 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*".

[1] http://www.supsi.ch/ist (Accessed 19.02.2019)
[2] https://52north.org (Accessed 19.02.2019)

- *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 Feature of Interest (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.

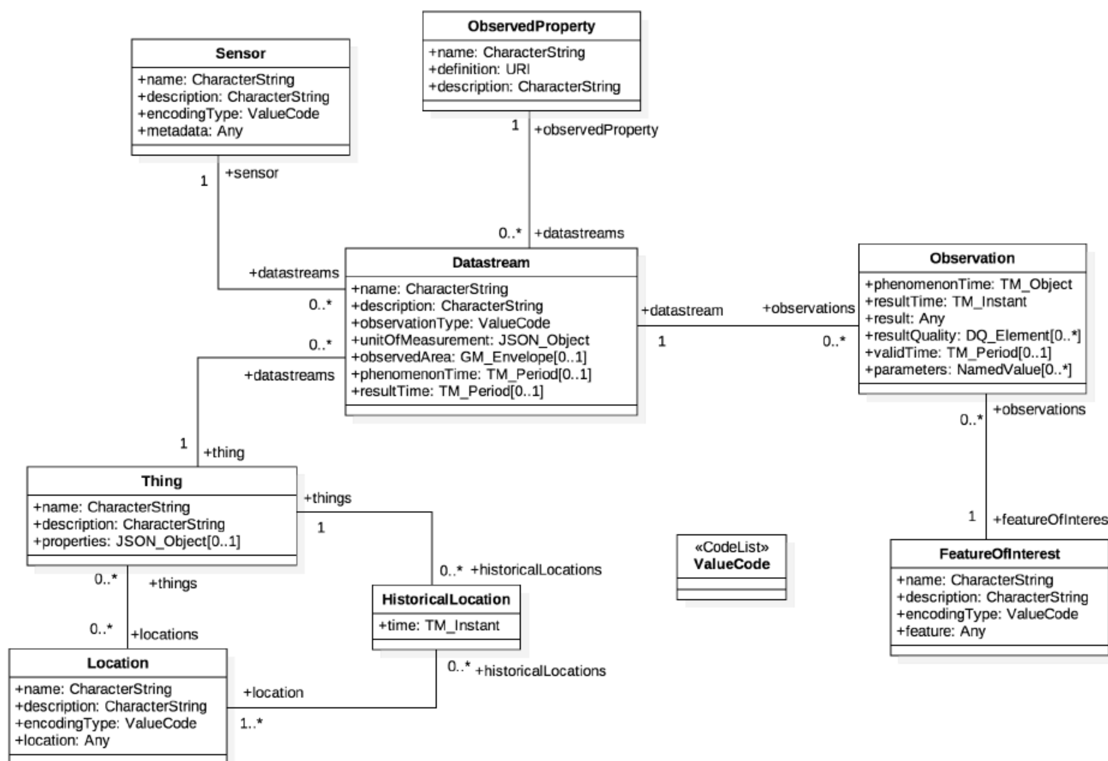


Figure 2.3.: SensorThings data model (Open Geospatial Consortium, 2016)

2.6.2.2. SensorThings operations

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 large set of query parameters to pinpoint the desired JSON objects, as in the example query URL below.

```
http://elcano.init.uji.es:8082/FROST-Server/v1.0/Things(2)/
Datastreams(12)/Observations?$select=phenomenonTime,result&$
filter=overlaps(phenomenonTime,2018-11-28T00:00:00.000Z/
2018-12-10T00:00:00.000Z)
```

The example URL presented above requests the observations produced in the Datastream with ID = 12, associated with the Thing with ID = 2. Furthermore, it will only consider observations between 2018-11-28 and 2018-12-10 and only select the *phenomenonTime* and *result* attributes for the output JSON.

2.6.2.3. SensorThings implementations

There are several server implementations of the SensorThings API available as open source software. Among the most established are the following:

- *GOST*: **Go** **S**ensor**T**hings was developed by Geodan [3] in Golang (Go) programming language. The implementation development is still ongoing, but it features the complete SensorThings Sensing functionality and includes a dashboard for data visualisation.
- *Mozilla-SensorThings*: The Mozilla Corporation started developing a Node implementation of SensorThings, but the development has been stalled since February 2017.
- *FROST*: The **Fr**aunhofer **O**pen-Source **S**ensor**T**hings (FROST) was developed by the Fraunhofer IOSB [4] (Institute of Optronics, System Technologies and Image Exploitation) in Java. FROST-developers are still working on the project to extend its features in accordance to the OGC SensorThings API

[3]www.geodan.com (Accessed 19.02.2019)
[4]www.iosb.fraunhofer.de (Accessed 19.02.2019)

Web Standard.

2.6.3. Web Thing API

In June 2017 the Mozilla Corporation revealed their activity in building the Web of Things. In February 2018, it announced the kick-off of *Project Things*, an experimental framework of software and services for connecting IoT devices to the web. Project Things consists of three core components: The *Things Gateway*, the *Things Cloud* and the *Things Framework*.

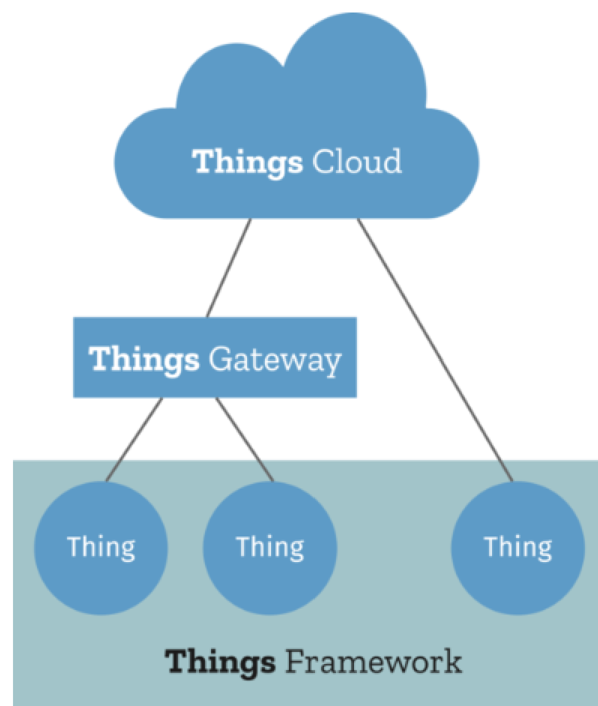


Figure 2.4.: Mozilla - Project Things[5]

In order to connect these services Mozilla is developing the *Web Thing API*. The Web Thing API proposes a plain JSON serialisation to describe Things and a HTTP and WebSockets protocol binding. Both are meant to complement W3C abstract data model and API for the Web of Things. Mozilla's Project Things provides utility for the Web Thing API for major programming languages including Java, Node.Js, Python and Arduino. While the plain standard does not support time series data storing and exploration, there have been projects to extend the API to use time series databases.

[5]hacks.mozilla.org/2017/06/building-the-web-of-things (Accessed 11.02.2019)

2.6.3.1. Web Thing data model

The Web Thing API is designed to make a highly configurable device network. Web Thing API defines embedded devices as *Things*. Things can be assigned unlimited *Properties*, *Actions* and *Events*:

- *Thing*: Representation of a physical device within the World Wide Web.
- *Properties*: Web Thing properties represent a single property of a device. These can be configured to be read-only or writable. A property can take different data types (boolean, integer, array etc.) in JSON format, but can also be images or video streams.
- *Actions*: Actions represent functions that can be carried out on a device. This can be to adjust device properties or to affect a required change of state.
- *Events*: A device can emit events that may occur at a certain change of state, for example when a device property reaches a certain value.

3. SEnviro for Agriculture

Agriculture has been a fundamental industry since the dawn of humanity. Over millennia it has been subject to constant change due to human efforts of improvement. Today, the global human population is greater than ever before and the importance of developing more efficient agricultural systems is considered of paramount importance. Innovative agricultural systems apply integration of various technologies, such as automated systems, wireless communication, sensor systems and mobile applications, eventually enabling the IoT in the agricultural domain (Kamilaris et al., 2016).

This chapter contains an introductory section for state of the art IoT applications for Smart Farming, then provides a detailed description of the *SEnviro for Agriculture* project, which was selected as use case scenario for the development of this master thesis.

3.1. Smart Farming

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

In an example project for monitoring vineyards, Burrell et al. (2004) developed a prototype for sensing nodes communicating with farmers using Short Message Service (SMS) communication. The system focuses on temperature and humidity readings and is built on open software and hardware. An Arduino[1] board with an added *GPRS shield*[2] provides GSM digital cellular network support.

A further example using Arduino microcontrollers in smart farming is presented in Devi et al. (2018), an irrigation support system for farmers in India. The developed WSN nodes include temperature and humidity sensors and a WiFi module to send observations to a web server, which farmers can access remotely. A GSM module also sends SMS notifications when observations reach threshold levels.

[1] www.arduino.cc (Accessed on 12.02.2019)
[2] http://wiki.seeedstudio.com/GPRS_Shield_V2.0 (Accessed on 12.02.2019)

Zhou et al. (2011) propose a sensor network system that connects agriculture to the IoT, focusing on system reliability, longevity, remote management, interoperability and low cost. Sensor nodes in farmlands and greenhouses periodically take relevant environmental measurements, which are collected by a relay node and sent to the communication server via a gateway instance. On the server data is stored in a database, where it is analysed by a Decision Support System (DSS) that publishes relevant guidance for farmers such as notification and alerts. The system hardware and software rely mostly open source products. The *ATmega1281*[3] microprocessor unit and *AT86RF230*[4] RF transceiver ensure power-efficient processing and long-range communication in sensor and relay nodes. *TinyOS*[5] low-power protocols guarantee resource efficiency on software level. The study shows high potential in open-source products for agricultural IoT applications, especially for setting up links among agronomists, regardless of their geographical location.

Smart Agriculture has also been applied in plant disease detection, such as in the research done in Mauri et al. (2011). This project takes a remote sensing approach to detect disease indicators in vineyard foliage. The proposed WSN consists of nodes equipped with sensors taking pictures of vines, which then are processed by the node in the field. If leaf status anomalies, such as a high percentage of brown leaves, are detected the node sends a message to the farmer via a relay node. The *Netgear WNDR3700*[6] wireless router provides the necessary processing capacity for on-site image processing.

In a further application of smart farming in viniculture by Anastasi et al. (2009). The system combines two WSNs, one for monitoring vineyards and other for monitoring the wine cellars. Field node sensors measure temperature, light exposure (e.g. Photosythetically Active Radiation) and humidity and additional sensing units on the border of the field measure wind speed and direction. Nodes in the winery measure temperature and relative humidity, which are necessary parameters to control ambient conditions for wine aging. Field nodes are built on the *TelosB*[7] due to its broad transmission range. Cellar nodes consist of *MICAz*[8] motes coupled with an *MTS310*[9] sensor board. Data from both networks is sent to a central storage unit and analysed for further decision support. The project results triggered farmers to perform corrective actions on their agricultural fields.

[3] www.microchip.com/wwwproducts/en/ATmega1281 (Accessed on 12.02.2019)
[4] www.microchip.com/wwwproducts/en/AT86rf230 (Accessed on 12.02.2019)
[5] http://webs.cs.berkeley.edu/tos (Accessed on 12.02.2019)
[6] www.netgear.com/support/product/WNDR3700v1.aspx#docs (Accessed on 12.02.2019)
[7] www.advanticsys.com/shop/mtmcm5000msp-p-14.html (Accessed on 12.02.2019)
[8] www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf (Accessed on 12.02.2019)
[9] www.memsic.com/wireless-sensor-networks/MTS310 (Accessed on 12.02.2019)

3.2. SEnviro components

Based on the *SEnviro* project from 2015, *SEnviro for Agriculture* takes the previously developed environmental monitoring system further and puts it into an agricultural context. 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 chapter refers to *SEnviro for Agriculture* merely as SEnviro.

Figure 3.1 shows an overview of the SEnviro architecture and components.

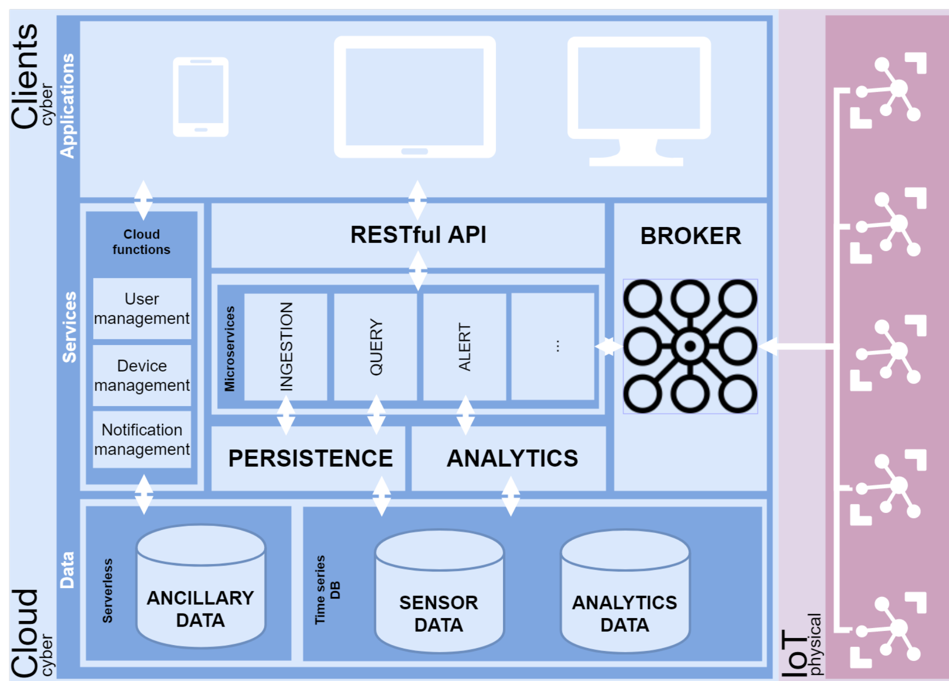


Figure 3.1.: Schema of SEnviro architecture; the pink section represents physical components, the blue section represents software elements

3.2.1. SEnviro nodes

At hardware level (represented in the pink section in *Figure 3.1*), the SEnviro sensorised platform was designed as a smart object, consisting of a similar hardware assembly as the platform presented in Trilles et al. (2015). SEnviro node components can be categorised into four groups depending on their functions: *core*, *sensors*, *power supply*, and *communication*.

The current SEnviro node design differs from its predecessor mainly in its com-

ponents. More concretely, SEnviro nodes were enhanced by using the Electron Particle[10] board as a *core*. The Electron board has an open source design with high performance. *Figure 3.2* shows the *core* of the proposed node.

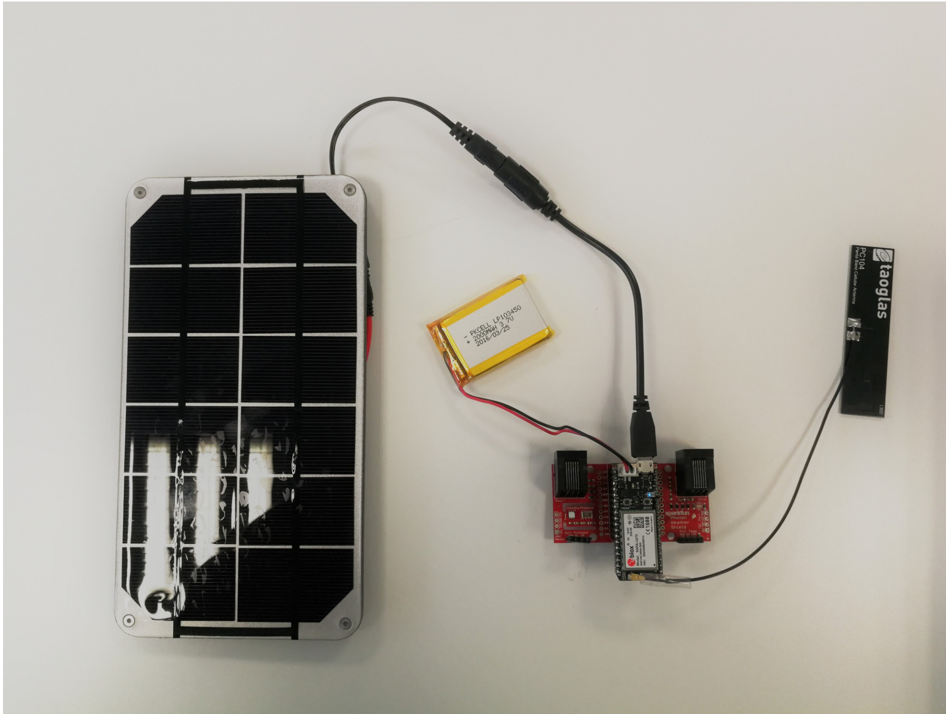


Figure 3.2.: SEnviro node core and power supply assembly

Among other improvements to the system presented in Trilles et al. (2015), SEnviro nodes in *SEnviro for Agriculture* most significantly excel in communication capabilities. The Electron Particle provides 2G and 3G connectivity, rendering it possible for SEnviro nodes to be successfully deployed in any area with mobile data coverage.

SEnviro nodes support Over-The-Air updates for internal software. This feature enhances node maintenance profoundly, allowing nodes to support potential future functionalities and behaviours without having to visit nodes at their locations and do updates manually. To offer an energetically efficient solution a 3.4W solar panel with a lithium battery of 2200 mA provides power to the system. An energy policy for standalone nodes is used to optimise the battery life. The Particle API exposes the battery readings, which are used to adapt the rate of data deliveries depending on the rate of charging and the battery level. To ensure unique node identification, a Quick Response (QR) code is assigned to each SEnviro node, which is used by *SEnviro Connect* (see *3.2.2 SEnviro Connect*) to register a device and start collecting data. A 3D model was designed for a 3D-printable protective box, which encloses hardware components and shields them from external hazards.

[10]docs.particle.io/electron (Accessed on 12.02.2019)

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 fully assembled and deployed SEnviro node is represented in *Figure 3.3*.

Figure 3.3.: Fully assembled and deployed SEnviro node.

3.2.2. SEnviro Connect

The blue section in *Figure 3.1* represents all the elements of SEnviro Connect with their corresponding relations. SEnviro Connect can be divided into three layers: *data*, *services* and *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*.

Firstly, the *broker* is used as a bridge to connect SEnviro nodes with the software platform. The broker is based on a *RabbitMQ*[11] instance, which supports MQTT publish-subscribe messaging (see *2.4 Machine-To-Machine Communication & MQTT*).

[11] www.rabbitmq.com (Accessed on 12.02.2019)

The *micro-services* component is used to provide various functionalities or capabilities that SEnviro Connect offers as a platform. In the current state of development, three functionalities have been considered: *ingestion*, *query* and *alert*. The *ingestion* ensures the access and importation of sensor data for immediate use, ergo for analysis or storage in a database. The *query* micro-service can retrieve sensor data to be consumed by clients from the application layer using a RESTful API. Finally, *alert* micro-service collects all alerts produced by the *analytics module* to transfer this information to final layer.
All micro-services were developed using Micro[12]. Micro can build cloud-native applications with ease and provides an opinionated framework for developing applications with a pluggable architecture.

The third component is the *persistence module*, which is responsible for storing and retrieving data from SEnviro nodes. *Ingestion* and *query* micro-services are connected to this module. Both components are provided by *InfluxDB*[13], an open-source time series database. InfluxDB Go-based database management system optimised for fast, high-availability storage and retrieval of time series data.

The *analytics module*'s primary objective is to define and execute analytics over the incoming sensor data from SEnviro nodes. The *alert* micro-service connects with this module, since alerts may depend on results from the analytics. This feature was installed using Kapacitor, an extension of InfluxDB for alerting and data processing. SEnviro Connect provides two kinds of analytics: one type focuses on the SEnviro node to monitor the node state, 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 a task alert in the *analytics module*. These task alerts are defined using bibliographic works and depend on meteorological phenomena. All the analytics work in real-time. When a new observation arrives, it is used to calculate each task alert and triggers an alarm for certain types of events. Each task alert is defined using TICKscript.

To conclude with the *service* layer, different *cloud functions* were defined to manage and build some ancillary features, such as *user*, *device* or *notification manager*. The *user manager* allows creating, editing, and removing users and assigning SEnviro nodes. The *device manager* creates, edits and removes SEnviro nodes. Moreover, the *notification manager* can launch notifications to clients, such as new alerts detected by the *analytics module*.

The *data* layer is responsible for the storage of all required data for the system, including data from the nodes, from analytics or from auxiliary data. It consists

[12]https://micro.mu (Accessed on 19.02.2019)
[13]www.influxdata.com (Accessed on 12.02.2019)

of two Influx databases for time-series data and a Firebase database, following a serverless approach.

Finally, the *applications* layer exposes the system on the client side. An Angular[14] web application built on modern web technologies, such as HTML5, JavaScript or Cascading Style Sheets (CSS) includes a responsive client that adapts to a device's properties (desktop, mobile or wearable). The web application provides two basic functionalities. One focuses on data visualisation, including a map and graph view of nodes and historical sensor data, and alerts from the disease models. The other provides a node management interface where users can insert nodes and edit their details.

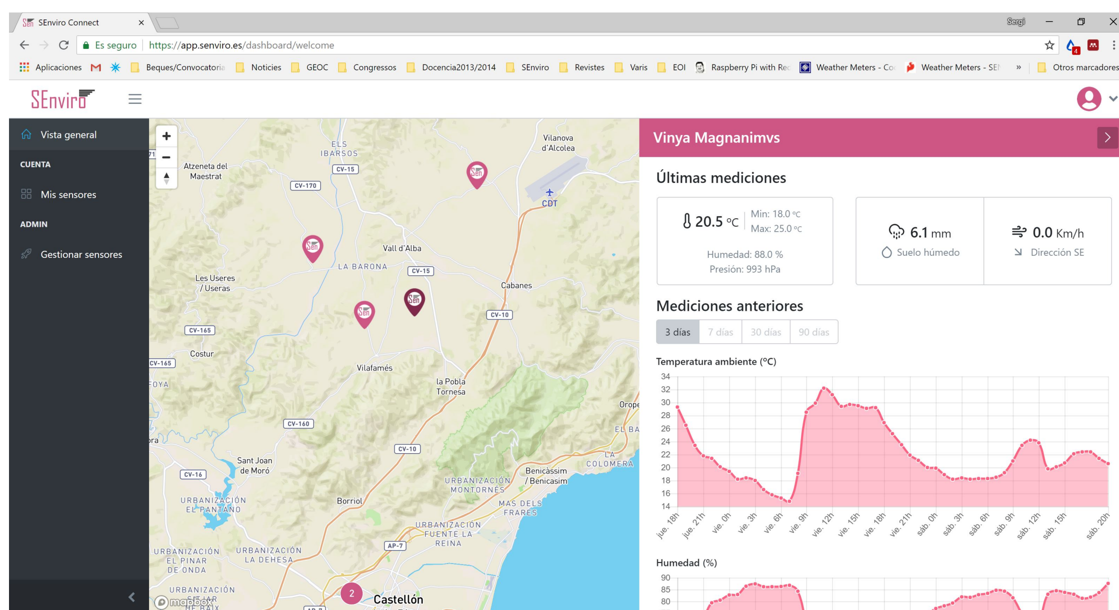


Figure 3.4.: A screen-shot of the SEnviro Connect view to visualise observations and alerts.

[14] https://angularjs.org (Accessed 20.02.2019)

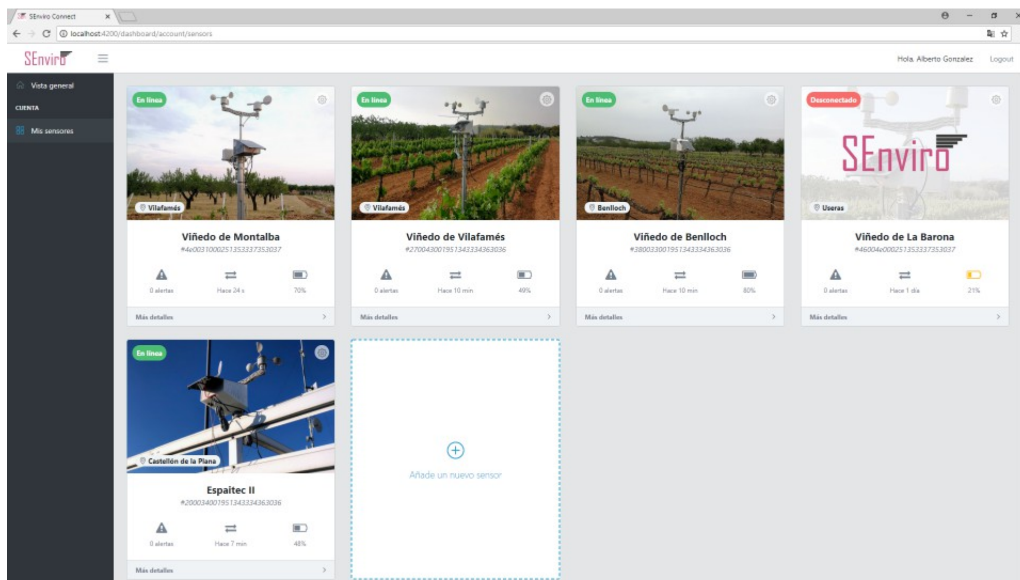


Figure 3.5.: A screen-shot of the SEnviro Connect view to manage SEnviro nodes.

4. Methodology

As mentioned in section *2.6 Open & Sensor Web Standards*, there are currently several organisations doing research in IoT standards and many solutions for a variety of use cases are now openly available on the Internet. Although it is planned to increase interoperability by open standard integration in *SEnviro for Agriculture*, this feature is still missing in the project. This study aims at increasing *SEnviro*'s interoperability by integrating a selection of Web Standards, contrasting the results and selecting the most adequate.

For this project it was decided to select one implementation each for the OGC's *SOS* and *SensorThings API* Web Standards and integrate them into the present SEnviro architecture. These two standards fulfil the needs of environmental monitoring applications, which require spatial referencing and time series functionality in order to see how observed phenomena behave over time on the monitored feature of interest. The experiment was taken further by deploying the *Web Thing API* by Mozilla. In contrast to OGC Sensor Web Standards, Web Thing does not support time series data storing. Nevertheless, it was considered of importance to evaluate the feasibility of extending this still emerging Web Standard for environmental monitoring applications. To embed the instances into the SEnviro architecture, adapter scripts were created that connect them with the SEnviro message broker.

For determining the suitability of the individual standards, the three standards instances were deployed in a dedicated experimental environment and compared in an in-depth contrasting juxtaposition. Section *4.1 Experimental Environment* describes the environment and architecture for the instance deployment. In section *4.2 Deployed Web Standard Implementations*, the Open Web Standard implementations included in the study are described, highlighting their key features, semantics and characteristics. The final section of this chapter explains the process of comparing the Web Standard deployments.

4.1. Experimental Environment

All Web Standard implementations and their adapters were deployed on *Elcano*, a Linux server provided by the Institute for New Imaging Technologies (INIT) at Universitat Jaume I.

Elcano hosts webapps for several projects, which run on the *Docker Enterprise Container Platform*, a powerful open source tool for operating-system level virtualisation. Docker allows developers to create, deploy and run applications in *containers*. Containers package applications with all the necessary components, such as libraries, files and other dependencies, avoiding incompatibilities and assuring that the application runs smoothly in any other Docker environment. Furthermore, unlike conventional Virtual Machines, Docker containers are run by a single operating-system kernel, making them more lightweight.

A Docker container for each Web Standard application (see *4.2 Deployed Web Standard Implementations*) was deployed on Elcano. Web standard adapter scripts (see *4.3 SEnviro Web Standard Integration*) connecting the applications to SEnviro RabbitMQ Broker (located on a dedicated SEnviro server) were also deployed as separate Docker containers, distributing the incoming observations from SEnviro nodes to the corresponding services. In *Figure 4.1* the highlighted section represents the addition to the already established SEnviro architecture.

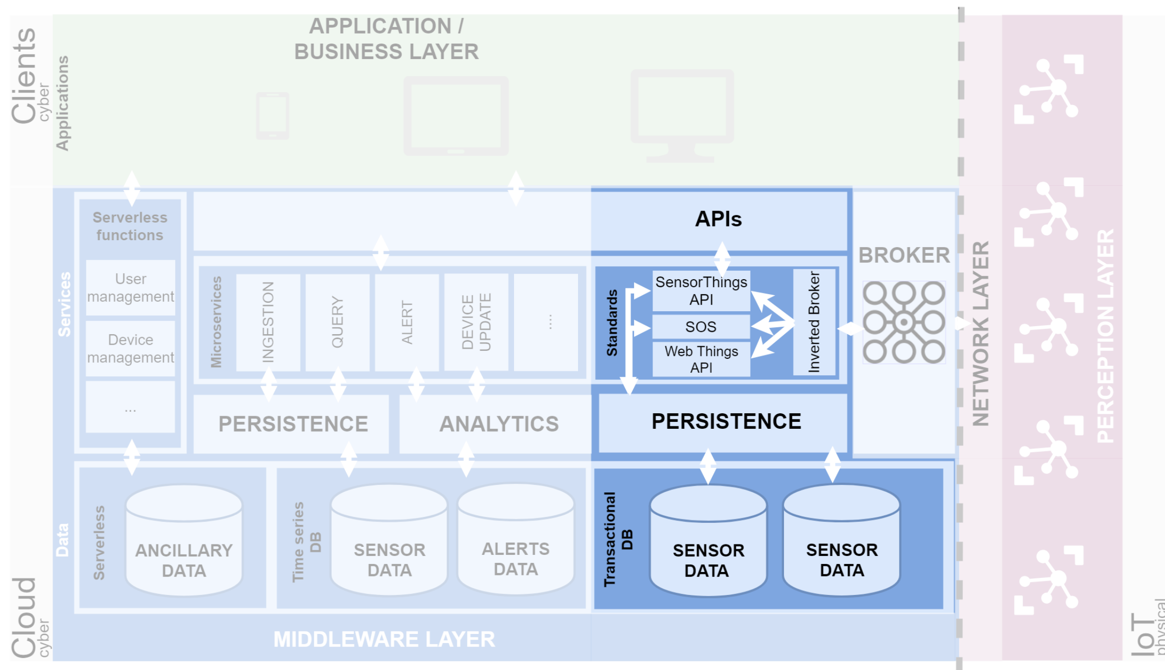


Figure 4.1.: Schematic view of Web Standards SEnviro integration

4.2. Deployed Web Standard Implementations

The following sections shed light on the Web Standard applications deployed for this project. The first section introduces 52North-SOS, based on the OGC's Sensor Observation Service. It is followed up by FROST, an implementation of the OGC's SensorThings API. For the Web Thing API there was no need to deploy a previously established implementation, so it was omitted in this section.

4.2.1. 52North-SOS

Creating an application that makes use of the Sensor Observation Service would demand time and resources which are outside of the scope of this study. Instead, an existing SOS-implementation was selected and deployed.

During the selection process *istSOS 2.3.1* and *52North-SOS 4.4.2* were deployed and tested for SEnviro in separate Docker containers. istSOS was favoured at the beginning since it is written in Python, which was also used for the development of the adapter scripts. Eventually 52North-SOS was selected due to several reasons.

Figure 4.2.: 52North SOS Client interface

52North-SOS features the SOS test client (see *Figure4.2*), 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. The decision was also influenced by the fact that 52North-SOS includes tested Docker configuration files, while with istSOS2 the Docker files had to be created using outdated installation manuals, making the deployment prone to errors. A further reason for the selection of 52North-SOS was the lack of istSOS2 documentation concerning certain necessary operations. Lastly, 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[1] 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 SOS data (see *Figures 4.3* and *4.4*). After some trials with the latest 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 as irrelevant to the scope of this study.

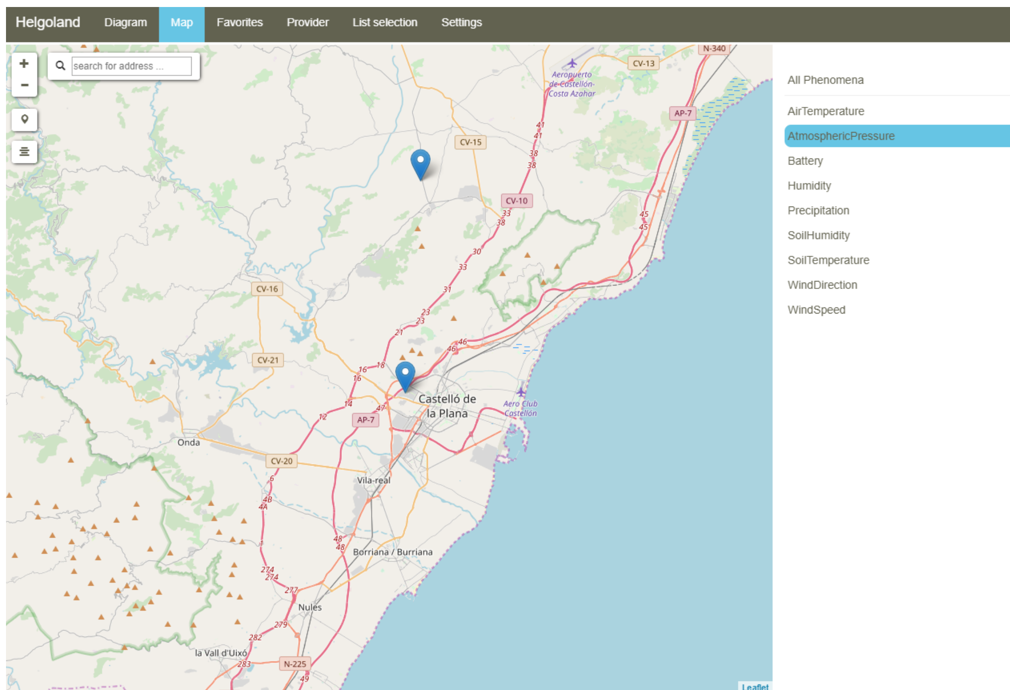


Figure 4.3.: Helgoland Map View

[1]http://tomcat.apache.org (Accessed 19.02.2019)

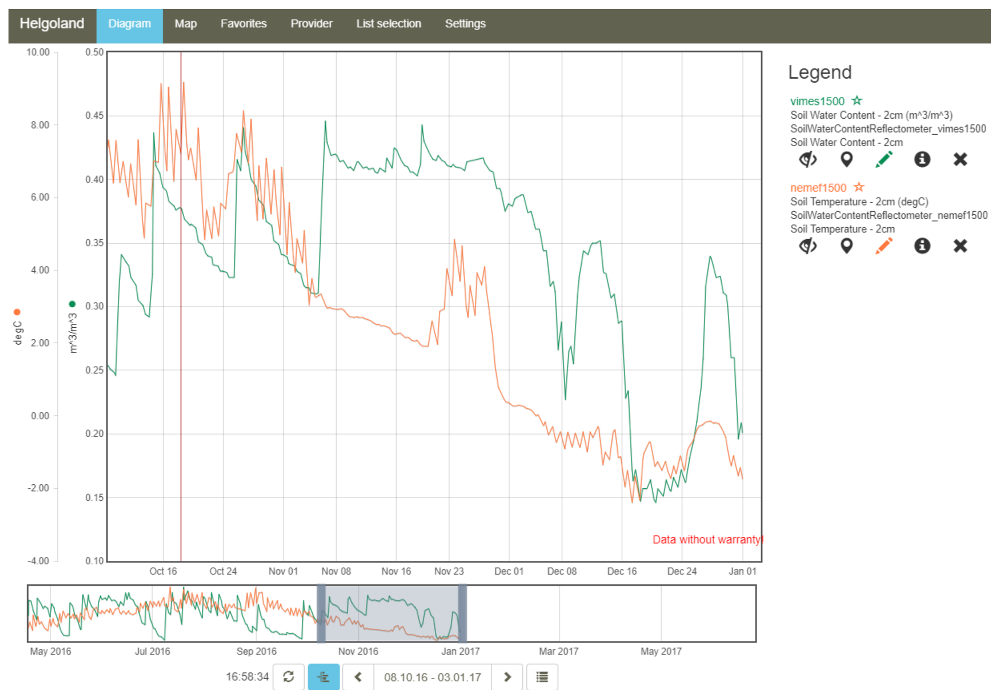


Figure 4.4.: Helgoland Diagram View

4.2.2. FROST-Server

After some in-depth research about potential SensorThings implementations, FROST was eventually selected for the integration into SEnviro. The decision was mostly based on the fact that FROST includes all the features in the OGC compliance test suite and passed it with a full success rate (see *Figure4.1*). It includes MQTT extensions for creating and updating data. Furthermore, FROST provides extensive documentation and deployment resources for easy deployment in Docker environments.

Conformance Class	Reference	Implemented	Test Status
Sensing Core	A.1	Yes	6/6
Filtering Extension	A.2	Yes	42/42
Create-Update-Delete	A.3	Yes	14/14
Batch Request	A.4	Yes	0/0
Sensing MultiDatastream Extension	A.5	Yes	18/18
Sensing Data Array Extension	A.6	Yes	3/3
MQTT Extension for Create and Update	A.7	Yes	4/4
MQTT Extension for Receiving Updates	A.8	Yes	13/13

Table 4.1.: FROST OGC Compliance Testing Status[2]

By default the java-based FROST-Server application launches an Apache Tomcat, but there are also options to configure web server specifications. The application stores all data in a PostGIS extended PostgrSQL database. Fraunhofer IOSB provides several FROST packages, which either comprise HTTP and MQTT operations together or keep them as individual bundles. FROST-Server does not include data visualisation applications, focusing more on the core functionalities of the API.

4.3. SEnviro Web Standard Integration

As mentioned in chapter *3 SEnviro for Agriculture*, SEnviro nodes transmit new values for observed phenomena using MQTT within a RabbitMQ broker. In order to store data in real-time, incoming messages from SEnviro must 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 correctly store the data. This involved general service configuration and inserting stations and

[2]www.github.com/FraunhoferIOSB/FROST-Server (Accessed 11.12.2018)

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 the the OGC standards, SOS and SensorThings, this consisted 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 to in Python for these operations, making use of *Pika*, a Python library to connect to RabbitMQ brokers.

RabbitMQ uses *topics* to categorise messages, which can be chained into routing keys. *Pika* 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/4e0022000251353337353037/SoilHumidity
```

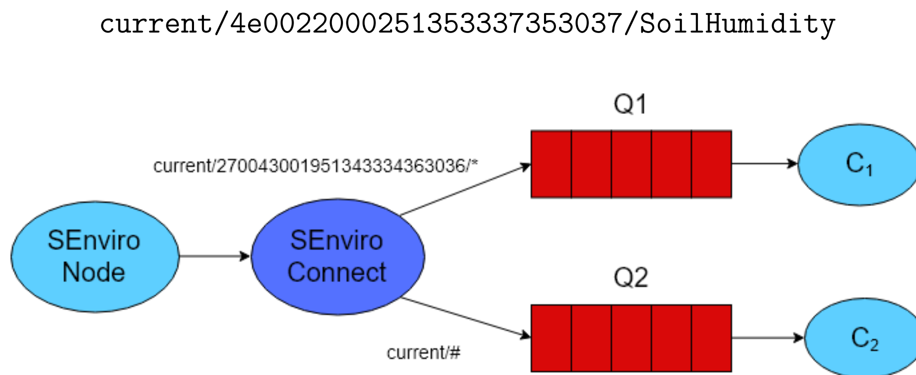


Figure 4.5.: SEnviro message queuing example schema

In the example in *Figure 4.5*, 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 270043001951343334363036. *Q2* queues all messages from all stations.

As mentioned above, the SEnviro node ID and the phenomenon specification is included in the routing key. SEnviro byte messages are structured in a JSON manner and contain only a *time* and a value *attribute*, as demonstrated in the following routing key and example message:

```
current/4e0022000251353337353037/AirTemperature
```

```
{"time":"2019-01-17 13:43:00","value":"6.581871"}
```

Web Standard adapter scripts catch and decode SEnviro byte messages from all

deployed SEnviro nodes and access the node and phenomenon details via the routing key. Using this information, a new message is created and sent to the corresponding web service.

4.3.1. SOS Adapter

After establishing the connection with the SEnviro broker and decoding the message and routing key of incoming messages, there are several steps necessary to create an *insertObservation* request. 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 *observed properties* are retrieved from the intercepted messages. However, some essential information for a successful *insertObservation* request could not be extracted. Therefore, some 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 HTTP POST request.

4.3.2. SensorThings Adapter

Similar to the SOS adapter, the FROST adapter establishes a connection with the SEnviro broker, reads the incoming messages and uses information from the messages to create a JSON object to post to the FROST server with a HTTP POST. Here the challenge lies in posting to the correct datastream.

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 posts observations to the datastream with ID = 11:

```
http://elcano.init.uji.es:8082/FROST-Server/v1.0/
    Datastreams(11)/Observations
```

4.3.3. Web Thing Adapter

The Web Thing Adapter created for SEnviro data combines the Python libraries *Pika* by RabbitMQ and *Webthing* by Mozilla IoT.

Before launching the instances, the script defines a Thing class for SEnviro nodes. Properties for the Thing's location and for the observed phenomena were added to this class, including details for data types, maximum and minimum values, units of measurement and property descriptions. An action was also defined for changing the Thing's coordinates, allowing users to update a node's location in case it is relocated. This action can be executed with a HTTP POST of a JSON object containing the new coordinates using the API's REST interface.

Since the RabbitMQ connection and the Web Thing server must run simultaneously, the *threading* library was used to run both instances on two parallel threads. When messages come in from the SEnviro Broker, the routing key from the incoming messages is used to update property values for the corresponding SEnviro node.

4.4. Comparative Analysis

The aim of this document is to provide insight into the potential of open standard integration to enhance environmental monitoring application interoperability. This is firstly achieved in a qualitative approach to explore the different Web Standard implementations' capabilities, comparing their core operations and identifying potential strengths and weaknesses. Secondly a quantitative analysis is conducted to quantify performance differences for the deployed instances. This involves monitoring performance parameters on the server and contrasting the results of the individual instances.

4.4.1. Performed operations

To compare the deployed Web Standard instances, a set of operations are executed for the services. For the qualitative analysis the request semantics are contrasted for data insertion, deletion, updates and retrieval. For the quantitative analysis, observation retrieval requests were scaled to obtain increasing quantities of data.

4.4.1.1. Data management

The deployed services for the corresponding Web Standards needed to be configured for SEnviro before any observations could be stored. This included inserting instances representing stations and their properties. Once the services were set up correctly, observations could be inserted using the corresponding adapters. Apart from data insertion operations, basic operations for deleting and updating data were investigated for the different services.

4.4.1.2. Accessing metadata

Accessing information about the sensors or sensor systems is a core operation for any monitoring service, providing the means to identify the origin of observations, the location of the monitoring station and information about the monitored parameters. The two example operations below show ways to access detailed information about *procedures*, representing sensor systems in SOS, and *things*, representing embedded devices in the SensorThings API:

- 52North-SOS:
  - **Operation**: SOS - describeSensor (HTTP POST)
  - **URL**:

    ```
    http://elcano.init.uji.es:8084/52n-sos-webapp/service
    ```

  - **Post data**:

    ```
    {
     "request":"DescribeSensor",
     "service":"SOS",
     "version":"2.0.0",
     "procedure":"270043001951343334363036",
     "procedureDescriptionFormat":
        "http://www.opengis.net/sensorml/2.0"
    }
    ```

- FROST (HTTP GET):

  ```
  http://elcano.init.uji.es:8082/FROST-Server/v1.0/Things(2)
  ?$expand=Locations,HistoricalLocations,Datastreams/
  ObservedProperty,Datastreams/Sensor
  ```

- Web Thing API (HTTP GET):

  ```
  http://elcano.init.uji.es:5000/0
  ```

4.4.1.3. Observation retrieval

Arguably the most important feature in any monitoring service is accessing the stored time series data to visualise phenomena's behaviour over time. This is not possible for the Web Thing API by default, since it does not support time series data storing. FROST and 52North-SOS have different approaches obtain observations. The operations were contrasted according to their semantics and on a performance level.

52North-SOS supports all the standard SOS operations, but their data visualisation tool, the Helgoland Client, uses its own request method and API to obtain sensor data. 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 time and value of observations, improving performance. An equivalent FROST SensorThings query, limiting result details to observation time and value, was created for comparison with Helgoland.

The test queries below obtain **1000 observations** within the time span of **2019-01-14 14:14:52** and **2019-01-21 11:20:52** for each service:

- **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:

    ```
    http://elcano.init.uji.es:8084/52n-sos-webapp/service
    ```

  - Post data:

    ```
    {
      "request": "GetObservation",
      "service": "SOS",
      "version": "2.0.0",
      "procedure": "27004300195134333436303 6",
      "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 a HTTP GET request to the server, which returns the requested values and time stamps and displays them in an interactive diagram.

```
http://elcano.init.uji.es:8084/52n-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 a 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 contraints.

```
http://elcano.init.uji.es:8082/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.

```
http://elcano.init.uji.es:8082/FROST-Server/v1.0/
Datastreams(18)/Observations?$top=1000&$select=
phenomenonTime,result&$filter=phenomenonTime%20gt%20
2019-01-14T14:14:52Z%20and%20phenomenonTime%20lt%20
2019-01-21T11:20:52Z
```

4.4.2. Qualitative Analysis

A crucial factor in the evaluation of the Web Standard instances is the quality of the service. This strongly affects the flexibility, scalability and eventually, the developer experience of applications. The quality of the services was assessed in multiple steps.

Firstly, the deployment and configuration process for each web service was described and compared. Subsequently a comparison of available operations for different uses (see *4.4.1 Performed operations*) was performed from data producer and data consumer perspectives. On the data producer side are operations to insert, update and delete entities like sensors, sensor platforms and observations. On the consumer side are operations for querying and obtaining data and metadata. Web Standard semantics were evaluated based on their encoding formats and data traffic protocols. The results for each Web Standard were summarised and contrasted. The

comparison was then used to determine the suitability for the use-case of SEnviro.

4.4.3. Quantitative Analysis

The quantification of differences in performance between the deployed web services were monitored on various levels and using a selection of tools. These were used to monitor *response time*, *response size*, *CPU* and *Memory*.

The following tools were used for performance monitoring:

- *Postman*: Postman[3] 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 returned file. Requests are saved in *collections* and can be run at a regular schedule and monitored on the Postman Dashboard, accessible in a user's workspace on the web browser. Postman informs the user when there are issues with collection monitors in an email alert. Tests run up to two times per hour in the free version of the software.

- *JMeter*: JMeter[4] is a project by the Apache Software Foundation. It is an open-source software Java application, designed for load testing functional behaviour and measure performance in web applications. JMeter can simulate heavy loads on a server, cluster or network with a large selection of application and protocol types. There are several Plugins available that can add and/or extend JMeter's functions. Monitoring the metrics in the web service's hosting server required adding the *PerfMon Metrics Collector* plugin to JMeter and deploying the *PerfMon Server Agent* tool on hosting server, which opens a port for JMeter to connect to remotely.

- *cAdvisor*: cAdvisor is a simple tool provided by Google for monitoring Docker container behaviour. Apart from a simple user interface showing graph visualisation of the container metrics, cAdvisor provides several APIs for accessing container metrics data.

4.4.3.1. Response times and sizes

Response times and sizes are measured for all metadata and observation retrieval operations using Postman Monitors. Since Postman Monitors run as cloud services,

[3] www.getpostman.com (Accessed 19.02.2019)
[4] https://jmeter.apache.org/download_jmeter.cgi (Accessed 19.02.2019)

test queries don't depend on network connectivity once they have been deployed. Requests for observations are monitored for 24 hours with 2 two requests per hour, resulting in 48 values per query. Postman Monitors automatically calculate response time averages, which are extracted and used for further analyses.

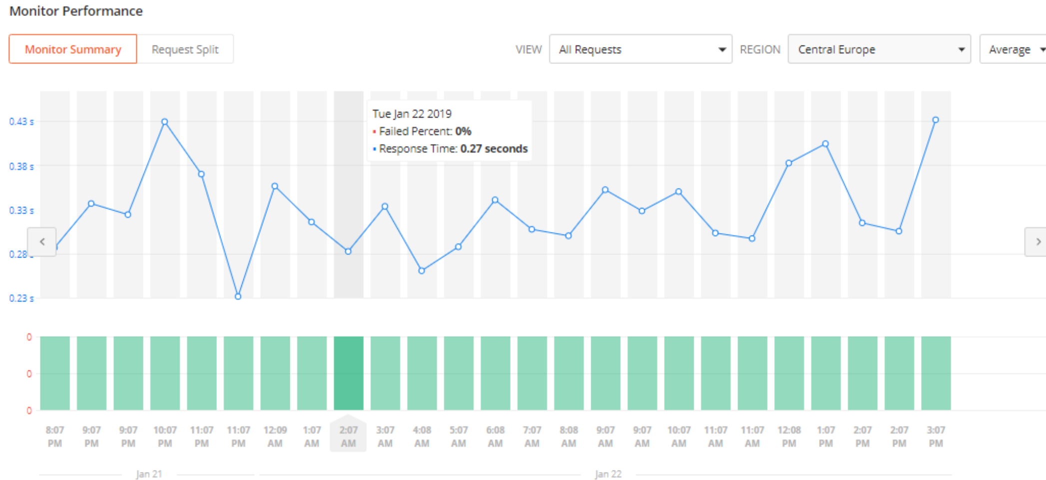


Figure 4.6.: Postman Response Time Monitoring Interface

4.4.3.2. Web service metrics

To compare the performance for obtaining observations, 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 1000 observations. The maximum of 1000 observations was selected due to the FROST default configuration, which sets the maximum number of FROST observations contained in a single response file to 1000. This can be modified in the source code and should be considered in further studies involving FROST.

The work flow for metrics monitoring relies on measuring the metrics of the individual Docker containers. cAdvisor provides the means to access the container metrics data an API. The selected REST API[5] 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 after 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 implemen-

[5]https://github.com/google/cadvisor/blob/master/docs/api_v2.md (Accessed 19.02.2019)

tations. The container CPU values are divided by the server CPU usage to reflect how many server resources the containers require in percent. Memory values are calculated in bytes and then converted to megabytes for data visualisation.

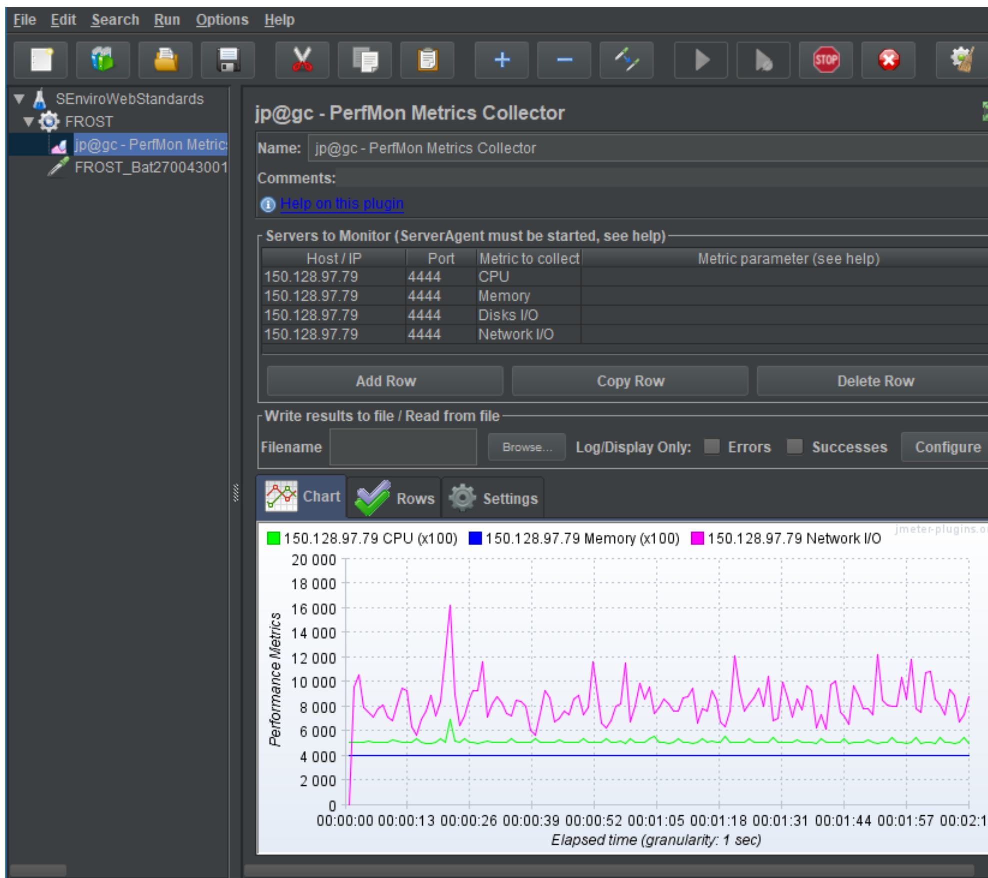


Figure 4.7.: JMeter Performance Monitoring interface

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. The output CSV files were eventually loaded into R, were data was analysed and plotted.

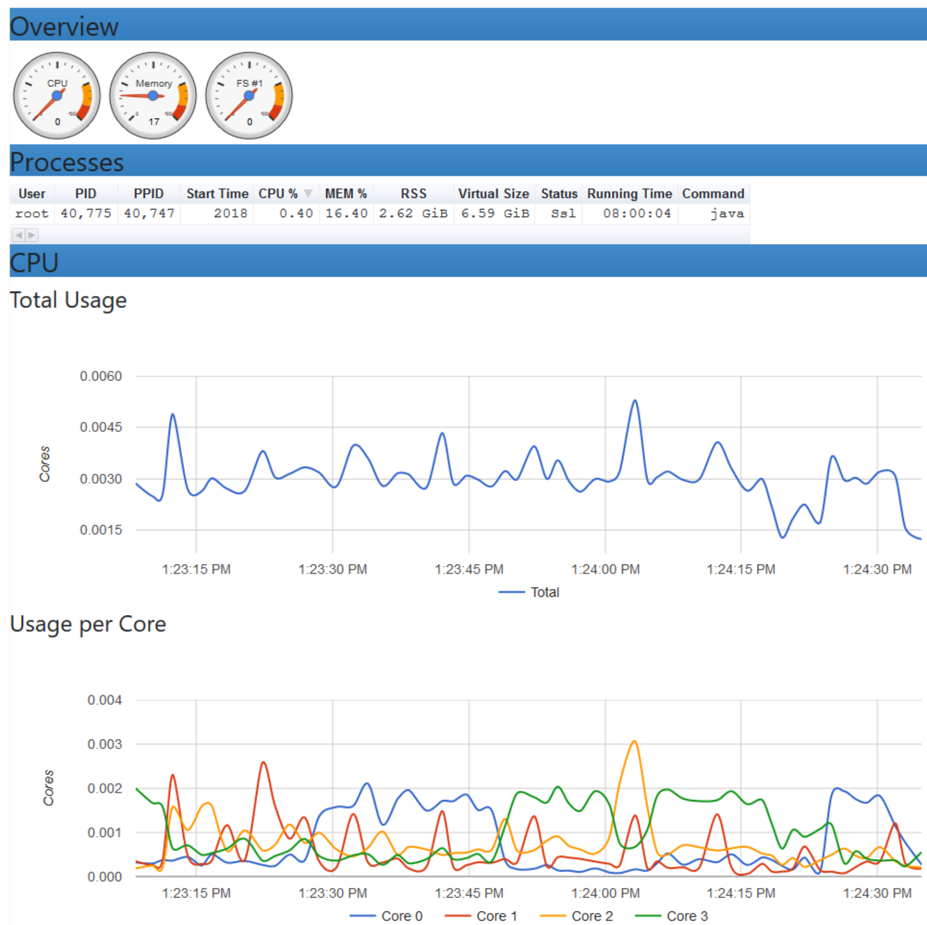


Figure 4.8.: cAdvisor interface

5. Web Standard Comparison

In order to make a tangible comparison, the deployed Web Standard instances were contrasted on multiple levels. It was determined that the core elements of comparison should be *performance* in a quantitative approach and *semantics* in a qualitative analysis.

This chapter presents results of the quantitative evaluation methods, descriptive sections about the semantic differences between the deployed instances and a discussion concerning the aptitude of the deployed Web Standard instances, considering the requirements of SEnviro for Agriculture, IoT and environmental monitoring applications in general.

5.1. Web Thing API evaluation

Eventually the Web Thing API was for the most part excluded from the Web Standard comparison. This was done due to its lack of features and low complexity, making it mostly inapt for direct contrasting.

Firstly, the standard does not define all essential operations for environmental monitoring applications. The Web Thing API adapter (see *4.3.3 Web Thing Adapter* for details) launches all the instances within the script by defining the Thing class with its properties (observed phenomena, location). Once the service is running, the standard does not provide operations to add further devices or properties. This strongly collides with necessary SEnviro features, since adding new monitoring stations is an essential functionality of the service. Similarly, the Web Thing API does not provide operations to remove devices.

Web Thing does provide means to access and update properties and metadata. Information about devices and properties can be obtained using HTTP GET requests with corresponding API URL. Available actions can be defined in the script launching the server, which can be configured to update information via HTTP POST requests. The following request obtains the current value of the observed properties:

```
http://elcano.init.uji.es:5000/0/properties
```

```
{
    "Coordinates": [
        39.993934,
        -0.073863
        ],
    "AirTemperature": -46.581871,
    "Humidity": 0.0,
    "AtmosphericPressure": -9.99,
    "Precipitation": 0.0,
    "WindDirection": 5.0,
    "WindSpeed": 4.32,
    "SoilTemperature": 0.0,
    "SoilHumidity": 2134.0,
    "Battery": 58.84375
}
```


A crucial feature the Web Thing API does not include by default is support for time series data, making it impossible to browse monitoring data over time. While extending the standard to store data in a database could be an option for future work, this would mean going beyond Web Thing's functionalities and therefore would not be compliant to the standard anymore. Attempts made by Mozilla to extend Web Thing to support time series data is discussed in *5.4 Discussion*.

5.2. 52North-SOS & FROST qualitative evaluation

In this section of analysis results presents different aspects of the service quality of 52North-SOS and FROST. These include the deployment process, service configuration and adaptability. Subsequently several operations for the deployed open standard implementations are described and contrasted. The operations were specified considering the requirements for SEnviro and Smart Farming projects in general. The first subsection focuses on the deployment and configuration of the Web Standard implementations. Subsequently, data producer operations, such as data insertion (sensors, stations, observations) are described. Finally, operations for data consumers are presented, which consist of metadata retrieval operations and several variants of querying and obtaining observation data.

5.2.1. Service deployment & configuration

Since both 52North-SOS and FROST include Docker deployment files (Dockerfile, docker-compose.yml), the deployments for 52North-SOS and FROST follow similar steps. However, the two Web Standard implementations have a distinct setup process and service configuration. The following two subsections shed light on the setup process for either service, including their configuration for SEnviro.

5.2.1.1. 52NorthSOS setup

52North-SOS includes a web user interface with a large 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 datasource, the spatial reference system, the services' and datasource's timezones, access rights and logging.

For the SEnviro configuration of 52North-SOS, transactional security was disabled for Elcano's IP address, 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 (see more details about inserting sensors into SOS in section *5.2.2.1 52North-SOS Transactional Operations*). As mentioned in section *4.3.1 SOS Adapter*, creating the adapter to divert SEnviro observations into SOS

required an extra file containing the coordinates of the stations.

5.2.1.2. FROST setup

FROST is a mere implementation of the SensorThings API and does not include a client interface. Service settings are configurable in the source code, but this was not necessary for SEnviro integration.

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 (see *5.2.2.2 FROST Create-Update-Delete operations* for more details).

5.2.2. Data management

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 the *InsertObservation* operations. 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. Both services have their own features and capabilities.

5.2.2.1. 52North-SOS transactional operations

Before observations can be inserted into SOS, entities have to inserted representing the devices generating the observation data and must also include information about the phenomena they are measuring for a successful observation insertion.

52North-SOS supports several encoding formats including JSON, which is used in this project due to its compatibility with Python, as mentioned in section *4.2.1 52North-SOS*. The 52North-SOS equivalent to the *RegisterSensor* operation is *InsertSensor*. For this operation *InsertSensor* request file must be posted to the server. The example JSON object below contains the mandatory information for a success-

ful insertion request.


```
{
"request": "InsertSensor",
"service": "SOS",
"version": "2.0.0",
"procedureDescriptionFormat": "http://www.opengis.net/sensorML/1.0.1",
"procedureDescription":
 "<sml:SensorML xmlns:swes=\"http://www.opengis.net/swes/2.0\"
 xmlns:sos=\"http://www.opengis.net/sos/2.0\"
 xmlns:swe=\"http://www.opengis.net/swe/1.0.1\"
 xmlns:sml=\"http://www.opengis.net/sensorML/1.0.1\"
 xmlns:gml=\"http://www.opengis.net/gml\"
 xmlns:xlink=\"http://www.w3.org/1999/xlink\"
 xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
 version=\"1.0.1\"><sml:member><sml:System><sml:identification>
 <sml:IdentifierList><sml:identifier name=\"uniqueID\">
 <sml:Term definition=\"urn:ogc:def:identifier:OGC:1.0:uniqueID\">
 <sml:value>270043001951343334363036</sml:value></sml:Term>
 </sml:identifier><sml:identifier name=\"longName\">
 <sml:Term definition=\"urn:ogc:def:identifier:OGC:1.0:longName\">
 <sml:value>270043001951343334363036</sml:value></sml:Term>
 </sml:identifier><sml:identifier name=\"shortName\">
 <sml:Term definition=\"urn:ogc:def:identifier:OGC:1.0:shortName\">
 <sml:value>270043001951343334363036</sml:value></sml:Term></sml:identifier>
 </sml:IdentifierList></sml:identification>
 <sml:capabilities name=\"offerings\"><swe:SimpleDataRecord>
 <swe:field name=\"offering270043001951343334363036\">
 <swe:Text definition=\"urn:ogc:def:identifier:OGC:offeringID\">
 <swe:value>offering270043001951343334363036</swe:value></swe:Text>
 </swe:field></swe:SimpleDataRecord></sml:capabilities>
 <sml:capabilities name=\"featuresOfInterest\"><swe:SimpleDataRecord>
 <swe:field name=\"featureOfInterestID\"><swe:Text>
 <swe:value>featureOfInterest270043001951343334363036</swe:value>
 </swe:Text></swe:field></swe:SimpleDataRecord></sml:capabilities>
 <sml:position name=\"sensorPosition\">
 <swe:Position referenceFrame=\"urn:ogc:def:crs:EPSG::4326\"><swe:location>
 <swe:Vector gml:id=\"STATION_LOCATION\"><swe:coordinate name=\"easting\">
 <swe:Quantity axisID=\"x\"><swe:uom code=\"degree\"/>
 <swe:value>-0.061000</swe:value></swe:Quantity></swe:coordinate>
 <swe:coordinate name=\"northing\"><swe:Quantity axisID=\"y\">
 <swe:uom code=\"degree\"/><swe:value>40.133098</swe:value></swe:Quantity>
 </swe:coordinate></swe:Vector></swe:location></swe:Position></sml:position>
 <sml:inputs><sml:InputList><sml:input name=\"senviroNodePhenomena\">
 <swe:ObservableProperty definition=\"senviroNodePhenomena\"/></sml:input>
 </sml:InputList></sml:inputs><sml:outputs><sml:OutputList>
 <sml:output name=\"AirTemperature\">
 <swe:Category definition=\"AirTemperature\">
```

```
  <swe:codeSpace xlink:href=\"AirTemperature\"/></swe:Category></sml:output>
  <sml:output name=\"Humidity\"><swe:Category definition=\"Humidity\">
  <swe:codeSpace xlink:href=\"Humidity\"/></swe:Category></sml:output>
  <sml:output name=\"AtmosphericPressure\">
  <swe:Category definition=\"AtmosphericPressure\">
  <swe:codeSpace xlink:href=\"AtmosphericPressure\"/></swe:Category>
  </sml:output><sml:output name=\"Precipitation\">
  <swe:Category definition=\"Precipitation\">
  <swe:codeSpace xlink:href=\"Precipitation\"/></swe:Category></sml:output>
  <sml:output name=\"WindDirection\"><swe:Category definition=\"WindDirection\">
  <swe:codeSpace xlink:href=\"WindDirection\"/></swe:Category></sml:output>
  <sml:output name=\"WindSpeed\"><swe:Category definition=\"WindSpeed\">
  <swe:codeSpace xlink:href=\"WindSpeed\"/></swe:Category></sml:output>
  <sml:output name=\"SoilTemperature\">
  <swe:Category definition=\"SoilTemperature\">
  <swe:codeSpace xlink:href=\"SoilTemperature\"/></swe:Category></sml:output>
  <sml:output name=\"SoilHumidity\"><swe:Category definition=\"SoilHumidity\">
  <swe:codeSpace xlink:href=\"SoilHumidity\"/></swe:Category></sml:output>
  <sml:output name=\"Battery\"><swe:Category definition=\"Battery\">
  <swe:codeSpace xlink:href=\"Battery\"/></swe:Category></sml:output>
  </sml:OutputList></sml:outputs></sml:System></sml:member></sml:SensorML>",
 "observableProperty": [
  "AirTemperature",
  "Humidity",
  "AtmosphericPressure",
  "Precipitation",
  "WindDirection",
  "WindSpeed",
  "SoilTemperature",
  "SoilHumidity",
  "Battery"
 ],
 "observationType": [
  "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement",
  "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_CategoryObservation",
  "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_CountObservation",
  "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_TextObservation",
  "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_TruthObservation",
  "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_GeometryObservation",
  "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_ReferenceObservation"
 ],
 "featureOfInterestType":
  "http://www.opengis.net/def/samplingFeatureType/OGC-OM/2.0/SF_SamplingPoint"
 }
```


The file must contain a list of details, including unique procedure ID, procedure long 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 approx. 4829 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 the Web Standard includes operations to delete or update details of procedures, which 52North added as extended operations to enable these actions. *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. The file is comparatively small in size, as demonstrated in the JSON version of the *DeleteSensor* request below. One detail worth noting is that the mandatory SOS offering created with the procedure will remain in the service even when the procedure it is linked to is deleted. Offering names act as unique identifiers, which means if a procedure is reinserted it will need a new offering ID.

```
{
  "request": "DeleteSensor",
  "service": "SOS",
  "version": "2.0.0",
  "procedure": "01234567890123456789O123"
}
```

After setting up sensors and their properties correctly within SOS, observations can be inserted using the *InsertObservation* operation. Similarly, to the previously described operations, this operation is 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 most importantly, the time and value of the observation. The object size for SEnviro *InsertObservation* requests is approx. 1185 bytes. The example JSON *InsertObservation* request below shows all the mandatory details required.

```
{
  "request": "InsertObservation",
  "service": "SOS",
  "version": "2.0.0",
  "offering": "offering01234567890123456789012345678901234",
  "observation": {
    "identifier": {
      "value": "1",
      "codespace": ""},
    "type":
    "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement",
    "procedure": "0123456789012345678901234",
    "observedProperty": "AirTemperature",
    "featureOfInterest": {
      "identifier": {
        "value": "featureOfInterest0123456789012345678901234",
        "codespace": ""},
      "name": [
        {
          "value": "0123456789012345678901234",
          "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.

52North 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* of the the observation items in the PostgreSQL database to from "F" to "T". SOS then excludes these from the service.

5.2.2.2. FROST ***Create-Update-Delete*** requests

The SensorThings API data model demands a different approach when inserting data. Every entity within a SensorThings API has its unique ID and 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 before enabling the storage of observations. The five SensorThings key components (Things, Datatreams, ObservedProperties, Sensors, Observations) are interrelated, with the core entity linked with the rest being the Datastreams. Consequently, all the other entities need to be predefined to start inserting Observations. 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:

```
http://elcano.init.uji.es:8082/FROST-Server/v1.0/Things
```

When new entities are created they are automatically assigned a unique ID. If an entity is removed its ID remains stored in the system and cannot be used again. All entities have mandatory properties that must be included when posting to the server. Metadata about the specific Thing can be added in the *properties* property related to specific Things are inserted in the *properties*. Furthermore, some 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 example Thing object:

```
{
  "name": "0123456789012345678901234",
  "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-0123456789012345678901234",
  "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 creates a Thing with its mandatory properties (name, 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 approx. 4731 bytes.

Inserting SensorThings 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 string in *ISO 8601* format and the value of the measurement, as shown below. The approximate size of SEnviro observations posted to FROST by the adapter is 65 bytes.

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

Properties of any SensorThings entity can be updated by executing a HTTP Patch request its unique URL with a JSON object containing the properties to be updated and the new values. Sending the example JSON object to the target URL (both shown below) with a HTTP PATCH request updates the name and description properties of a registered sensor with ID=7.

```
http://elcano.init.uji.es:8082/FROST-Server/v1.0/Sensors(7)
```

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

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

5.2.3. Retrieving metadata

The values and time stamps 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.

5.2.3.1. 52North-SOS metadata operations

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 most 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 (Open Geospatial Consortium, 2007). The code below shows the 52North-SOS JSON version of the request. An example for 52North-SOS *getCapabilities* response is not presented here, considering the size of the response object (over 1600 lines of code).

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

Further operations use similar requests to get information about station 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.

5.2.3.2. FROST metadata queries

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 2.3. 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:

```
http://elcano.init.uji.es:8082/FROST-Server/v1.0/Things(2)
?$expand=Datastreams($expand=ObservedProperty)
```


```
{
"name" : "270043001951343334363036",
"description" :
"Senviro monitoring station with ID: 270043001951343334363036",
"Locations@iot.navigationLink" :
"http://elcano.init.uji.es:8082/FROST-Server/v1.0/Things(2)/Locations",
"HistoricalLocations@iot.navigationLink" :
"http://elcano.init.uji.es:8082/FROST-Server/v1.0/Things(2)
        /HistoricalLocations",
"Datastreams@iot.navigationLink" :
"http://elcano.init.uji.es:8082/FROST-Server/v1.0/Things(2)/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/2019-01-30T14:46:55.060Z",
    "resultTime" : "2019-01-30T14:26:27.000Z/2019-01-30T14:46:14.000Z",
    "ObservedProperty" : {
      "name" : "Battery",
      "definition" : "https://en.wikipedia.org/wiki/Electric_battery",
      "description" : "Battery readings in %",
      "@iot.id" : 9,
      "@iot.selfLink" :
      "http://elcano.init.uji.es:8082/FROST-Server/v1.0/ObservedProperties(9)"
    },
    "@iot.id" : 22,
    "@iot.selfLink" :
    "http://elcano.init.uji.es:8082/FROST-Server/v1.0/Datastreams(22)"
}],
"MultiDatastreams@iot.navigationLink" :
"http://elcano.init.uji.es:8082/FROST-Server/v1.0/Things(2)/MultiDatastreams",
"@iot.id" : 2,
"@iot.selfLink" :
"http://elcano.init.uji.es:8082/FROST-Server/v1.0/Things(2)"
}
```


The Thing with ID=1 is returned with all its main properties and with its ex-

panded datastreams, which in this case is only "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.

5.2.4. Observation queries

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

5.2.4.1. 52North-SOS queries

SOS observations are obtained using the *getObservation* request. A JSON file containing the query parameters is posted to the server, 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*.

```
{
  "request": "GetObservation",
  "service": "SOS",
  "version": "2.0.0",
  "procedure": "270043001951343334363036",
  "offering": "offering270043001951343334363036",
  "observedProperty": "AirTemperature",
  "featureOfInterest": "featureOfInterest270043001951343334363036",
  "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 mentioned in section 5.2.2.1. 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 prior to invoking the operation. Example request and response objects are presented below.

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

```
{
  "type" :
    "http://www.opengis.net/def/observationType/
            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" : "featureOfInterest270043001951343334363036"
```

```
    },
    "name" : {
      "codespace" : "http://www.opengis.net/def/nil/OGC/0/unknown",
      "value" : "270043001951343334363036"
    },
    "sampledFeature" : "parent270043001951343334363036",
    "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
  }
}
```


5.2.4.2. FROST queries

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:

```
http://elcano.init.uji.es:8082/FROST-Server/v1.0/Datastreams(12)/
Observations?$count=true&$skip=500&$top=50&$select=resultTime,result
&$orderby=result
```

The query returns the top 50 observations from the Datatream with ID=12, 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 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 func-

tions. *Table 5.1* and *Table 5.2* list built-in operators and functions that can be used within a filter.

By using the operators and functions in *Tables 5.1* and *5.2* 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 2500 m^3/m^3.

```
http://elcano.init.uji.es:8082/FROST-Server/v1.0/Datastreams?$
filter=substringof('SoilHumidity',name)&$expand=
Observations($filter=result lt 2500)
```

Operator	Description	Example
eq	Equal	/ObservedProperties?$filter=name eq 'Area Temperature'
ne	Not equal	/ObservedProperties?$filter=name ne 'Area Temperature'
gt	Greaterthan	/Datastreams(id)/Observations?$filter=result gt 20.0
ge	Greater than or equal	/Datastreams(id)/Observations?$filter=result ge 20.0
lt	Less than	/Datastreams(id)/Observations?$filter=result lt 100
le	Less than or equal	/Datastreams(id)/Observations?$filter=result le 100
and	Logical and	/Datastreams(id)/Observations?$filter=result le 3.5 and FeatureOfInterest/id eq '1'
or	Logical or	/Datastreams(id)/Observations?$filter=result gt 20 or result le 3.5
not	Logical negation	/Things?$filter=not startswith(description,'test')
( )	Precedence grouping	/Datastreams(id)/Observations?$filter=(result sub 5) gt 10

Table 5.1.: Built-in operators for the SensorThings Filter operator

String Functions	
bool substringof(string searchString,string baseString)	substringof('Sensor Things',description)
bool endswith(string baseString, string suffix)	endswith(description,'Things')
bool startswith(string baseString, string prefix)	startswith(description,'Sensor')
int length(string p0)	length(description) eq 13
string tolower(string p0)	tolower(description) eq 'sensor things'
string toupper(string p0)	toupper(description) eq 'SENSOR THINGS'
Date Functions	
int year	year(resultTime) eq 2015
int month	month(resultTime) eq 12
int day	day(resultTime) eq 8
int hour	hour(resultTime) eq 1
int minute	minute(resultTime) eq 0
int second	second(resultTime) eq 0
Math Functions	
round	round(result) eq 32
floor	floor(result) eq 32
ceiling	ceiling(result) eq 33
Geospatial Functions	
double geo.distance(Point p0, Point p1)	geo.distance(location, geography'POINT (30 10)')
double geo.length(LineString p0)	geo.length(geography'LINESTRING (30 10, 10 30, 40 40)')
bool geo.intersects(Point p0, Polygon p1)	geo.intersects(location,geography'POLYGON ((30 10, 10 20, 20 40, 40 40, 30 10))')
Spatial Relationship Functions	
bool st_equals	st_equals(location, geography'POINT (30 10)')
bool st_disjoint	st_disjoint(location, geography'POLYGON ((30 10, 10 20, 20 40, 40 40, 30 10))')
bool st_touches	st_touches(location, geography'LINESTRING (30 10, 10 30, 40 40)')
bool st_within	st_within(location, geography'POLYGON ((30 10, 10 20, 20 40, 40 40, 30 10))')
bool st_overlaps	st_overlaps(location, geography'POLYGON ((30 10, 10 20, 20 40, 40 40, 30 10))')
bool st_crosses	st_crosses(location, geography'LINESTRING (30 10, 10 30, 40 40)')
bool st_intersects	st_intersects(location, geography'LINESTRING (30 10, 10 30)')
bool st_contains	st_contains(location,geography'POINT (3 1)')
bool st_relate	st_relate(location, geography'POLYGON ((30 10, 10 20, 20 40, 40 40, 30 10))', 'T****')

Table 5.2.: Built-in functions for the SensorThings Filter operator

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 within the query parameters have been served. Output objects for unmodified FROST observation requests have the following format:


```
{
  "@iot.nextLink" :
  "http://elcano.init.uji.es:8082/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":
        "http://elcano.init.uji.es:8082/FROST-Server/v1.0/
                Observations(1)/Datastream",
     "FeatureOfInterest@iot.navigationLink":
        "http://elcano.init.uji.es:8082/FROST-Server/v1.0/
                Observations(1)/FeatureOfInterest",
     "@iot.id" : 1,
     "@iot.selfLink" :
        "http://elcano.init.uji.es:8082/FROST-Server/v1.0/
                Observations(1)"
    }]
}
```

5.3. 52North-SOS & FROST performance evaluation

The results for the monitored performance parameters were extracted, stored and analysed. Analysis results were then visualised on a set of plots, which were created in *R*, a powerful tool for statistical computing and graphics. The following sections present the the results from the parameters selected for the performance evaluation of the deployed Web Standard implementations. Response times and file sizes were measured using Postman collections, while JMeter and cAdvisor was used to monitor CPU and memory. As mentioned in 4.1, all project instances were deployed on Elcano server. *Figure 5.1* shows Elcano's core specifications.

```
Processor        :   4x Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz
Memory           :   16719MB (8458MB used)
Operating System :   Ubuntu 16.04.4 LTS
Kernel           :   Linux 4.4.0-121-generic (x86_64)
Computer Name    :   elcano
```

Figure 5.1.: Elcano specifications

5.3.1. Response times and sizes

This section contains tables and graph visualisation for response time and size data from the performance analysis.

	Results							
Observation count	**1**	**100**	**200**	**400**	**500**	**600**	**800**	**1000**
52North-SOS	1.25	114	228	456	570	684	912	1110
52North-SOS Helgoland Client	0.34	2.56	4.8	9.29	11.54	13.79	18.28	22.77
FROST	0.593	56	111	223	279	334	446	557
FROST (reduced)	0.567	8.55	16.64	32.8	40.88	48.96	65.12	81.28

Table 5.3.: Average HTTP response sizes in Kilobytes [kb] for observation retrieval from Postman

Table 5.2 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 1110 kb at 1000 observations.

Helgoland Client requests return the smallest file sizes, with returned files holding 22.77 kb at 1000 observations, though the files also contain the smallest amount of metadata. Standard FROST requests reach up to 557 kb at 1000 observations, which is reduced to 81.28 when reducing output files to contain only time stamps and values. *Figure 5.2* shows the file sizes growing at a linear rate.

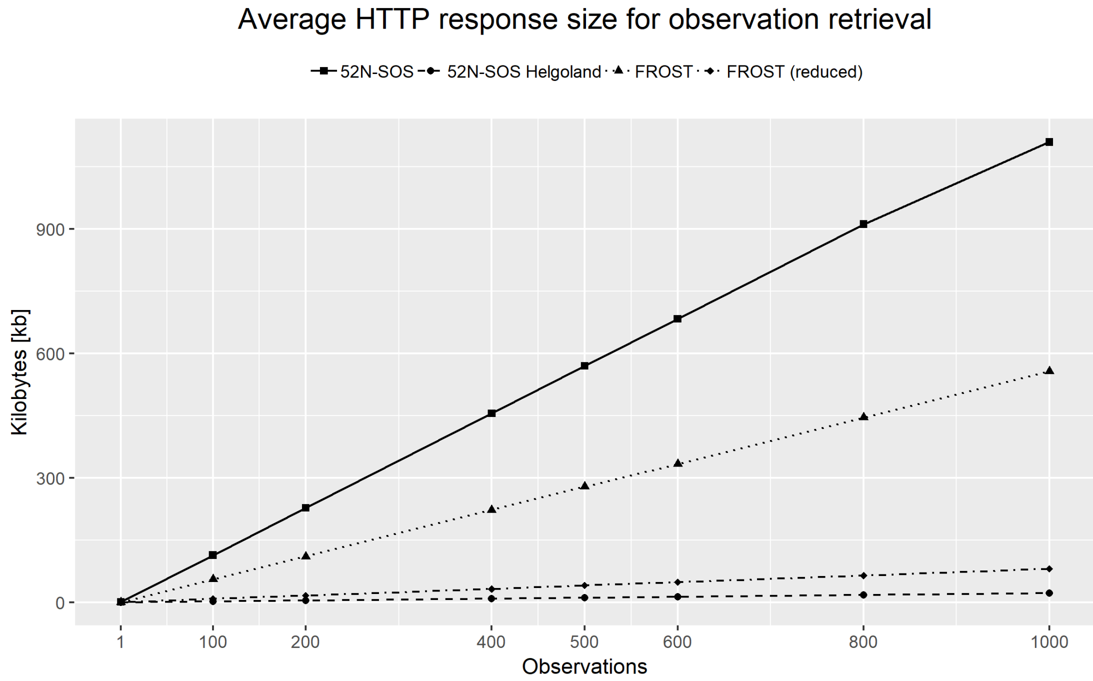


Figure 5.2.: Average Postman response sizes

The results in *Table 5.4* show the fastest response times for reduced FROST requests at an average speed 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. This becomes visible in *Figure 5.3*. 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, surpassing 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.

	Results								
Observation count	**1**	**100**	**200**	**400**	**500**	**600**	**800**	**1000**	**Avg**
52North-SOS	260	161	168	201	236	203	280	324	229
52North-SOS Helgoland Client	138	180	211	208	199	235	401	294	233
FROST	185	251	238	285	336	436	381	410	315
FROST (reduced)	169	150	189	204	228	253	280	272	218

Table 5.4.: Average HTTP Response Times in Milliseconds [ms] for observation retrieval from Postman

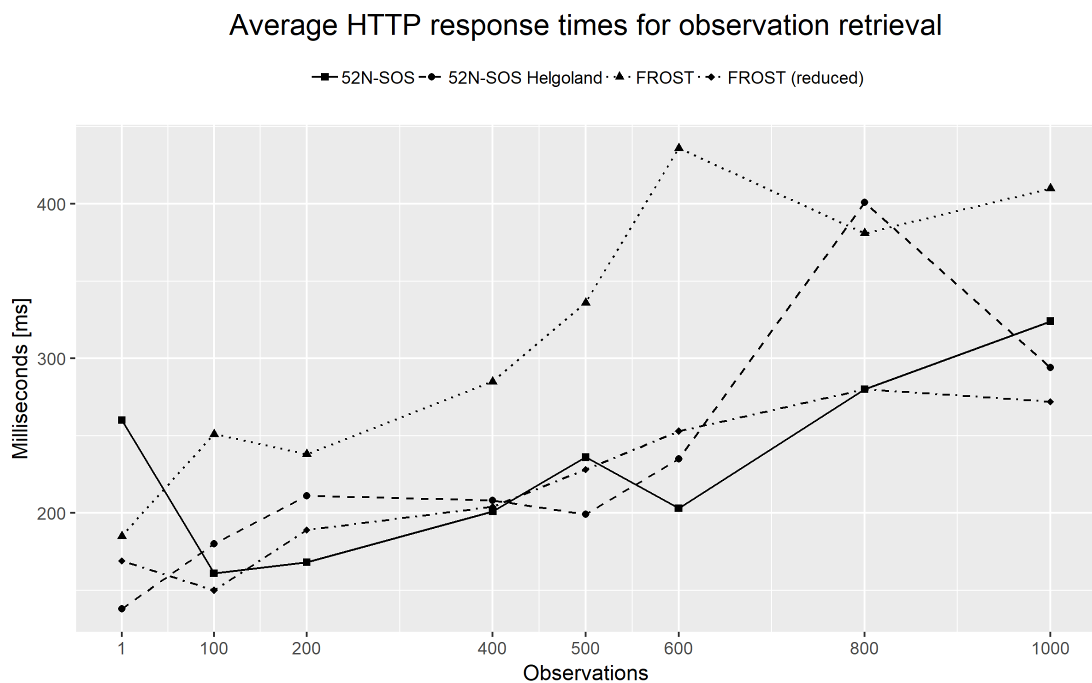


Figure 5.3.: Average Postman response times

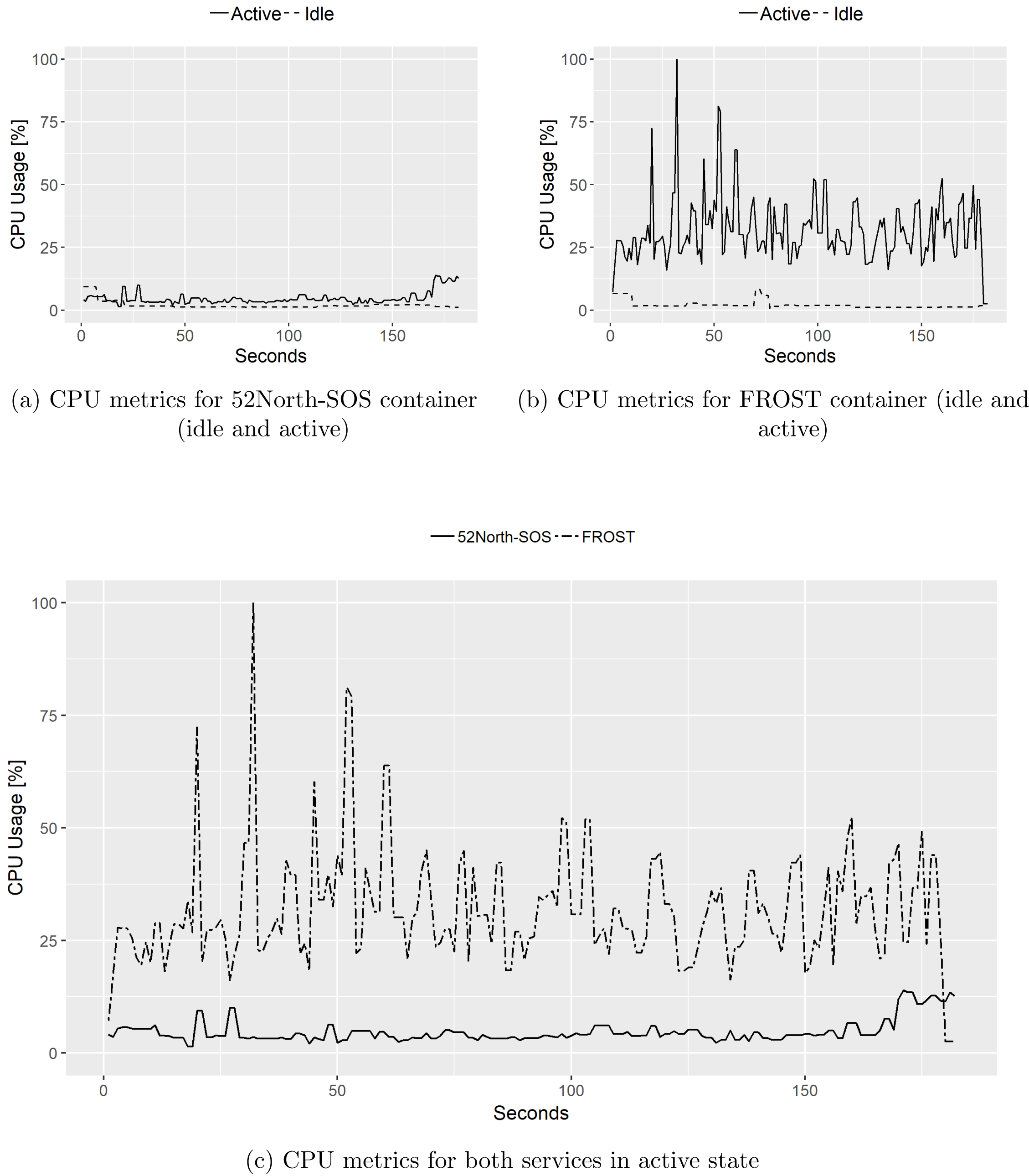
5.3.2. Container metrics

The query HTTP requests used for the response times and sizes analysis (see *5.3.1 Response times and sizes*) for FROST and 52North-SOS were configured in JMeter and executed in test runs of three minutes with approx. one request per second.

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 (see *4.4.3.2 Web service metrics*). *Figure 5.4* shows graph visualisations of the CPU metrics output for both services.

When regarding the individual services and comparing the CPU metrics in idle and active state in *Figures 5.4* (a) and (b), a CPU increase in both services becomes evident.
In FROST the difference is drastic, averaging 2.07% CPU in idle state in contrast to 31.80% during requests of 1000 observations, resulting in an average increase of 29.72%. The CPU maximum in FROST reaches almost 100%, while showing further spikes reaching between 50% and 75%. The FROST CPU maximum is almost 70% higher than the average in active state.
In 52North-SOS the difference in CPU usage is more moderate, holding an average of 1.96% when idle and an average of 4.75% when active, producing an average increase of 2.80%. Maximum values for 52North-SOS reach 13.97%, 9.22% higher than the average during requests.

Plotting the results for both services in active state (*Figure 5.4* (c)) shows FROST's higher CPU usage becomes more evident. While holding similar CPU values in idle state, FROST 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%.



(a) CPU metrics for 52North-SOS container (idle and active)

(b) CPU metrics for FROST container (idle and active)

(c) CPU metrics for both services in active state

Figure 5.4.: Above: Plots from 52North-SOS and FROST container CPU usage in idle state and during requests of 1000 observations; Below: CPU metric behaviour from both services during requests of 1000 observations

The results from the previous plots are mostly reflected in the CPU usage values at different response sizes. *Figure 5.5* shows the average CPU values of both FROST and 52North-SOS increasing amounts of requested observations. FROST average CPU usage shows all values for different response sizes between 27.8% and 38.%. In 52North-SOS, average CPU values lie between 4.06% and 5.94%. 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.

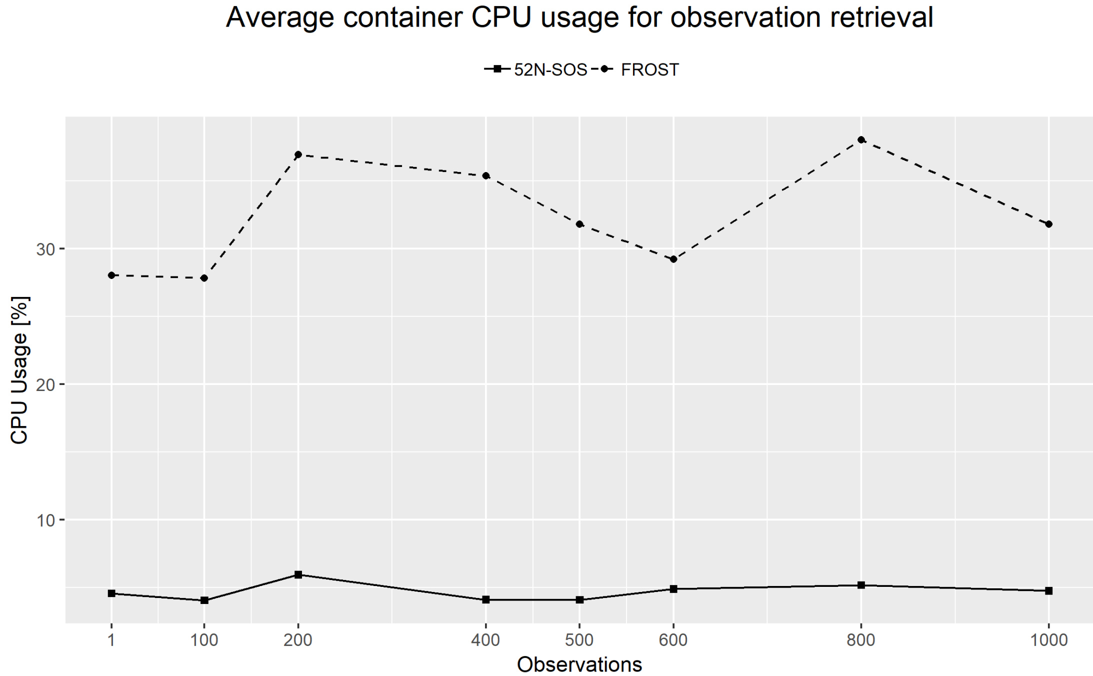


Figure 5.5.: Average CPU usage for different response sizes in FROST and 52North-SOS

To sum up the CPU usage behaviour for FROST and 52North-SOS, both services show a noticeable activity during constant observation requests. The usage does not change with increasing response sizes up to 1000 observations though. Overall, 52North-SOS shows a lot lower and more stable CPU values.

In idle state, 52North-SOS uses approx. 2572MB RAM, while FROST requires 3116MB. These values include the RAM for the corresponding database containers, which may increase with a growing amount of data. The graphs in *Figure 5.6* show the RAM behaviour under request activity of 1000 observations. Both graphs show very little activity during the monitoring run, though the average values are higher than in idle state. FROST average memory usage in active state lies at 3363MB, which results in an increase of 247MB. Similarly, 52North-SOS memory usage lies at 2741.7MB, incrementing RAM usage by 170MB.

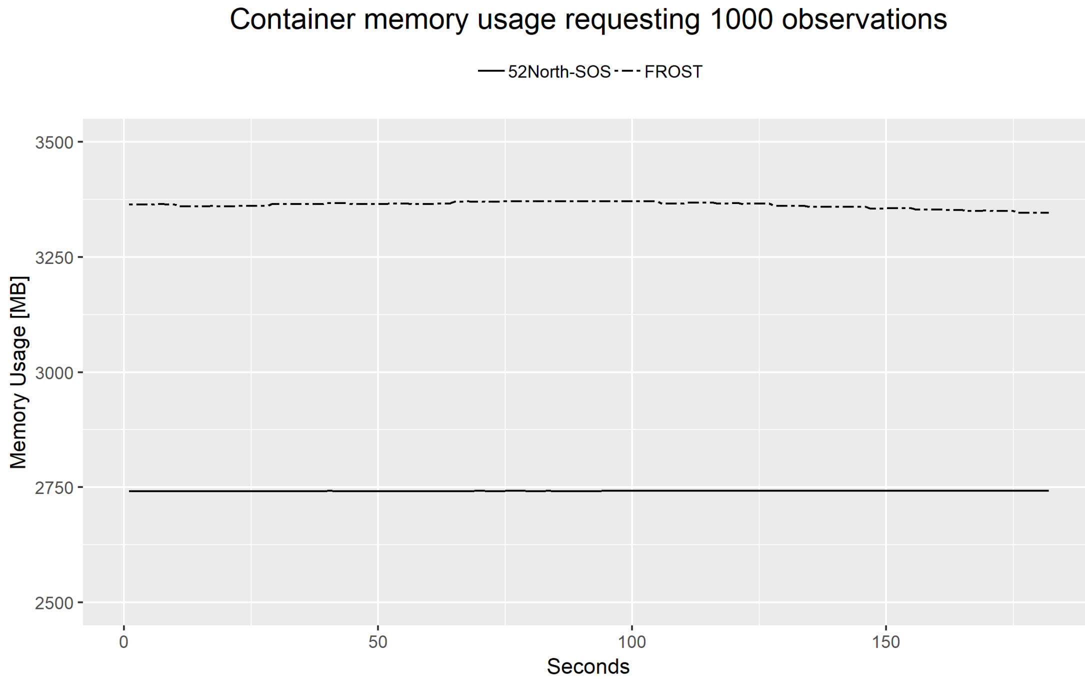


Figure 5.6.: Memory metrics for 52North-SOS and FROST requests of 1000 observations

When looking at the RAM behaviour during different observation requests, average values do not show significant changes reflecting the response sizes. 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.

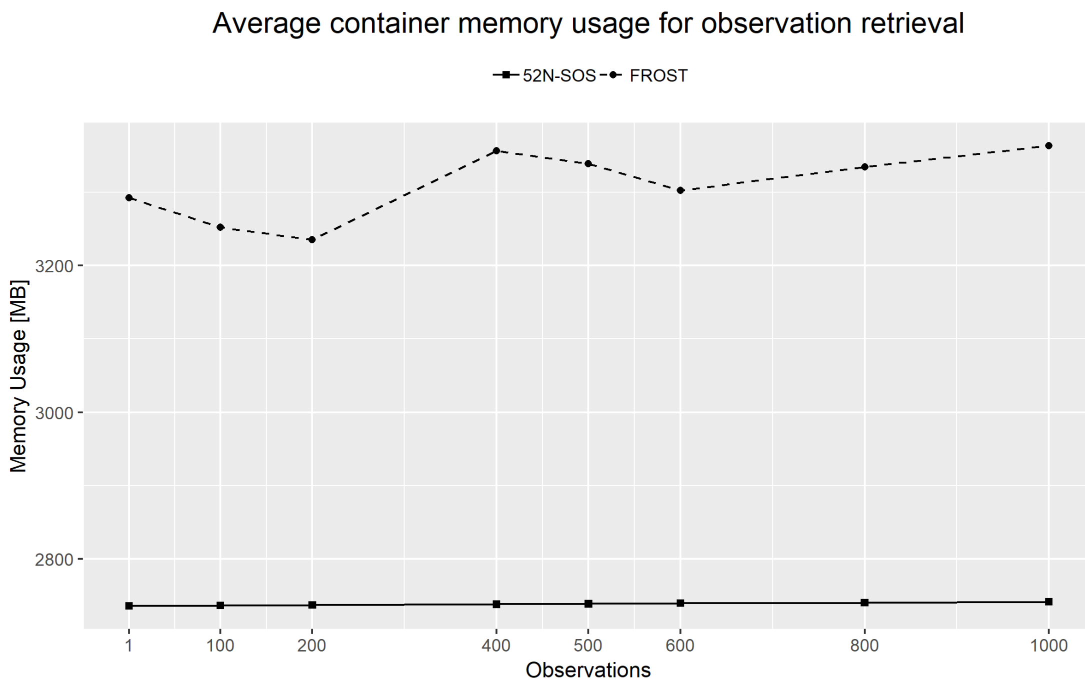


Figure 5.7.: Memory usage for different request sizes in FROST and 52North-SOS

5.4. Discussion

In the scope of this project the Web Thing API, 52North-SOS and FROST have been intensively tested and experimented upon by integrating them into SEnviro project. In this section of the document the application deployment process, their flexibility and their adaptability are discussed. Furthermore, the previously presented analysis results of qualitative and quantitative methodologies are used to determine the aptitude of the Web Standard implementations in SEnviro and other potential Smart Farming applications. The insights from the discussion are also compared to other related projects, adding value to the analysis.

The Web Thing API is an emerging Web Standard and has been in development since early 2018. The Web Standard is still in its early days and is still very limited in several aspects. Its features indicate that it is meant more as an IoT support system for household devices. The service does not include operations to insert new devices once the server is running, thereby lacking an essential feature for applications like SEnviro. Moreover, the Web Standard does not include support for storing observations, which further makes it unapt for time series data analysis. Nevertheless, the Web Thing data model and its sensing and tasking capabilities can be adapted for simple environmental monitoring applications. The adapter created in this project updates the property values of the things inserted during the script execution with the values from the SEnviro MQTT messages. The Web Thing API includes action and event classes, which are designed for the inclusion system alerts when device properties reach certain levels. This opens possibilities for monitoring applications that inform users about the environment in the device's environment in real time. There is also potential for combining the action and event operations for process automation within a system of devices.

There have been efforts by the Mozilla IoT community to extend the Web Thing API for storing and visualising time series data. (Bobin, 2018) developed an application that connects to things within the Web Thing API, reads their properties and stores the values in a Prometheus[1] time series database. Prometheus includes a dashboard for graph visualisation where data can be plotted over time. Although this implementation of the Web Thing API does show promising results for environmental monitoring applications, it does not address the lack interoperability within the IoT as long as it is not included as an official feature into the Web Standard.

52North started developing its SOS implementation in 2010. Over the past 9 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

[1]https://prometheus.io/ (Accessed 19.02.2019)

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 an extremely popular encoding format in recent years due to its simple syntax and easy serialisation to JavaScript variables, making it more compatible with present web browsers. Further arguments that favour the usage of JSON are that it has little overhead and that less band width is required to transmit messages (Tamayo et al., 2012). 52North-SOS supports JSON format for all the SOS core and transactional operations and most of the enhanced SOS operations.

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. Requesting observations also requires posting an object containing the query parameters, while in FROST this can be done with a GET request to the target URL using the API's query extensions. Request output is also larger in SOS for the most part than it is in FROST GET requests, leading to a general inflation of data traffic. A list of the different operations, their request type input and output sizes is shown in the *Table 5.5* .

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 own 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 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 very large spectrum of tasks, many of which are not necessary for IoT environmental monitoring applications like SEnviro, where devices run with limited resources.

Pradilla et al. (2015) 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 (Open Geospatial

[2]https://github.com/Juanvx/SOSLite (Accessed 19.02.2019)

Consortium, 2014) and aiming to adapt SOS to IoT scenarios. The results show an improvement in response times for several SOS operations. In SOSFul[3] (Pradilla et al., 2018), 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 SEnviro during the SOS implementation selection process. 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.

In terms of performance 52North-SOS *getObservation* requests outperformed FROST 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 5.5*). Furthermore, 52North's Helgoland client uses an API that improves response speed even further by reducing output information.

	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 5.5.: 52North-SOS and FROST request types and approximated input and output object sizes in bytes for SEnviro

As the name implies, the development of SensorThings API is grounded on the requirements of of an IoT environment. SensorUp[4], founded by Steve Liang, one of the chief developers of SensorThings, has published several research papers comparing SOS and SensorThings and describes the differences between the Web Standards in an article on their web site (SensorUp, 2016). The developers claim they designed

[3]https://github.com/Juanvx/SOSFul (Accessed 19.02.2019)
[4]https://sensorup.com/ (Accessed 19.02.2019)

the API to provide better scalability, performance, discoverability, real-time capability and developer experience.

A feature that strongly favours SensorThings over other standards is its data publish/subscribe support via MQTT, avoiding heavier 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 preconfiguration 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 definitely be considered in future work.

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 data model and API makes inserting data extremely flexible. 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 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 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 showing off its interoperability.

In terms of response times FROST 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 "resource hungry" than FROST, both in idle as in active states. This did come as a surprise, since the FROST 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 and therefore performance could be optimised.

Unmodified response sizes from observation requests are smaller in FROST. This promises reduced data traffic when requesting sensor data, for example in web applications for visualising large amounts of observations. As mentioned in *4.4.3.2 Web service metrics*, to make a tangible quantitative comparison requests were limited to 1000 observations within the performance analysis. Attempts to retrieve larger amounts of observations were made though. 52North-SOS would frequently freeze or crash when requested amounts of data were too large. FROST 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 and 52North-SOS CPU and memory usage do not increase proportionally to the amount of observation requests. However, this may change when requesting larger data sets.

Although the performance analysis does not favour FROST 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 and 52North-SOS at the mere Web Standards, the SensorThings API's data model and operations comply more fully with the IoT concept and and with the requirements of current environmental monitoring and Smart Farming applications.

Conclusion & Future Work

In this study the potential of Web Thing API, Sensor Observation Service (SOS) and SensorThings API open standards for interoperability enhancement of environmental monitoring and Smart Farming IoT applications has been investigated. Implementations of each standard were deployed (52North-SOS, FROST) or developed (Web Thing API), tested and results were contrasted in terms of performance and web service quality. The experiment was done in the scope of the *SEnviro for Agriculture* project, an IoT full stack for monitoring vineyards.

Due to its various limitations and and lacking support for time series data the Web Thing API was discarded as candidate for SEnviro interoperability enhancement. The Web Thing API can be used for simple real-time monitoring of environmental parameters and shows high configurability. Future work could include extending the standard to include time series data support.

Investigating SOS it was concluded to be outdated in a lot of aspects, lacking support for modern web technology trends, such as the use of JSON, RESTful binding and MQTT. For these reasons the well-established SOS implementation 52North-SOS was deployed. 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. What 52North-SOS adds in functionality it lacks in flexibility and scalability, which has a strong impact on the developer and user experience. Furthermore, the myriad of configurations and settings in the client interface render the software overwhelming. Since SEnviro has it's 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 implements the complete SensorThings 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 SOS, performance issues can easily be overcome by using URL extensions to select only the required data, maintaining a low overhead.

This study did not only answer 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. Secondly, the potential of the Web Thing API extension with Prometheus time series databases should be considered researching, since it might be a feasible solution for different types of environmental monitoring applications. 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. FROST should also be further experimented upon, focusing on its MQTT publish/subscribe capabilites, since they could bypass resource intensive HTTP requests for publishing observations in the service.

Bibliography

Anastasi, G., Farruggia, O., Re, G. L., and Ortolani, M. (2009). Monitoring high-quality wine production using wireless sensor networks. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–7.

Bassi, A. and Horn, G. (2008). Internet of Things in 2020: A Roadmap for the Future. *European Commission: Information Society and Media*, 22:97–114.

Bobin, J. (2018). Visualizing Your Smart Home Data with the Web of Things. https://hacks.mozilla.org/2018/05/visualizing-your-smart-home-data-with-the-web-of-things/. Accessed: 2019-02-02.

Burrell, J., Brooke, T., and Beckwith, R. (2004). Vineyard computing: Sensor networks in agricultural production. *Pervasive Computing, IEEE*, 3:38– 45.

Collina, M., Corazza, G. E., and Vanelli-Coralli, A. (2012). Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. *IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC).*

Derhamy, H., Eliasson, J., and Delsing, J. (2017). IoT Interoperability—On-Demand and Low Latency Transparent Multiprotocol Translator. *IEEE Internet of Things Journal*, 4(3).

Devi, M., Reddy, M., and Reddy, G. (2018). Iot based agricultural system. *International Journal of Innovations & Advancement in Computer Science*, 7:412–416.

Fredericks, J. and Botts, M. (2018). 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*, 3(1).

Gridling, G. and Weiss, B. (2007). Introduction to Microcontrollers. Vienna University of Technology, Institute of Computer Engineering, Embedded Computing Systems Group.

Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*.

Huang, C.-Y. and Wu, C.-H. (2016). A Web Service Protocol Realizing Interoperable Internet of Things Tasking Capability. *Sensors*.

IEEE (1991). IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. *IEEE Std 610*, pages 1–217.

Kamilaris, A., Gaoy, F., Prenafeta-Boldú, F. X., and Ali, M. I. (2016). Agri-IoT: A Semantic Framework for Internet of Things-enabled Smart Farming Applications. *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT).*

Kotamäki, N., Thessler, S., Koskiaho, J., Hannukkala, A. O., Huitu, H., Huttula, T., Havento, J., and Järvenpää, M. (2009). Wireless in-situ Sensor Network for Agriculture and Water Monitoring on a River Basin Scale in Southern Finland: Evaluation from a Data User's Perspective. *Sensors 2009.*

Latvakoski, J., Iivari, A., Vitic, P., Jubeh, B., Alaya, M. B., Monteil, T., Lopez, Y., Talavera, G., Gonzalez, J., Granqvist, N., Kellil, M., Ganem, H., and Väisänen, T. (2014). A Survey on M2M Service Networks. *Computers 2014*, 4:130–173.

Lazarescu, M. T. (2013). 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.*

Lee, C.-Y. and Lee, G.-B. (2005). Humidity Sensors: A Review. *IEEE Sensor Letters.*

Mainetti, L., Patrono, L., and Vilei, A. (2011). Evolution of Wireless Sensor Networks towards the Internet of Things: a Survey. *SoftCOM 2011, 19th IEEE.*

Mauri, J. L., Bosch, I., Sendra, S., and Serrano, A. (2011). A Wireless Sensor Network for Vineyard Monitoring That Uses Image Processing. In *Sensors.*

Moiş, G. D., Sanislav, T., Folea, S. C., and Zeadally, S. (2018). Performance Evaluation of Energy-Autonomous Sensors Using Power-Harvesting Beacons for Environmental Monitoring in Internet of Things (IoT). *Sensors.*

Open Geospatial Consortium (2007). *Sensor Observation Service.* Open Geospatial Consortium.

Open Geospatial Consortium (2014). *Best Practice for Sensor Web Enablement Lightweight SOS Profile for Stationary In-Situ Sensors.* Open Geospatial Consortium.

Open Geospatial Consortium (2016). *OGC SensorThings API - Sensing.* Open Geospatial Consortium.

Open Geospatial Consortium (2019). opengeospatial.org. http://www.opengeospatial.org. Accessed: 2019-01-14.

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

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

Reed, C. (2004). Data integration and interoperability: Iso/ogc standards for geo-information. *Development directions*.

SensorUp (2016). Comparison of SensorThings API and Sensor Observation Service – Part 1. https://sensorup.com/iot/comparison-of-sensorthings-api-and-sensor-observation-service/. Accessed: 2019-02-02.

Spann, T., Lawrence, C., Azzola, F., G.Simmons, D., Styger, E., and Smith, T. (2018). *The 2018 DZONE Guide to Internet of Things - Harnessing Device Data*, volume V. DZONE.

Sutaria, R. and Govindachari, R. (2013). 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)*.

Swan, M. (2012). Sensor Mania! The Internet of Things, Wearable Computing, Objective Metrics, and the Quantified Self 2.0. *Journal of Sensor and Actuator Networks*.

Tamayo, A., Granell, C., and Huerta, J. (2012). Using SWE Standards for Ubiquitous Environmental Sensing: A Performance Analysis. *Sensors 2012*.

Trilles, S., Gonzáles Pérez, A., Zaragoza-Soria, F. J., and Huerta, J. (2018). SEnviro for Agriculture: An IoT full stack for monitoring vineyards – Early steps. *AGILE*.

Trilles, S., Luján, A., Belmonte, O., Montoliu, R., Torres-Sospedra, J., and Huerta, J. (2015). Senviro: A sensorized platform proposal using open hardware and open standards. *Sensors*, 15(3):5555–5582.

Want, R., Schilit, B. N., and Jenson, S. (2015). Enabling the Internet of Things. In *2015 IEEE Computer, Volume: 48 , Issue: 1*. Google.

Weinberg, B. (2014). The Internet of Things and Open Source (Extended Abstract). In *Interoperability and Open-Source Solutions for the Internet of Things*, SoftCOM 2014.

Yick, J., Mukherjee, B., and Ghosal, D. (2008). Wireless sensor network survey. *Computer Networks*, 52:2292–2330.

Zhou, X., Li, Z., Li, S., and Ma, J. (2011). Connecting agriculture to the internet of things through sensor networks. In *2011 IEEE International Conference on Internet of Things and 4th IEEE International Conference on Cyber, Physical and Social Computing (iThings/CPSCom 2011)(ITHINGS/CPSCOM)*, volume 00, pages 184–187.

Appendix

The following code blocks can also be found on Github on https://github.com/danji90/senviro (Accessed 22.02.2019)

FROST-Server SEnviro Adapter:

```
1  import pika
2  import requests
3  import json
4  import sys
5  import ast
6  from datetime import datetime
7
8  # FROST-Server baseUrl
9  baseUrl = "http://elcano.init.uji.es:8082/FROST-Server/v1.0"
10
11 connection =
    ↪  pika.BlockingConnection(pika.ConnectionParameters(host='senviro.init.uji.es',
    ↪  credentials=pika.credentials.PlainCredentials(username='senvmq',
    ↪  password='senviro.2018')))
12 channel = connection.channel()
13
14 channel.exchange_declare(exchange='amq.topic',
15                          exchange_type='topic',
16                          durable=True)
17
18 result = channel.queue_declare(exclusive=True)
19 queue_name = result.method.queue
20
21 binding_keys = sys.argv[1:]
22
23 if not binding_keys:
24     sys.stderr.write("Usage: %s [binding_key]...\n" % sys.argv[0])
25     sys.exit(1)
26
27 for binding_key in binding_keys:
28     channel.queue_bind(exchange='amq.topic',
29                        queue=queue_name,
30                        routing_key=binding_key)
31
32
33 print(' [*] Waiting for logs. To exit press CTRL+C')
34
35 def insertObservation(nodeID, phenomenon, body):
```

```
36
37      # Datastreams base url
38      url = baseUrl + '/' + 'Datastreams'
39
40      # Load name and id of present datastreams
41      dataStreams = requests.get(url + '?$select=name,id').json()
42
43      # Empty variable for holding the target datastream id
44      sendingDatastreamID = None
45
46      # Get correct datastream id using the routing key parameters of the
         ↪ message, save it in variable
47      for stream in dataStreams['value']:
48          if stream['name'] == phenomenon+'-'+nodeID:
49              sendingDatastreamID = stream['@iot.id']
50
51
52      date = datetime.strptime(str(body['time']), '%Y-%m-%d %H:%M:%S')
53
54      # Create observation object to post
55      postObs = {"resultTime" :  str(date.isoformat()) ,"result" :
         ↪  float(body['value'])}
56      print(postObs)
57
58      # Post object to url/datastreams(id)/observations
59      try:
60          req = requests.post(url + '(' + str(sendingDatastreamID) + ')' + '/' +
             ↪  'Observations', json = postObs)
61          req.raise_for_status()
62          print(req, "####", phenomenon, " observation for station " + nodeID + "
             ↪  inserted at " + str(datetime.now().isoformat()))
63      except:
64          print(req, "####", "Could not insert observation at " +
             ↪  str(datetime.now().isoformat()))
65
66  def callback(ch, method, properties, body):
67
68      # Extract thing unique name and observable property from message routing
         ↪  key
69      thingName = str(method.routing_key).split('.')[2]
70      obsPropName = str(method.routing_key).split('.')[3]
71
72      # Decode byte message, throw error if decodification fails
73      try:
74          msg = ast.literal_eval(body.decode('utf-8'))
75      except:
76          print("Error: Message decodification failed, corrupted byte message
             ↪  (time: " + str(datetime.now().isoformat()) + ")")
```

```
    # Call insertObservation function, posts observations to frost db
    insertObservation(thingName, obsPropName, msg)

    # Message acknowledgement
    ch.basic_ack(delivery_tag = method.delivery_tag)

channel.basic_consume(callback, queue=queue_name)

channel.start_consuming()
```

52North-SOS SEnviro Adapter:

```
import pika
import requests
import json
import sys
import ast
from datetime import datetime

# SOS baseUrl
baseUrl = "http://sos:8080/52n-sos-webapp/service"

with open('thingsSOS.json') as json_data:
    things = json.load(json_data)

with open('insertObservation.json') as json_data:
    observationTemplate = json.load(json_data)

uoms = {"AirTemperature":"°C","Humidity":"%","AtmosphericPressure":"Pa",
"Precipitation":"mm","WindDirection":"","WindSpeed":"m/s",
"SoilTemperature":"°C","SoilHumidity":"m^3/m^3","Battery":"%"}

connection = pika.BlockingConnection(pika.ConnectionParameters(host=
'senviro.init.uji.es',credentials=pika.credentials.PlainCredentials(username=
'senvmq', password='senviro.2018')))
channel = connection.channel()

channel.exchange_declare(exchange='amq.topic',
                         exchange_type='topic',
                         durable=True)

result = channel.queue_declare(exclusive=True)
```

```
31 queue_name = result.method.queue
32 
33 binding_keys = sys.argv[1:]
34 
35 if not binding_keys:
36     sys.stderr.write("Usage: %s [binding_key]...\n" % sys.argv[0])
37     sys.exit(1)
38 
39 for binding_key in binding_keys:
40     channel.queue_bind(exchange='amq.topic',
41                        queue=queue_name,
42                        routing_key=binding_key)
43 
44 
45 print(' [*] Waiting for logs. To exit press CTRL+C')
46 
47 def insertObservation(nodeID, phenomenon, body):
48 
49     # Create timestamp from message
50     timestamp = str(datetime.strptime(str(body['time']), '%Y-%m-%d
    ↪  %H:%M:%S').isoformat())+"+00:00"
51 
52     # select correct thing from stations
53     coordinates = list(filter(lambda x: x["id"] == str(nodeID), things))
54 
55     # Create observation object to post
56     postObs = observationTemplate
57     postObs["offering"] = "offering"+str(nodeID)
58     postObs["observation"]["procedure"] = str(nodeID)
59     postObs["observation"]["observedProperty"] = phenomenon
60     postObs["observation"]["featureOfInterest"]["identifier"]["value"] =
    ↪  "featureOfInterest"+str(nodeID)
61     postObs["observation"]["featureOfInterest"]["name"][0]["value"] =
    ↪  str(nodeID)
62     postObs["observation"]["featureOfInterest"]["sampledFeature"] =
    ↪  ["parent"+str(nodeID)]                    # SampledFeature important for
    ↪  core operations
63     postObs["observation"]["featureOfInterest"]["geometry"]["coordinates"] =
    ↪  coordinates[0]["location"]
64     postObs["observation"]["phenomenonTime"] = timestamp
65     postObs["observation"]["resultTime"] = timestamp
66     postObs["observation"]["result"]["uom"] = uoms[str(phenomenon)]
67     postObs["observation"]["result"]["value"] = float(body["value"])
68 
69     # Post object to service url
70     try:
71         req = requests.post(baseUrl, json = postObs)
72         req.raise_for_status()
```

```
73
74            print(req, "####", phenomenon, " observation for station " + nodeID + "
           ↪  inserted at " + str(datetime.now().isoformat()))
75        except:
76            print(req, "####", "Could not insert observation at " +
           ↪  str(datetime.now().isoformat()))
77
78    def callback(ch, method, properties, body):
79
80        # Extract thing unique name and observable property from message routing
       ↪  key
81        thingName = str(method.routing_key).split('.')[2]
82        obsPropName = str(method.routing_key).split('.')[3]
83
84        # Decode byte message, throw error if decodification fails
85        try:
86            msg = ast.literal_eval(body.decode('utf-8'))
87        except:
88            print("Error: Message decodification failed, corrupted byte message
           ↪  (time: " + str(datetime.now().isoformat()) + ")")
89
90        # Call insertObservation function, posts observations to sos db
91        insertObservation(thingName, obsPropName, msg)
92
93        # Message acknowledgement
94        ch.basic_ack(delivery_tag = method.delivery_tag)
95
96    channel.basic_consume(callback, queue=queue_name)
97
98    channel.start_consuming()
```

WebThing SEnviro Adapter:

```
1   from asyncio import sleep, CancelledError, get_event_loop
2   from webthing import (Action, Event, MultipleThings, Property, Thing, Value,
3                         WebThingServer)
4   # from multiprocessing import Process
5   import logging
6   import random
7   import time
8   import uuid
9   import threading
10  import sys
```

```
11  import pika
12  import requests
13  import json
14  import ast
15  from datetime import datetime
16
17  ### WebThing setup
18  # Define Action class
19
20  class updateLocation(Action):
21      # Allows users to update a node's coordinates
22      def __init__(self, thing, input_):
23          Action.__init__(self, uuid.uuid4().hex, thing, 'updateCoords',
           ↪   input_=input_)
24
25      def perform_action(self):
26          self.thing.set_property('Coordinates', self.input['Coordinates'])
27
28  # Define class for things (nodes) with properties and available actions
29  class senviroNode(Thing):
30
31      def __init__(self, name, latlon):
32          type = []
33          description = "This monitoring station is an assemlby of sensors
           ↪  measuring environmental parameters in agricultural fields"
34          Thing.__init__(self,
35                         name,
36                         type,
37                         description)
38
39          self.add_property(
40              Property(self,
41                       'Coordinates',
42                       Value(latlon),
43                       metadata={
44                           '@type': 'LevelProperty',
45                           'label': 'Coordinates',
46                           'type': 'array',
47                           'description': 'Latitude and longitude coordinates of
                            ↪   the station',
48                           'unit': 'degree'
49                       }))
50
51          self.AirTemperature = Value(0.0)
52          self.add_property(
53              Property(self,
54                       'AirTemperature',
55                       self.AirTemperature,
```

```
56                     metadata={
57                         '@type': 'LevelProperty',
58                         'label': 'AirTemperature',
59                         'type': 'number',
60                         'description': 'The degree or intensity of heat
                            ↪  present in the area',
61                         'unit': 'degree celsius'
62                     }))
63 
64         self.Humidity = Value(0.0)
65         self.add_property(
66             Property(self,
67                      'Humidity',
68                      self.Humidity,
69                      metadata={
70                          '@type': 'LevelProperty',
71                          'label': 'Humidity',
72                          'type': 'number',
73                          'description': 'Ratio of the partial pressure of water
                             ↪  vapor to the equilibrium vapor pressure of water
                             ↪  at a given temperature',
74                          'minimum': 0,
75                          'maximum': 100,
76                          'unit': 'percent'
77                      }))
78 
79         self.AtmosphericPressure = Value(0.0)
80         self.add_property(
81             Property(self,
82                      'AtmosphericPressure',
83                      self.AtmosphericPressure,
84                      metadata={
85                          '@type': 'LevelProperty',
86                          'label': 'AtmosphericPressure',
87                          'type': 'number',
88                          'description': 'Pressure within the atmosphere of
                             ↪  Earth',
89                          'unit': 'pascal'
90                      }))
91 
92         self.Precipitation = Value(0.0)
93         self.add_property(
94             Property(self,
95                      'Precipitation',
96                      self.Precipitation,
97                      metadata={
98                          '@type': 'LevelProperty',
99                          'label': 'Precipitation',
```

```
100                             'type': 'number',
101                             'description': 'Product of the condensation of
                                 ↪  atmospheric water vapor',
102                             'unit': 'millimeters'
103                         }))
104
105             self.WindDirection = Value(0.0)
106             self.add_property(
107                 Property(self,
108                         'WindDirection',
109                         self.WindDirection,
110                         metadata={
111                             '@type': 'LevelProperty',
112                             'label': 'WindDirection',
113                             'type': 'number',
114                             'description': 'Wind direction in integer values
                                 ↪  (0-7)',
115                             'unit': 'null'
116                         }))
117
118             self.WindSpeed = Value(0.0)
119             self.add_property(
120                 Property(self,
121                         'WindSpeed',
122                         self.WindSpeed,
123                         metadata={
124                             '@type': 'LevelProperty',
125                             'label': 'WindSpeed',
126                             'type': 'number',
127                             'description': 'Wind speed readings in m/s',
128                             'unit': 'meters per second'
129                         }))
130
131             self.SoilTemperature = Value(0.0)
132             self.add_property(
133                 Property(self,
134                         'SoilTemperature',
135                         self.SoilTemperature,
136                         metadata={
137                             '@type': 'LevelProperty',
138                             'label': 'SoilTemperature',
139                             'type': 'number',
140                             'description': 'The degree or intensity of heat
                                 ↪  present in the soil',
141                             'unit': 'degrees celsius'
142                         }))
143
144             self.SoilHumidity = Value(0.0)
```

```
145         self.add_property(
146             Property(self,
147                      'SoilHumidity',
148                      self.SoilHumidity,
149                      metadata={
150                          '@type': 'LevelProperty',
151                          'label': 'SoilHumidity',
152                          'type': 'number',
153                          'description': 'Water content per soil ratio present
                          ↪  in the soil',
154                          'unit': 'm^3/m^3'
155                      }))
156 
157         self.Battery = Value(0.0)
158         self.add_property(
159             Property(self,
160                      'Battery',
161                      self.Battery,
162                      metadata={
163                          '@type': 'LevelProperty',
164                          'label': 'Battery',
165                          'type': 'number',
166                          'description': 'Battery readings in %',
167                          'unit': 'percent'
168                      }))
169 
170         self.add_available_action(
171             'updateCoords',
172             {
173                 'label': 'updateCoords',
174                 'description': 'Update station coordinates',
175                 'input': {
176                     'type': 'object',
177                     'required': [
178                         'Coordinates'
179                     ],
180                     'properties': {
181                         'Coordinates': {
182                             'type': 'array',
183                             'unit': 'degree'
184                         },
185                     },
186                 },
187             },
188             updateLocation)
189 
190 
191 # Create nodes for present stations, takes node ID and coordinates as
    ↪  arguments
```

```
192 node270043001951343334363036 = senviroNode("270043001951343334363036",
    ↪ [40.133098,-0.061000])
193 node4e0022000251353337353037 = senviroNode("4e0022000251353337353037",
    ↪ [39.993934,-0.073863])
194
195 ### Rabbitmq configuration
196 def rabbitReceiver():
197
198     connection = pika.BlockingConnection(pika.ConnectionParameters(host=
199     'senviro.init.uji.es',
        ↪ credentials=pika.credentials.PlainCredentials(username=
200     'senvmq', password='senviro.2018')))
201
202     channel = connection.channel()
203     channel.exchange_declare(exchange='amq.topic',
204                              exchange_type='topic',
205                              durable=True)
206
207     result = channel.queue_declare(exclusive=True)
208     queue_name = result.method.queue
209
210     binding_keys = ['#']
211     channel.queue_bind(exchange='amq.topic',
212                        queue=queue_name,
213                        routing_key=str(binding_keys[0]))
214
215     print("Waiting for messages...")
216
217     def callback(ch, method, properties, body):
218
219         # Decode byte message, throw error if decodification fails
220         try:
221             msg = ast.literal_eval(body.decode('utf-8'))
222         except:
223             print("Error: Message decodification failed, corrupted byte message
                ↪ (time: " + str(datetime.now().isoformat()) + ")")
224
225         thingName = str(method.routing_key).split('.')[2]
226         obsPropName = str(method.routing_key).split('.')[3]
227
228         node = globals()['node'+thingName]
229
230         try:
231             node.set_property(obsPropName, float(msg['value']))
232             print("inserted: ", thingName, obsPropName, float(msg['value']))
233         except:
234             print("Could not insert ", obsPropName, " values for ", thingName,)
235
```

```
236
237         ch.basic_ack(delivery_tag = method.delivery_tag)
238
239         # return updateMsg(dummy)
240
241     channel.basic_consume(callback, queue=queue_name)
242
243     channel.start_consuming()
244
245 # Define RabbitMQ connection as simultaneous thread
246 mq_recieve_thread = threading.Thread(target=rabbitReceiver)
247
248 def run_server():
249
250     # Define webthing server, add the two predefined nodes
251     server = WebThingServer(MultipleThings([node270043001951343334363036,
    ↪   node4e0022000251353337353037],'senviroWeb'),port=5000)
252
253     # Start RabbitMQ thread
254     mq_recieve_thread.start()
255
256     try:
257         # Start WebThing server
258         print('starting the server')
259         server.start()
260     except KeyboardInterrupt:
261         logging.debug('canceling the sensor update looping task')
262         # node270043001951343334363036.cancel_update_level_task()
263         print('stopping the server')
264         server.stop()
265         print('done')
266
267 # Run server
268 run_server()
```