Master thesis
An OGC Sensor Observation Service for GPS and mobile sensors

Roland Müller

B-IT Master Studies Autonomous Systems

University of Applied Sciences Bonn-Rhein-Sieg

Fraunhofer Institute for Intelligent Analysis and Information Systems

IAIS Advisor: PD Dr. Michael Mock

FH Professor: Prof. Dr. Paul Plöger

June 30, 2010

I, the undersigned below, declare that this work has not previously been submitted to this or any other university, and that unless otherwise stated, it is entirely my own work.

DATE

Roland Müller

Contents

Nomenclature

AES	Advanced Encryption Standard
API	Application Programming Interface
CSV	Comma-Separated Values
DES	Data Encryption Standard
DOM	Document Object Model
DTD	Document Type Definition
EPSG	European Petroleum Survey Group
ER	Entity Relationship
ESS	Emergency Support System
FOI	Feature Of Interest
GIS	Geographic Information System
GML	Geography Markup Language
GPRS	General Packet Radio Service
GPS	Global Positioning System

GPX	GPS Exchange Format
GUI	Graphical User Interface
IDE	Integrated Development Environment
IMEI	International Mobile Equipment Identity
ISO	International Organization for Standardization
JAXB	Java Architecture for XML Binding
JAXP	Java API for XML Processing
JCE	Java Cryptographic Extension
JDBC	Java Database Connectivity
JSR	Java Specification Request
MD5	Message-Digest Algorithm 5
MTJ	Mobile Tools for Java
O&M	Observation & Measurements
OCI	Oracle Call Interface
OGC	Open Geospatial Consortium
OWS	OGC Web Services Common Specification
RDBMS	Relational Database Management System
RSA	Rivest, Shamir & Adleman
SATSA	Security and Trust Services API for J2ME

SAX	Simple API for XML
SensorML	Sensor Markup Language
SHA	Secure Hash Algorithm
SOAP	Simple Object Access Protocol
SOS	Sensor Observation Service
SQL	Structured Query Language
SRID	Spatial Reference System Identifier
StAX	Streaming API for XML
SWE	Sensor Web Enablement
TML	Transducer Markup Language
TPTP	Eclipse Test & Performance Tools Platform Project
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
URN	Uniform Resource Name
WMS	Web Map Service
WST	Web Standard Tools
XML	Extensible Markup Language
XSD	XML Schema Definition

1. Introduction

The "Emergency Support System" (ESS) project[1] is developing an emergency management system whose architecture ideally should comply with the OGC ("Open Geospatial Consortium") standards. Mobile sensors (e.g. mobile GPS-enabled sensor stations for toxic values) will provide emergency case related information to a central back-end. The ESS system should then assist in providing the collected data through a portal to the emergency operator.

This work has been partly funded under the EU ESS project, which has the contract number 217951.

1.1. Motivation

As the focus of this work is positioned in the environment of the "Sensor Web", we first describe this term, followed by a task definition.

1.1.1. Sensor Web

The term "Sensor Web" was first introduced in 1997 by a NASA employee, defining a collection of distributed environmental sensors, which conjointly appear as one single unit. There should be a standardized access for publishing and requesting sensor data

[1]`http://www.ess-project.eu/`

via the World Wide Web, enabling a high interoperability of different heterogeneous sensor networks.

For defining sensor descriptions, data exchange formats and interfaces to access this data, standard documents had to be created. These are housed by the “Open Geospatial Consortium” (OGC) under the umbrella brand “Sensor Web Enablement” (SWE). Main idea of the SWE is a dynamic system in which new sensors can be added during runtime, and whose measurement data is requestable in real-time through standardized web services. The sensor metadata should be human-readable by employing XML encodings, and measurement data be geo-located and time-dependent for allowing client software to do data processing without a priori knowledge of the systems components. Further the sensors shall act autonomously, and the system should be able to handle alerts which can be triggered if measurements exceed certain thresholds. [4]

1.1.2. Task

The task of this thesis is to develop an OGC-compliant Sensor Observation Service (SOS) – a component of the SWE – for GPS related sensor data in this context. It should, in contrast to existing implementations, support full mobility of the sensors and be configurable with respect to adding different kinds of sensors. In particular, mobile phones should be considered as sensors, which transmit their data to the SOS server through the transactional SOS interface.

The SOS specification is split into 3 sub-profiles, from which according to the specification the core profile is mandatory. Furthermore the transactional profile is required for allowing the registration of new GPS sensors and the insertion of new measurements in the current scenario. It has to be evaluated, which functions of the enhanced profile are needed, and which ones of the upcoming version 2.0 of the specification are required for enabling non-stationary sensors.

For the description of the sensors, in this case mobile phones, a SensorML (Sensor Markup Language) model must be used to fulfil the SOS standard. For the same reason, the transmission of the measured values has to follow the "Observation & Measurements" (O&M) XML schema.

On the mobile phone side, a sensor client software must be implemented, which is capable of registering at the SOS and sending measurements in O&M format as well as transmitting GPS coordinates as raw data.

To retain privacy, user-dependent access permissions on sensor data should be set up and the transmission of data be encrypted.

After the implementation it should be measured experimentally, to what extent the OGC compliance – in particular the data transmission formats – induces an overhead compared to existing data formats such as GPX or binary formats. The aim is to determine the scalability between both approaches on a large number of sensors. Furthermore the server should be able to filter the data for offering measurements of certain regions and time periods.

1.2. Application scenarios

Being involved with the Emergency Support System (ESS) project, the primary application scenario for a mobile enabled Sensor Observation Service is already predetermined. But aside from that, other use cases are also possible.

1.2.1. Emergency Support System

The ESS project was started in June 2009 with the goal of setting up a portable crisis management system. Stationary or mobile sensor platforms like UAVs (Unmanned Aerial Vehicles) collect environmental sensor data which is sent in real-time to a central server. The fusion of the collected data may then assist the system operator in deciding on how to setup a rescue plan [1]. Due to the instantaneous access to the sensor's measurements, the operator can immediately detect changes in the environment and refine his decisions.

Currently being in the proof of concept phase, the capabilities of the project partner's components and their integration were demonstrated during a field test in June 2010. As the collected environmental measurement data is currently stored in CSV files and then spread by using a FTP server, a better integration of the components by employing a Sensor Observation Service is highly desirable.

1.2.2. Fleet tracking

Another scenario is the tracking of vehicles. The buses of a public transportation company may send their current positions to the central web service. Linking the location data with traffic information, the passengers waiting at the bus stop can follow the bus position on their mobile phone and get an estimation of its arrival time.

In non-public freight forwarding companies, the current position of a truck shall not be disclosed to third party persons. Therefore an encryption of the location data is desirable.

1.3. State of the art

According to the OGC web site there are 9 organizations which offer products that implement the SOS standard [20]. For some of the implementations neither the source code nor a demo application are available, for which reason they are not covered here.

1.3.1. Comparison of SOS implementations

The remaining implementations can be split into two groups: the ones which primarily implement the SOS/SWE, and the ones where the SOS rather is a side-product. To the first group we count the *OSCAR SOS* by *1Spatial Group Ltd.*[2], the *52° North SOS*[3] and the *OOSTethys JAVA SOS Toolkit*[4]. In contrast to that, the *OSGeo project* offers *deegree*[5] as a Java framework for web-based spatial applications and *MapServer* [6] for the web-based display of maps. The last implementation mentioned here is the *Constellation* server by *Geomatys*[7] which has the goal to implement a framework for managing spatial data.

The aforementioned servers were analyzed for being adaptable to the requirements of the ESS project. Important properties are the programming language which they are written in and the kind of data storage they use. In regards of performance, there is also an interest in which XML parser they use to interpret the incoming requests. A comparison of these aspects is depicted in table 1.1.

The OGC-specific terms which are used in this section, will be described in the Basics chapter.

[2]`http://81.29.75.200:8080/oscar/`
[3]`http://52north.org/maven/project-sites/swe/sos/index.html`
[4]`http://www.oostethys.org/downloads/oostethys-toolkit-java`
[5]`http://www.deegree.org/`
[6]`http://mapserver.org/`
[7]`http://www.constellation-sdi.org/`

Developer	Name	Version	Open Source	Implements	XML proc.	Data source
1Spatial Group Ltd.	OSCAR	2.0	No	Core & Transactional	?	?
52° North	SOS	3.1.1	Yes, Java	Core & Transactional; GetFeatureOfInterest, GetResult	XMLBeans	PostgreSQL
OOSTethys	JAVA SOS Toolkit	0.4.3	Yes, Java	Core	XMLBeans	OPeNDAP, NetCDF
OSGeo	deegree	2.3	Yes, Java	Core	DOM	PostgreSQL, Oracle, JDBC
OSGeo	MapServer	5.6.3	Yes, C++	Core; DescribeObservationType	libxml2 XPath	file-based
Geomatys	Constellation	0.5.2	Yes, Java	Core & Transactional; GetResult	Geotoolkit	PostgreSQL

Table 1.1.: Comparison of different SOS implementations

1.3.2. 52° North SOS

As it seems to be the furthest developed implementation, the 52° North implementation was analyzed in more detail. First we tried to register a sensor with multiple phenomena to the SOS. While this was successful, the following *InsertObservation* request only was accepted, when there was not more than one phenomenon specified at a time.

Having a look at the database model of the server, it is fairly complex as it consists of 16 tables with 23 interconnections for the referential integrity.

Further the server does not support mobile sensors in an OGC-compliant manner. However there is already a page about an own mobile extension in the documentation wiki[8]. It is barely filled with content but references to a paper [23] and an *UpdateSensor* operation.

Paper discussion

In the paper, the current O&M model is extended by two elements, namely a *SamplingFeature* and a *DomainFeature*. The former represents the position where the measurement took place, the latter describes the area in which the sensor is moving (e.g. a lake). This is realized by making modifications to the SOS operations: for the core profile, a time parameter is added to the *DescribeSensor* request which enables the client to retrieve the sensor's position at a given timestamp. *GetObservation* is extended by new *SampleFeature* and *DomainFeature* filters and additionally, an *UpdateSensor* operation was invented.

The corresponding *UpdateSensor* example in the wiki includes a link to a non-existing schema definition file in the header, making further investigation of the allowed content difficult. In the body, the updated parameters are the timestamp, current position and self-defined features including the "domain feature" and Boolean values which indicate

[8] `http://52north.org/twiki/bin/view/Sensornet/SosMobileExtension`

if the sensor is mobile and if it is active.

1.4. Problem formulation

Apart from the limitation that only one phenomenon may be contained in an *InsertObservation* request, the 52° North implementation and its extension approach come along with several problems.

First, the mobile sensor not necessarily knows its domain feature if it enters an area where it has not been before and which it was not expected to visit on launch. The server in contrast is able host a large feature database, and map the current sensor locations to domain features. It also could easily react to changing domains, being a growing lake due to a flood or a moving feature like an iceberg.

The second issue are the additional Boolean flags of the new *UpdateSensor* operation. If the sensor runs out of battery or if its data transmission fails, it is not able to set itself into an inactive state. For the server it would be possible to declare the state by calculating the passed time since the last received observation. Also the mobility property could be determined by comparing the last received positions of the sensor.

Another problem rather applies to the SOS standard than to its implementations. The *DescribeSensor* response returns the SensorML document which was published during a *RegisterSensor* operation, including the sensor's position. Updated location information of mobile sensors, which the server gets by *InsertObservation* requests, is not carried over to the SensorML document. Therefore the *UpdateSensor* operation was proposed, but for each new measurement of a moving sensor it requires the invocation of two operations instead of one. *UpdateSensor* is needed for updating the sensor position and *InsertObservation* for inserting the measured values for the phenomena.

Finally, the proposed extensions are not conform to the SOS 1.0 standard. It seems that the primary goal of the authors was not to point out how to extend the OGC standard to cope with mobile sensors, but how to extend the 52° North implementation for their reference project. This also results in bloating up its database model from 16 tables to 22 tables.

2. Basics

This section gives an introduction of the terms and standards which are relevant for the SOS implementation, beginning with a short XML introduction followed by the OGC standards which are built upon XML.

2.1. XML

The “Extensible Markup Language” (XML) is a meta language which is used to define other languages.

2.1.1. Structure & processing

Basically, an XML document is a simple text document, which facilitates the readability for humans and the interoperability between different machines and applications. By employing special markup tags, put into angle brackets, a classification of the actual content is possible. The content itself is housed in between two markup tags, an opening and a closing one, latter beginning with a slash (/) symbol. Additionally, the markup tags may contain further parameters called attributes. [9, p. 38]

A document for specifying a GPS location may look like the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<position coordinateSystem="WGS84">
  <latitude>50.7801</latitude>
  <longitude>7.1826</longitude>
</position>
```

Listing 2.1: Sample XML document

For the processing of XML documents, so-called parsers are used. They offer an interface for easing access to the input file in several programming languages, and they relieve the programmer of having to verify its structure.

There are two ways of checking the consistency of an XML document. Most XML parsers begin with verifying if it is *well-formed*. This procedure ensures that the document has a proper syntax: if every opened markup tag is closed again properly, if the nesting of the tags is correct, etc.

The second check is called *schema validation*. By setting up a DTD (Document Type Definition) or an XSD (XML Schema Definition) it is possible to construct a set of rules which declares constraints for the content of the XML file. [11] For the above example, the schema against which the document is validated could specify that the root element of the document always reads `position`.

Namespaces as they are known in programming languages like C++ are also available for XML. If an XML file relies on several XSD schemata, their namespaces can be specified as attributes of the root markup element. The further markup tags can then be distinguished by using prefixes.

2.1.2. XML parsers

For the interpretation and processing of incoming XML data, several different approaches exist which are splittable into three classes: DOM parsing, push parsing and pull parsing. [24, p. 33]

DOM

The DOM (Document Object Model) holds the whole XML document in memory, structuring it in a tree representation. This facilitates a random access to all elements, attributes and text contents of the document at the cost of memory usage. Therefore it is not suitable for very large documents. An advantage of DOM is the possibility of being able to modify the document or to create a new one, and to write it back into an XML file. DOM is – like the following both approaches – included in the "Java API for XML Processing" (JAXP) and offers direct access to the document content with functions like `getElementsByTagName()`, `getAttribute()` and `getValue()`.

Push

A push parser processes the document sequentially and generates events during the parsing operation. These events trigger callback functions in which the document data can be accessed. This has the advantage that the data processing can already begin without having to wait that the parsing of the whole file has finished. Also the document does not have to be kept in memory with the drawback that no random access to the content is possible. The most common API which implements this approach is called SAX (Simple API for XML), throwing events like `startDocument`, `endDocument`, `startElement`, `endElement` and `characters` for the text content.

Pull

The pull approach is also event-based, similar to push. Though here not the parser generates the events, but the application does by moving a cursor or an iterator. In opposite to the push model, pull also features the generation of XML, but there is no random access to the document elements possible, as it is with DOM. An API for pull is called StAX (Streaming API for XML) with methods like `hasNext()` and `next()` for checking the availability of tags and for navigating through them.

2.1.3. Binding

A completely different method for accessing XML documents is called binding. The JAXB (Java Architecture for XML Binding) has a special compiler `xjc` that reads XML schema definition files, from which corresponding Java objects are generated. During runtime, the XML document of interest is then mapped to instances of these objects, called unmarshalling. Also the opposite direction is possible by serializing an object back to XML (marshalling).

2.2. Relevant OpenGIS standards

The Open Geospatial Consortium (OGC) is a non-profit organization which was founded in 1994 with the aim to create standardized interfaces for the exchange of spatial information. The resulting standards are published under the trademark “OpenGIS”. With currently having more than 30 standards, only a subset of them is relevant for the implementation of a Sensor Observation Service. These are the standards for the SOS [19], the “Sensor Model Language” (SensorML) [18] the “Observations & Measurements” (O&M) [15] and the “OGC Web Services Common Specification” (OWS) [16].

Aforementioned standards are, together with further ones, part of OGC’s Sensor Web

Enablement (SWE)[1]. As the name already suggests itself, this initiative focuses on making it possible to exchange sensor data via web services.

Due to the SOS standard being the core component of this work, we devote it an own subsection.

2.2.1. Important nomenclature

For the better understanding of the standards which are described in the following, we first introduce the most important terms of the OGC-specific nomenclature that is defined in [15]:

- A *procedure* creates a measurement value, be it a real sensor or a simulated one
- The *observed property* or *phenomenon* specifies what is measured, for example water temperature, wind speed, ...
- An *observation* delivers the result of a measurement for an observed property at a certain point of time, e.g. $20° \ C$ or $50 \ km/h$
- Observations that are related to each other can be grouped into an *(observation) offering*
- The *feature of interest* is the object for which an observation is made, for example a lake
- The *URN* stands for "Uniform Resource Name" and is an unique identifier for a resource like a phenomenon or a coordinate reference system. It always begins with `urn:` followed by a namespace, which in our case reads `ogc`. Then sub-namespaces may follow and finally the actual resource name, a full example being `urn:ogc:def:property:OGC::AirTemperature`

[1]`http://www.opengeospatial.org/ogc/markets-technologies/swe`

2.2.2. Sensor Model Language

SensorML is a language for describing the characteristics of sensor systems. In the SOS application, it contains metadata information of the sensor, namely a unique sensor ID and optionally further data like manufacturer, model number or a contact person.

The more important information is the sensor's location, which is stored together with the corresponding coordinate reference system. The actual phenomena which the sensor is able to measure are stored in an *inputs* section. Mostly they are directly carried over to the *outputs* section, only appended by the unit of measurement. But an output may also be the result of pre-processed input data, for example if the sensor system is capable of combining two inputs (distance & time) into one output (speed).

2.2.3. Observations & Measurements

The purpose of O&M is the standardized XML encoding of observations. An observation begins with the sampling time in GML (Geography Markup Language) format [17], which usually encapsulates an ISO 8601 timestamp that consists of a date, the time and the time zone (e.g. `2010-05-31T12:00:00+02:00`). If the O&M document contains multiple measurements, also a time range can be specified by using two timestamps.

Subsequently, the procedure – that is the sensor ID – is written down, followed by the observed properties and the feature of interest the observation belongs to.

The final *result* section holds the actual content: the measurement values. If only one measurement should be delivered, for example the most recent one of a sensor, this can be done with a *DataRecord*. It has a subsection for each phenomenon, containing its quantity, unit of measurement and the actual value.

For multiple measurements, a *DataArray* is the element of choice. It first specifies the number of measurements, followed by a record with the quantity and unit of measurement for each phenomenon. Then the concatenated measurement values follow in a long string. It has a CSV-like format whose separators can individually be specified.

2.2.4. OGC Web Services Common Specification

From the OWS standard, the *GetCapabilities* operation is the most important, which will be described in the following SOS section. Further the *ExceptionReport* is needed to return a failure document, if the client put invalid parameters into its request or if the server has an error. The *ExceptionReport* consists of an exception code like "InvalidParameterValue" and a text that contains a more detailed error message. Optionally, the *locator* parameter can indicate the markup tag of the client's request, in which the error occurs.

2.3. Sensor Observation Service

The SOS is a web service which was invented to offer a standardized interface to request sensor metadata and real-time observations. Its operations are split into three profiles which are illustrated in table 2.1 and described below.

A request can either be sent via HTTP GET or via HTTP POST. The difference between them is how the parameters of the request are transmitted to the server. Using GET, the parameters of the request are appended to the URL of the server address. The POST method in contrast sends the parameters in the body of the HTTP message, which makes it easier to send larger amounts of data than with GET [8]. The root markup tag of a POST request must always contain the SOS operation, with the *service* attribute being "SOS".

It depends on the server which of the methods it implements. Most of them allow both POST and GET for the *GetCapabilities* request, whose response indicates which method(s) is/are allowed for the other SOS operations.

Profile	**Core**	**Transactional**	**Enhanced**
Type	mandatory	optional	optional
Operations	GetCapabilities	RegisterSensor	GetObservationById
	DescribeSensor	InsertObservation	GetResult
	GetObservation		GetFeatureOfInterest
			GetFeatureOfInterestTime
			DescribeFeatureType
			DescribeObservationType
			DescribeResultModel

Table 2.1.: Sensor Observation Service profiles & operations

Example documents for the operations that are described in the following can be found in the appendix, and should be read in parallel for gaining a better understanding.

2.3.1. Core profile

The core profile is the only mandatory profile, meaning that its three operations must be implemented in every SOS. A common sequence of communication between a web client and a SOS, consisting of the core profile requests and their appropriate responses, is shown in figure 2.1.

GetCapabilities

The first request that a client sends to a newly discovered SOS is *GetCapabilities*, which does not need any further parameters. It is also used for other OGC web services than SOS like the "Web Map Service" (WMS).

A service metadata document will be returned as the response, with the first sections giving information about the server in OWS format. This can include properties like

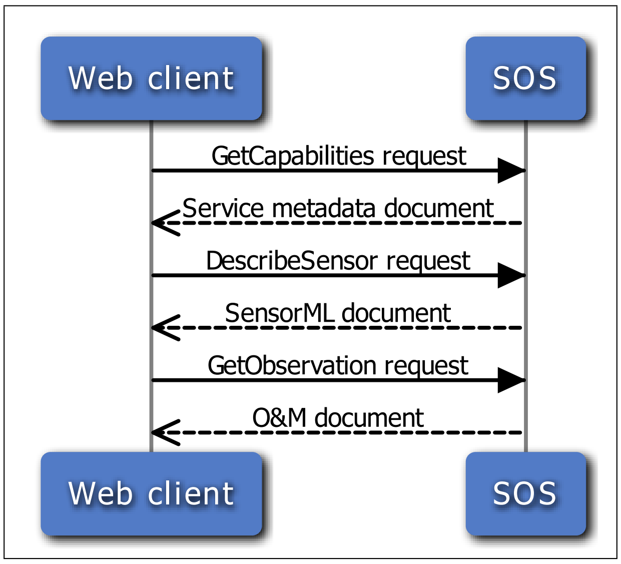


Figure 2.1.: Sequence diagram of core profile operations

server name, description, version and the responsible company with a contact person.

Among the following *OperationsMetadata* markup tag, all the operations the server supports are listed. It is denoted if the respective operation is invokable via POST and/or GET, which parameters are available and what they may contain. Taking *Get-Observation* as an example, its parameters include the available sensor IDs and the time range in which observations are available.

With the next section, the SOS-specific content of the capabilities begins. The filter capabilities describe, which kind of spatial, temporal and result filters the server supports. If implemented, this enables the client to request data of a certain region, a limited time range or a specific result set.

Finally, under the contents tag the observation offerings with their unique IDs are listed. Here we find abstract information like description and intended application as well as the observed properties and the spatial and temporal boundaries of the measurements.

DescribeSensor

The *DescribeSensor* request is used to get information about a particular sensor. It only contains the procedure, that is the sensor ID of which further information is desired. A list of valid sensor IDs is contained in the GetCapabilities response document. The *DescribeSensor* response is a SensorML document, whose content was already described above. Alternatively, the returned sensor description may be specified by using the "Transducer Markup Language" (TML)[2]. It is primarily used for streaming sensor data and not covered here in detail.

GetObservation

For requesting the actual measurements, the *GetObservation* operation is used. In its request, the indication of the offering and observed properties is mandatory. The client can get them from the SensorML response of a previously executed *DescribeSensor* request. Optionally, the procedure and spatial and temporal filters may be specified for *GetObservation*.

To the spatial filters, we count the bounding box, which enables us to request all observations within a rectangular area. Further it is possible to define a spatial filter area by specifying a closed polygon. The temporal filters allow the client to request observations *after*, *before*, and *equal* to a certain point in time, or *within* a time period.

In response to the *GetObservation* request we get an O&M document, which is again described above.

2.3.2. Transactional profile

All the other ones only fetching data from the server, the two operations of the transactional profile allow the insertion of new data into the SOS. Their implementation is

[2]`http://www.opengeospatial.org/standards/tml`

optional, as new data may also be inserted by using an other, proprietary method.

RegisterSensor

Beneath the root element of *RegisterSensor*, there are two sections. The *SensorDescription* contains a SensorML or TML document which specifies the sensor to be inserted into the SOS. Under the *ObservationTemplate*, an O&M template is specified which can be used for the following *InsertObservation* operation.

The response of *RegisterSensor* only consists of the *AssignedSensorId* tag, which contains the ID of the newly inserted sensor.

InsertObservation

The *InsertObservation* request needs a sensor ID for which the observation should be stored in the SOS and an observation in the already discussed O&M format.

In the response, there is only the *AssignedObservationID* generated by the SOS.

2.3.3. Enhanced profile

The enhanced profile contains further optional requests for querying data from the SOS. There is not paid further attention to them, as they are not required for our application. For completeness, they are listed in table 2.1 and can be looked up in [19].

2.3.4. Upcoming version 2.0

While this project focuses on the SOS 1.0 specification, the SOS 2.0 version is currently in development. So far, only OGC members are granted access to the draft documents. What already can be seen from them is that the specification will be split into several documents according to the different profiles.

The core profile seems to inherit the same operations from version 1.0, whereas the transactional profile now additionally allows the update and deletion of sensors. By enabling the possibility of updating the sensor metadata, it seems to be possible to refresh the sensor's position of the up to now static sensor description, allowing a better handling of mobile sensors.

Furthermore, all requests and responses will be embedded into SOAP envelopes. As there are no schema definitions available so far, there could only be speculated about more detailed improvements over version 1.0.

3. Development & testing environment

This section gives an overview of the tools which were used to implement the SOS and the secondary applications. The main programming language for the implementations is Java.

3.1. IDE

For the Java programming, the open source *Eclipse*[1] Integrated Development Environment (IDE) was employed. Due to its plug-in architecture, the functionality of the IDE can easily be enhanced.

An important plugin which was used is the "Web Standard Tools" (WST) for the integration of the *Apache Tomcat* servlet container. It enables the automatic deployment of the implemented servlets to a Tomcat server, facilitating the development. Similarly, the "Mobile Tools for Java" (MTJ) extension integrates the deployment of mobile phone applications to the phone emulators.

[1] `http://www.eclipse.org/`

3.2. Web server

After compiling the Java servlet application and packaging it to a web archive, it can be run on the free *Tomcat*[2] server by the *Apache Software Foundation.* Tomcat is written in Java, enabling the possibility to run it on a large variety of platforms, and consists of several components. The "Catalina" container runs the Java servlets and the "Coyote" connector delivers them to the clients using standard protocols like HTTP and HTTPS.

To achieve a better performance, the *Tomcat Native Connector* (TC-native) can be used. It does so by replacing some Java I/O components of the Tomcat server with ones that are written in C and optimized for the underlying operating system.

3.3. Database

The *Oracle* Relational Database Management System (RDBMS) is the flagship of the *Oracle* company. It exists in several versions with different features, beginning with the free "Express Edition" which is limited to use only 1 CPU, 1GB RAM and a maximum database size of 4GB. The smallest commercial version is the "Standard Edition One", which supports up to 2 CPUs and has no memory or space limitations. Up to 4 CPUs can be used with the "Standard Edition" which also supports clustering, enabling the possibility of spreading a database over several machines. The most features are available with the "Enterprise Edition" that does not have a CPU limitation and amongst others includes advanced security options and data compression. [10, p. 7]

For the Oracle RDBMS there is an extension called "Oracle Spatial" which enables spatial support for the database, following the OGC "Simple Features SQL" standard [14]. It facilitates the calculation of distances between locations, the selection of points that are located within a defined area, etc.

[2] `http://tomcat.apache.org/`

Important parameters for defining a spatial datum are the type of geometry (point, line, polygon), the "Spatial Reference System ID" (SRID) following the EPSG standard[3] (e.g. 4326 for WGS 84 without altitude or 4979 for WGS 84 with altitude) and the coordinates of the element which is specified. Some of the spatial queries need a special spatial index, which speeds up the data selection by using rectangular approximations instead of the exact shapes. A parameter called *tolerance* indicates, above which distance two points are regarded as being different. Its minimum value is 0.05 which equates to 5cm. The smaller the tolerance, the more processing power is required. A bigger tolerance decreases the precision and two different locations which are lying near to each other, may be misinterpreted as being the same. [6]

Two tools which were used for the database administration and development are the free "Oracle SQL Developer"[4] and the "Oracle SQL Developer Data Modeler"[5]. Both of them being platform-independent Java applications, the former is useful for operations like browsing database contents, testing SQL queries, modifying table structures and deleting table content. Latter was employed for designing a database model by drawing an entity-relationship diagram, which then can be exported to SQL `CREATE` statements.

3.4. XML editor & validator

Altova XMLSpy[6] is a commercial Windows application for editing XML files. From its large functional range, primarily the options for formatting XML files with indentation and the checks for well-formedness and schema validation were used. It is also helpful for creating new XML files from scratch, by specifying a schema definition file and then being offered code completion with incorporating namespaces that are specified in the

[3]`http://www.epsg-registry.org/`
[4]`http://www.oracle.com/technology/products/database/sql_developer/index.html`
[5]`http://www.oracle.com/technology/products/database/datamodeler/index.html`
[6]`http://www.altova.com/xmlspy.html`

header of the new file.

There is also an optional plugin for the Eclipse IDE that replaces the Eclipse XML validators with the XMLSpy implementation, which offers more features.

3.5. Mobile phones & emulators

The mobile phone emulators which were used during development are the one from the *Java ME SDK 3.0*[7] and the *Nokia S60 SDK for Symbian 3rd Edition FP2 v1.1*[8].

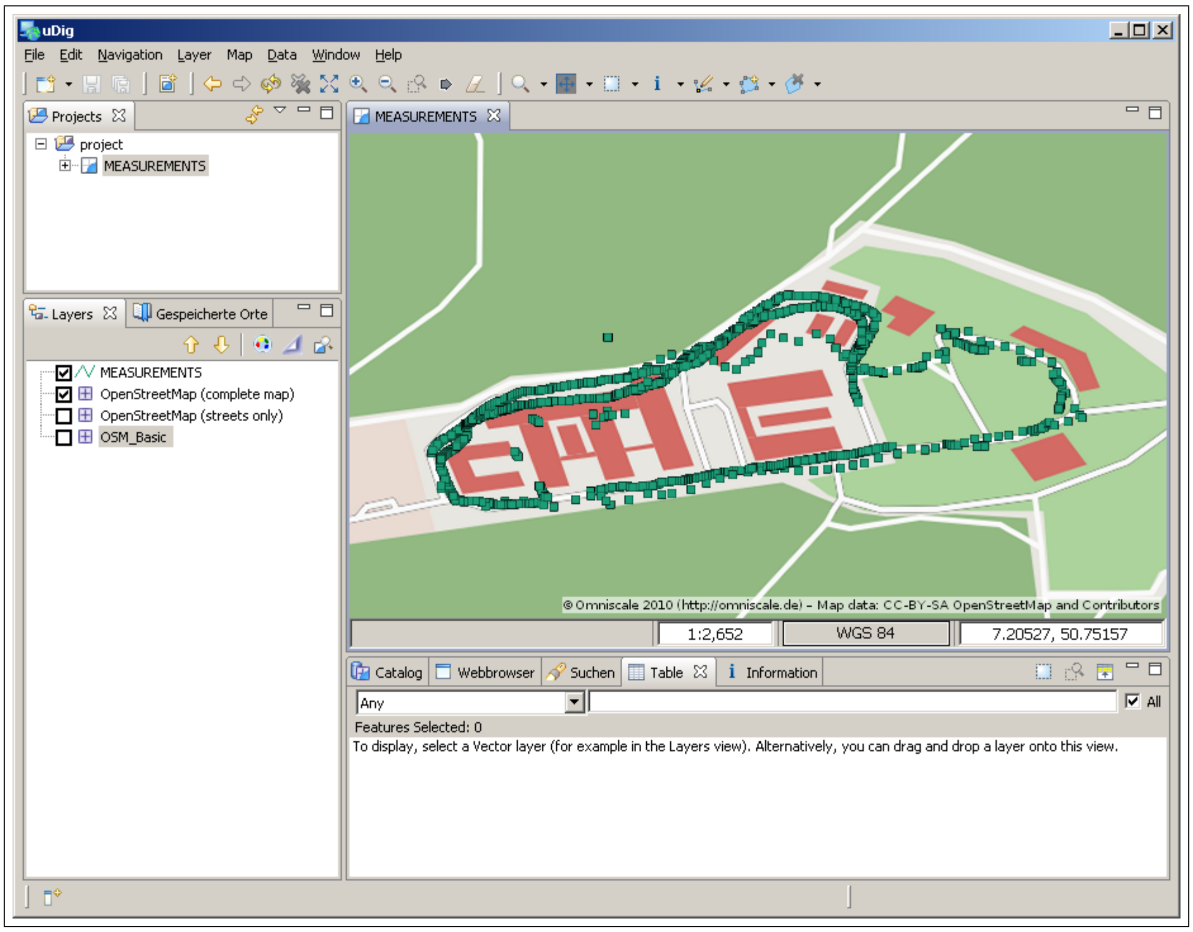
For the tests on a real hardware device, the *Nokia 5800 Xpress Music* and the *Nokia N95 8GB* were used. Both are running versions of *Symbian OS* which allow the execution of Java ME applications and they also implement the JSRs 179 (Location API for Java ME)[9] and 177 (Security and Trust Services API for J2ME)[10], which are needed for acquiring the current location from the integrated GPS device and for the demand to encrypt the measurement data before transmission.

3.6. GIS

The "User-friendly Desktop Internet GIS" (uDig)[11] is a geographic information system (GIS) and was used for the visualization of sensor data. As uDig is able to connect to an Oracle Spatial database and also offers a plugin for fetching sensor data from a SOS, it was helpful for a graphical verification of the database and the whole SOS functionality. Together with map data from a "Web Map Service" (WMS) like the University of Heidelberg is providing for free[12], it is directly visible if the sensor is

[7] `http://java.sun.com/javame/downloads/sdk30.jsp`
[8] `http://www.forum.nokia.com/Tools_Docs_and_Code/Tools/Platforms/S60_Platform_SDKs/`
[9] `http://jcp.org/en/jsr/detail?id=179`
[10] `http://jcp.org/en/jsr/detail?id=177`
[11] `http://udig.refractions.net/`
[12] `http://www.osm-wms.de/`

located where it should be. A screenshot of this application, drawing a self-recorded track which is fetched from an Oracle Spatial database, is shown in figure 3.1.


uDig
File Edit Navigation Layer Map Data Window Help
Projects
project
MEASUREMENTS
Layers
Gespeicherte Orte
MEASUREMENTS
OpenStreetMap (complete map)
OpenStreetMap (streets only)
OSM_Basic
MEASUREMENTS
© Omniscale 2010 (http://omniscale.de) - Map data: CC-BY-SA OpenStreetMap and Contributors
1:2,652
WGS 84
7.20527, 50.75157
Catalog
Webbrowser
Suchen
Table
Information
Any
All

Features Selected: 0

To display, select a Vector layer (for example in the Layers view). Alternatively, you can drag and drop a layer onto this view.

Figure 3.1.: uDig GIS

4. Requirements analysis

This chapter describes the scope of the project by setting the requirements which we want to achieve and by defining its limitations.

4.1. Desired functionality

In order to compare the overhead of the OGC-compliant XML encoding to a raw data transmission, two independent implementations should be made. Their specific functionality requirements are assigned below.

4.1.1. Sensor Observation Service

For the SOS implementation, these are the goals which were set:

- The server should best possible fulfill the OGC compliance
- All operations of the core and transactional profiles must be supported, namely: GetCapabilities, DescribeSensor, GetObservation, RegisterSensor and InsertObservation
- All operations must support the HTTP POST method, with GetCapabilities also supporting GET
- The implementation must cope with mobile sensors

- Mobile phones should be enabled to act as sensors
- It should be possible to handle multiple phenomena for each sensor
- In the DescribeSensor response, the latest sensor position should be contained
- The measured data set must be reducible by applying spatial & temporal filters
- Like in the 52° North implementation, a special keyword "latest" should be insertable in the GetObservation temporal filter instead of a timestamp, to receive the *latest* observation
- The implementation should adapt to existing server infrastructure, consisting of Apache Tomcat servlet containers and Oracle RDBMSs

4.1.2. Raw data transmission

The alternative raw data transmission aims on achieving the following:

- A small proof of concept client-server communication must be implemented without the focus on robustness and a pleasant GUI
- The data transmission should utilize user-dependent access permissions and encryption
- This application and the SOS should share the same database for storing the measurements

4.2. Constraints

The following constraints apply for the Sensor Observation Service:

- As the OGC standards are very extensive, our SOS implementation should primarily center on sensor templates for sensors of the IMEGO[1] company, which are part of the ESS project documents [7]
- For the SOS, user-dependent access permissions would be possible by using HTTP basic access authentication. In web browsers, this is known by offering a login window to the user. But as the SOS clients are mostly applications like GIS that do not support HTTP authentication, this approach is not feasible. Therefore the only security option for the SOS is to use HTTPS instead of HTTP, which is not a matter of implementation, but a configuration option of the Tomcat connector

[1]`http://www.imego.com/`

5. Concept

In this chapter, the design decisions of the application, its structure and the employed technologies are presented.

5.1. Enhancing an existing implementation

The most obvious method for building a Sensor Observation Service that supports mobile sensors would be the modification of an existing implementation.

Recalling the ones that were presented in the introductory section, only three of them were pure SOS implementations, with two also opening their source code (OOSTethys and 52° North). The OOSTethys server only implements the SOS core profile and uses a file-based database instead of a RDBMS, which already violates the desire to integrate it into the existing server environment. In contrast to that, the 52° North server additionally supports the transactional profile and uses a PostgreSQL RDBMS with the PostGIS extension, which enables the support of spatial operations.

Staying with the 52° North implementation, the *InsertObservation* function had to be extended as the aforementioned IMEGO sensor templates contain multiple phenomena and we want to keep the XML overhead the smallest possible.

Further a migration to the Oracle RDBMS would be very complex due to the high number of database tables and interconnections, which has been mentioned in the state of the art section.

Another aspect is the license of the 52° North SOS. The GPLv2 demands that modifications of the source code must be published if the application is spread. It would require an examination, if this is granted within the surrounding ESS project.

To sum it up, because of the complex database structure, the missing possibility of inserting multiple phenomena at a time, the lack of an OGC-compliant way to support mobile sensors and due to the licensing terms, the 52° North implementation does not provide an appropriate basis for our project. Consequently the decision was made not to base our implementation on the 52° North SOS, but to build up a new web service from scratch.

5.2. Application structure

To give a first overview of the whole application and its components, the structure is depicted in figure 5.1.

On the left side, the data source which creates new measurements is presented in the form of a mobile phone. It determines its current position from an integrated or external Bluetooth GPS device. On the phone, a Java application links the position data with further measurements, which it may get from an acceleration sensor. This observation is then either sent in an OGC-compliant way to the SOS via the InsertObservation request, or in form of encrypted raw data to an UDP or TCP server.

The SOS will be implemented as a servlet and run on an Apache Tomcat servlet container. It connects to an Oracle RDBMS for storing and retrieving the sensor metadata and measurement values.

For the alternative raw data transmission path, a stand-alone Java server application is responsible. It has to store the received values in the same database as the SOS server does.

Now an external web client can connect to the SOS and use OGC-compliant requests

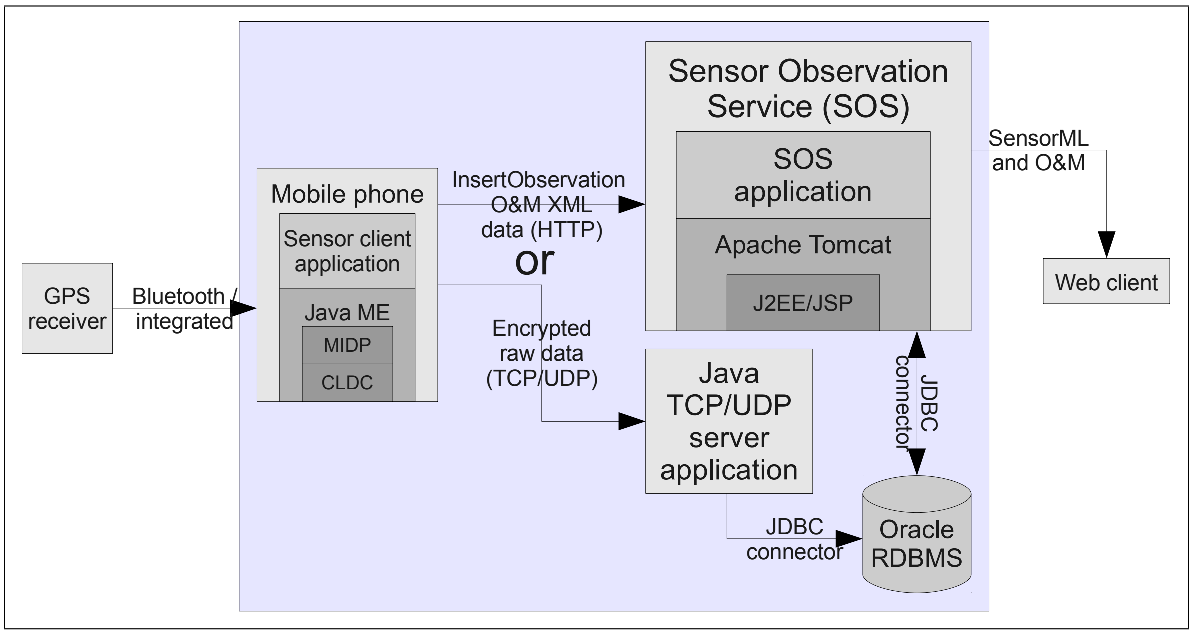


Figure 5.1.: Application structure

to fetch server and sensor metadata or measurements, or also insert new sensors or observations into the web service.

5.3. XML parser & generator

One critical question concerning the architecture of the implementation was the choice of a way to access XML in Java. As pre-generated classes are easier to handle than manually interpreting the XML markup tags, we first tried to feed the JAXB compiler with the SOS XSD schemas. Unfortunately, `xjc` failed with critizising that several elements were defined repeatedly, presumably due to faulty interdependencies of the schema definitions. With manually correcting the mentioned XSD files, a re-run of the compiler resulted in similar failures at other locations. Correcting these for three further dependency levels without success, the binding approach was dropped.

Additionally, a performance comparison resulted in DOM being 20% faster on average than JAXB [21]. As the incoming XML requests for the SOS are not very large-sized and therefore their tree representation does not consume much memory, the DOM approach was favored over SAX since this allows easier access to the content during the implementation. Due to SAX not being able to create XML which is needed for the SOS responses, DOM enables us to have the same API in both directions.

5.4. OGC-specific quirks

This section describes how the OGC standards were utilized to cope with mobile sensors, which constraints result from that, and which other improvements were made.

5.4.1. Enabling mobile sensors

In the basics section, the OGC nomenclature was introduced which already indicates that the standards were primarily defined for stationary (also called "in-situ") sensors. Here we describe how we accommodate mobile sensors into the OGC standards, without violating their compliance.

A major problem for coping with mobile sensors is the lack of a function that allows to update the position information. As mentioned before, the *DescribeSensor* SensorML response normally returns the sensor location which the server got once during the initial sensor publication in the *RegisterSensor* request. Instead of implementing an additional *UpdateSensor* operation, which is not part of the standard so far, our approach is to transmit the mobile sensor's current position inside the "feature of interest" of an *InsertObservation* request. In the *GetObservation* O&M response, the location parameters (latitude, longitude, altitude) are treated as phenomena and listed together with the other sensor measurements. By replacing the registered position with the most recent one during a *DescribeSensor* request, the response stays fully compliant in terms of XSD schema validation.

If a client of the 52° North SOS wants to track a mobile sensor's position over a specific period, it has to invoke lots of *DescribeSensor* requests. For all points in time during that period, the according position must be requested individually. The advantage of our approach is that due to declaring the position parameters as phenomena, we enable the client to request the whole track with only one *GetObservation* request. This also enables an easy usage of temporal and spatial filters.

Feature of interest

The "feature of interest" (FOI) brings the same problems with it as the proposed "domain feature" extension for the 52° North server does, because the sensor not necessarily knows if it is currently above a lake, a forest, etc. Using mobile phones as sensors, the phone owner could be the FOI with measuring his heart beat via a Bluetooth arm wrist. Also the underground of the owner may be an FOI: if he is moving on a bicycle, the phone's acceleration sensor may be employed to register pot-holes in the road. The detection of the current road would require further intelligence by employing a map-matching algorithm.

Whether the sensor or the server determining the current FOI, in both cases there arises the problem that if the mobile sensor passes a lot of different FOIs during its tour, this would heavily bloat up the *GetCapabilities* document, as they are all listed inhere for each sensor. Thus the decision was made to set the "feature of interest" equal to the *procedure*, that is the sensor ID. If there should arise a need for a different FOI later on, this is implementable without big effort by adding an additional column to the sensors or measurements database table.

Observation offering

A further ambiguous situation arises with the “observation offering”, that combines several related measurements into a group and which is the first required parameter in the *GetObservation* request. There are different criteria for the grouping and they are mostly dependent on the intended use of the measurements. The 52° North server for example defines offerings equal to the phenomena. This facilitates the request of attributes like temperature or wind speed over a large area that contains many sensors. But if a client wants to receive all measurements of a specific sensor/procedure, it has to send multiple requests, one for each phenomenon. A different choice is to set the offerings equal to the procedures. This allows an easy retrieval of all phenomena for a specific sensor, but on the other hand it requires multiple requests if the client needs the temperatures of a large area containing several sensors.

As the ESS sensor template document suggests to choose the latter variant [7, p. 16], our implementation will also use the *procedure* as the offering.

5.4.2. Further modifications

The following restriction was already indicated in the SensorML section. As we only want to cope with physical sensors, we assume that the sensor inputs are always equal to the outputs. This reduces the database structure in which the sensor metadata is stored.

Officially, the *timePosition* parameter in the GetObservation request only allows to contain an ISO 8601 timestamp for requesting observations of a certain point in time. Inspired by the 52° North implementation, a special *latest* keyword should allow to request the newest measurement of a specific sensor. This is useful for obtaining the last known sensor position, including the related values for all its phenomena.

Despite the standardization, there are two different methods for indicating the current sensor location. In the *InsertObservation* example of the OGC SOS standard, the position is stored in the result section, thus being a phenomenon. The 52° North examples as well as the IMEGO template put the position into the "feature of interest" section. As we want to stick with the IMEGO template, we also chose the second alternative.

5.5. Database

The database will run on a different machine than the servlet container does. This should result in a higher performance of the whole system, as each machine can use the full power for its dedicated function. Also due to the costly license of the Oracle RDBMS, one dedicated database machine where multiple application servers can connect to is preferable.

For facilitating the implementation of the spatial filters, which are needed to query sensors that are located within a specific area, the positions must be stored as spatial geometries. Further this enables us to visualize the sensor locations with the uDig GIS for validation purposes.

5.6. Raw sensor data transmission

As the alternative path of transferring raw sensor data from the mobile phone via UDP or TCP to a Java server application should use encryption, we have to select an encryption algorithm and find a method for employing user-dependent encryption keys.

The JSR 177 implements a subset of the "Java Cryptographic Extension" (JCE)[1]. It contains symmetric (DES, AES) and asymmetric (RSA) cryptographic algorithms as well as hash functions (MD5, SHA-1) and digital signatures. [13, p. 11]

[1]`http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`

To avoid the effort for deploying individual keys for each mobile phone, which would be needed for an asymmetric encryption, we decided to use a symmetric one. The idea is that the mobile phone user enters a password, which is then used as the encryption key for the data that he wants to send to the server. As we want to have a user-dependent encryption, a unique user ID is needed. For that, the IMEI number of the mobile phone shall be used. The server which receives the encrypted package uses the IMEI to fetch the user's password from the database, with which it can decrypt the data again.

From a security-related view, it is a bad idea to store the plain passwords in the database, as someone who gets access to the database server could easily fetch the user's passwords. Therefore only their hash values should be stored. This also has the advantage that due to the design of the hash functions, the effective encryption key always has the same length. As the cryptographic algorithms need a minimum key length of 56 bit for DES – which is not given if the user enters a password that consists of only 4 characters – the fixed 128 bit length of MD5 and the 160 bit of SHA-1 are on the safe side.

So the whole data package which the mobile phone sends to the server consists of the unencrypted IMEI and the number of measurements, followed by the encrypted location and the measurement values. A visualization of this concept can be seen in figure 5.2. The number of measurements is needed for letting the server know the real length of the encrypted data. This is due to the cryptographic algorithms working with blocks of a fixed size. If the data which should be encrypted does not match the block size, it is filled up to reach the block size. This procedure is called *padding*.

Since the transmitted data does not contain the names of the phenomena, which are needed to store the measurements in the same database as the SOS uses, the server must be configurable so that the operator can specify them.

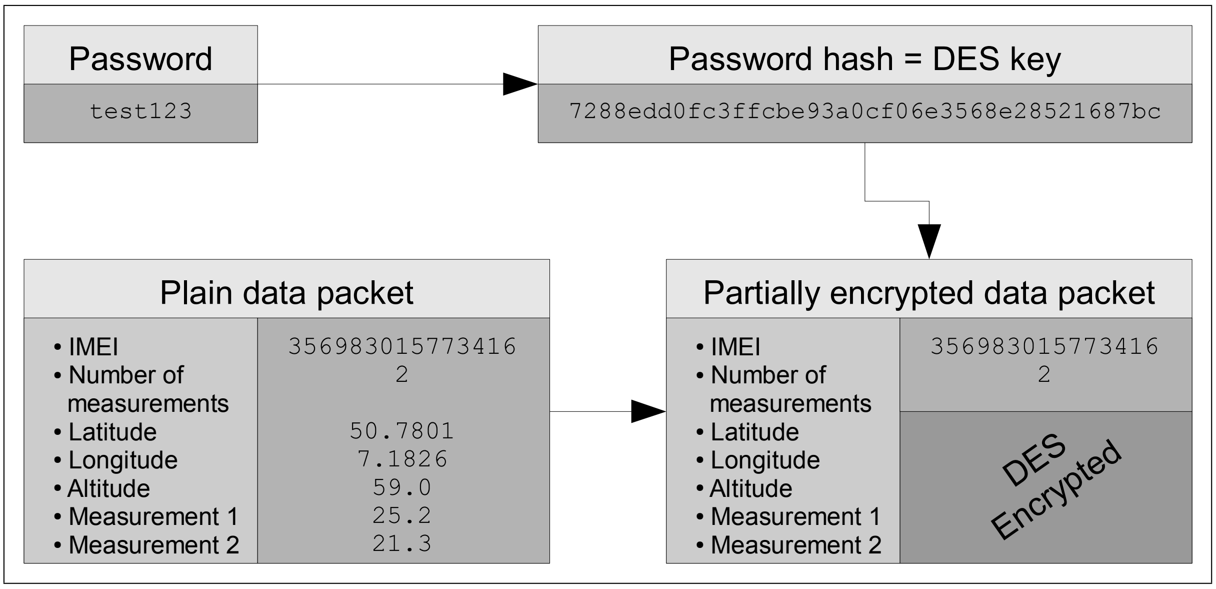


Figure 5.2.: Encryption in raw data transmission

6. Implementation

In this section, the implementation of the main SOS web servlet with its crucial functions is described, together with the underlying database. Subsequently the raw data transmission client/server applications follow as well as the performance testing client and a tool for importing existing data.

6.1. SOS servlet

The servlet that implements the Sensor Observation Service consists of two packages, from which one contains the implementation itself and the other contains classes that serve as data storage for the SOS operations and their requests and responses. An UML diagram of the package containing the core SOS classes is depicted in figure 6.1.

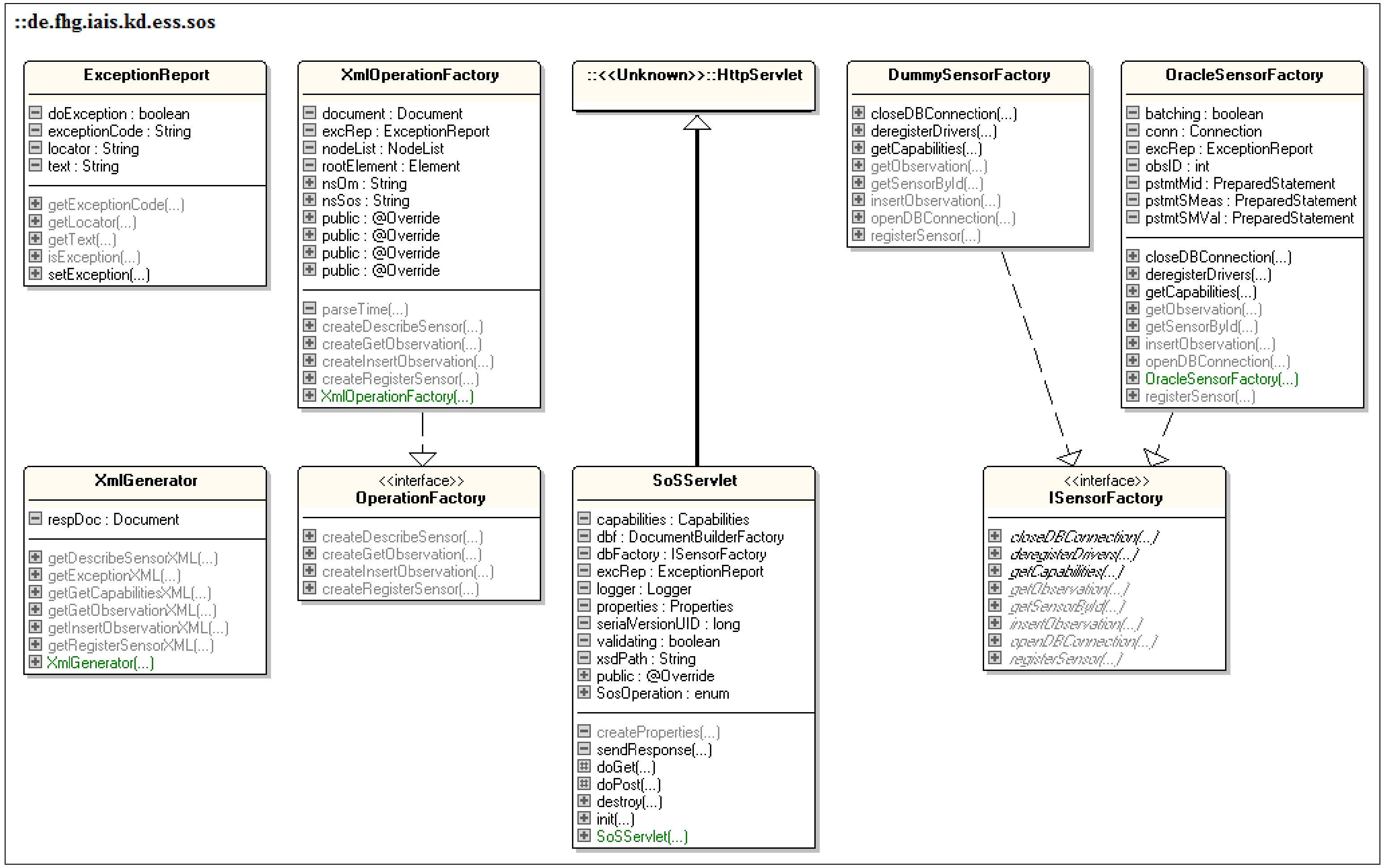


Figure 6.1.: SOS main package UML diagram

6.1.1. Main class

The `init` method is mandatory for every servlet. It is executed if the servlet is deployed to the server, and every time again when the server is restarted. In our implementation it initializes the Java Logger API which is used to output the servlet startup parameters and to print out warnings if a failure occurs. Using the *Properties* class, server settings like the database address and login parameters can be read from a file and easily be changed, without the need of re-compiling or re-deploying the servlet. Further the DOM document builder is initialized for parsing incoming requests and for creating the outgoing responses, and the database connection is opened. If the latter fails, the servlet falls back to a dummy database. This ensures that the server still boots up on a DB connection failure, being able to respond to incoming client requests. The dummy database will also be useful for doing white box tests later on.

As it was described in the basics SOS chapter, the requests may be done using the HTTP GET or POST operations. We find both of them again as functions of the Java servlet specification (`doGet` and `doPost`).

Since GET only must be supported by the *GetCapabilities* operation, and *GetCapabilities* does not need any further parameters, the content of `doGet` is pretty short: the request is checked for the *service* and *request* parameters that are appended to the URL, and which must be equal to `SOS` and `GetCapabilities`. If correct, the capabilities document is returned, otherwise the client gets an *ExceptionReport*.

The `doPost` method is more complex as there is an incoming XML document that needs to be parsed. A setting in the properties file declares, if the document shall undergo a schema validation in addition to the mandatory check for well-formedness. The schema validation has the advantage of making the servlet more robust against invalid incoming requests, at the cost of performance and response time. Normally, the validator retrieves the locations of the needed XSD schema documents from the header of the incoming

XML request. Then it fetches them, together with their further dependencies, and executes the validation. This leads to a further slowdown, as the external servers which host the documents – in our case the OGC servers – also need time for transmitting them over the net. To eliminate this bottleneck, the *EntityResolver* is used. It can manipulate the communication between the parser and external entities by overwriting the `resolveEntity` method [2, p. 282]. Figuratively, this can be seen as a search & replace function. In our implementation, it replaces the links to the external schema definition files with ones that are cached locally in a sub-directory of the server. Aside from the performance improvement, this allows it to run the SOS in a local network without an internet connection, and also keeps it running if the OGC servers should have a breakdown.

After the parsing, we check for the root element of the incoming document to determine the requested SOS operation. Basically, the handling of each operation follows the same procedure. First we pick out the required parameters of the request, use them for querying the database and finally wrap the database response into the corresponding OGC-conform XML document, which is then returned to the client. The detailed processing will be described in the appropriate sub-classes of the servlet. If there should occur an error in one of them, the servlet will send an *ExceptionReport* as the response.

6.1.2. XML parsing

The actual handling of the incoming XML request is done in the *XmlOperationFactory* class. Basically, it is the same for each of the SOS operations: first we check for the existence of the required attributes of the root element, from which the `service="SOS"` and `version="1.0.0"` attributes are mandatory.

Then we look for the presence of every needed markup tag – punishing missing ones by returning an exception – climbing down each branch of the DOM tree to the level of the actual text content. This is then either directly stored in an instance of the data storage classes or pre-processed before, if necessary.

One of the situations where we need further pre-processing of the input data is if an ISO 8601 timestamp occurs, like in the *GetObservation* request. For converting the timestamp into a Java `Date` object, an external class is used. It is called *TimeParser* and originates from the Fosstrak[1] project of the ETH Zurich.

6.1.3. XML generation

The generation of the response XML is done in the *XmlGenerator* class in a top-down way similar to the parsing. First, the root element that defines the document type is created, together with its required attributes and the employed namespaces. Then the further branches are created node by node. Their dynamic content is fetched from the appropriate instances of the data container classes which have previously been filled with database content.

6.1.4. Database access

In the *OracleSensorFactory*, the first function initializes the connection to the database by using the properties from the main program. Further there are methods that correspond to the SOS operations, which fetch data from or insert data into the database. They use the parsed input parameters of the request to query the database, and its response data is put into data storage objects again, which feed the XML generator.

If there are several web clients connecting simultaneously to the SOS, this induces the servlet container to create multiple threads of the `doPost` method from the main class. As we only use *one* database connection, we have to surround all further database queries with the `synchronize` keyword.

[1] `http://www.fosstrak.org/`

Statement vs PreparedStatement

Another important point is to use the *PreparedStatement* object instead of the *Statement* for executing SQL queries with dynamic parameters. To explain their difference, we first give examples of both methods in the listings 6.1 and 6.2.

```
public void sampleFunction(String uName) {
  ResultSet rs;
  Statement stmt = conn.createStatement();
  rs = stmt.executeQuery(
      "SELECT * FROM users WHERE uname = '" + uName + "'"
  );
  // do result processing here
  rs.close();
  stmt.close();
}
```

Listing 6.1: Sample java.sql.Statement

```
public void sampleFunction(String uName) {
  ResultSet rs;
  PreparedStatement pstmt = conn.prepareStatement(
      "SELECT * FROM users WHERE uname = ?"
  );
  pstmt.setString(1, uName);
  rs = pstmt.executeQuery();
  // do result processing here
  rs.close();
  pstmt.close();
}
```

Listing 6.2: Sample java.sql.PreparedStatement

In the *Statement* example, the *username* parameter is directly concatenated to the main SQL query. This has the disadvantage that a malicious client could specify a malformed user name to modify the actual SQL query. By passing the string `alice';DROP TABLE users; --` as the user name to the function, the final statement reads `SELECT * FROM users WHERE uname = 'alice';DROP TABLE users; --'`. This has the effect that after the real SQL command, another one is executed which deletes the whole user table. This attack is known as *SQL injection.* The setter method of the *PreparedStatement* in contrast checks the input parameters for malicious content to avoid such kind of manipulations. [12, p. 160]

Another advantage of this alternative is its superior performance, if the same statement is executed multiple times with varying parameters. For each *Statement*, the JDBC driver has to check its correctness and build an execution plan, before actually executing it. Using a *PreparedStatement*, this only has to be done once during the first execution. Afterwards the object knows that the SQL query is fixed, and only its parameters change. [22, p. 562]

Batching

Another performance improvement is available only for queries that write data into the database. It is called *batching* and a part of the JDBC 2.0 specification. Instead of sending insert queries one by one to the database, they are collected and sent together in a package. The Oracle JDBC driver implements its own batching aside from the standard JDBC batching. It only works for the *PreparedStatement* by configuring it with `((OraclePreparedStatement)pstmt).setExecuteBatch(int arg0)` where `arg0` specifies how many queries should be collected before they are sent to the server. [3, p. 442]

The disadvantage of this method is, that it first destroys the real-time of the SOS since it can take a lot of time until the specified amount of queries is reached, depending on the number of sensors and their activity. Second, a problem with the referential integrity arises. If a query is batched and not inserted into the database so far, a second sub-query that relies on data of the previous one may result in an error. To resolve this issue a *stored procedure* was written, which is a function that is executed on the database side and contains the main query with all required sub-queries in our case. For the JDBC client, it appears as *one* database function. But as we tried to batch this custom function, this resulted in a BatchException whose cause we were not able to track down. The final solution was to disable the referential integrity of the affected database tables and to manually care for the validity of the sub-queries on batching.

Instead of waiting for the number of batched queries to be reached, it is possible to explicitly induce them to be sent. This is done with the `sendBatch()` method of the *OraclePreparedStatement*. In our case it is executed in the function that closes the database connection on server shutdown.

6.2. Oracle database

The database layout was created with the *Oracle SQL Developer Data Modeler* under consideration of the database normalization criteria to avoid redundancies. An overview of the tables in form of an ER-diagram is shown in figure 6.2.

In the *sensors* table, the mandatory sensor ID and its optional metadata are saved. A foreign key from the *users* table can specify, which user is responsible for the sensor. In *sensorphenomena*, all phenomena are stored, with each of them referencing to their appropriate unit of measurement from the *units* table. The purpose of the *sensspmapping* table is to map an individual number of phenomena to each sensor.

For storing the observations, the *measurements* table was created. It must contain the sensor ID which it was received from, and the corresponding timestamp. The position

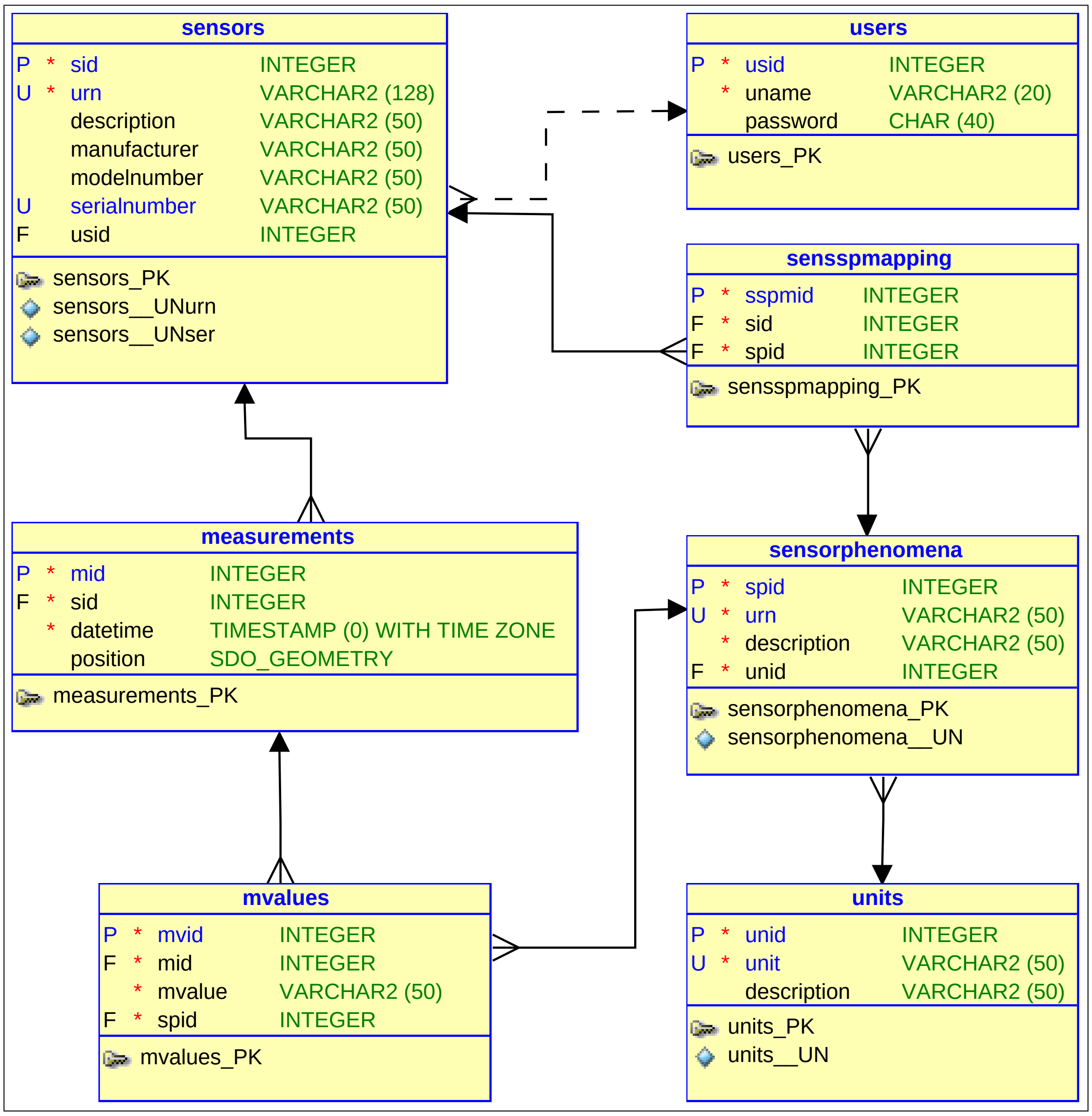


Figure 6.2.: Entity relationship diagram of the database layout

is contained in a spatial geometry for which a spatial index was created. As the number of measurement results may vary for each observation, they are stored in the additional table *mvalues*, which also have a mapping to their appropriate *sensorphenomena* entries.

The measurement values are stored as strings without distinguishing between data types like integer, float, etc. This facilitates the storage, and as the input and output of the SOS are always XML text documents, a differentiation is unnecessary.

During the implementation of the spatial filter queries it was observed that in spite of using a spatial index, the response time was pretty long. If directly executing the spatial queries on the database, we sometimes received Java exceptions if invalid parameters were used. This deduces that the spatial extension of the Oracle RDBMS is – in opposite to the database core – written in Java, which may be the cause of the response delay. So the spatial operator that is needed to select sensors within a certain area is only used for areas which are defined as polygons. In the GetCapabilities request, the accordant filter is defined by the OGC keyword *Contains*. If instead the *BBOX* filter for a rectangular bounding box is used, we perform a manual range comparison of the specified coordinates, instead of using the slower functions of the Oracle Spatial extension.

6.3. Java ME phone client

The mobile phone application for the raw sensor data transmission was written in Java ME. Figure 6.3 shows the UML diagram of its classes.

In the *Positioning* class, the location listener of the JSR 179 (Location API) is implemented, configured to retrieve a new location every second. It triggers the processing of the whole application, as with each received coordinate, a data packet has to be sent to the server. The new location with its corresponding timestamp are handed over to the root class *GPSSensorMIDlet* where they are supplied with two random numbers that demonstrate the measurements. Later on, they can be replaced with real measurements, like ones from the accelerometers in the phone which are accessible via the JSR 256 (Sensor API).

The aforementioned data is then delivered to the *Encryption* class. Inhere, the

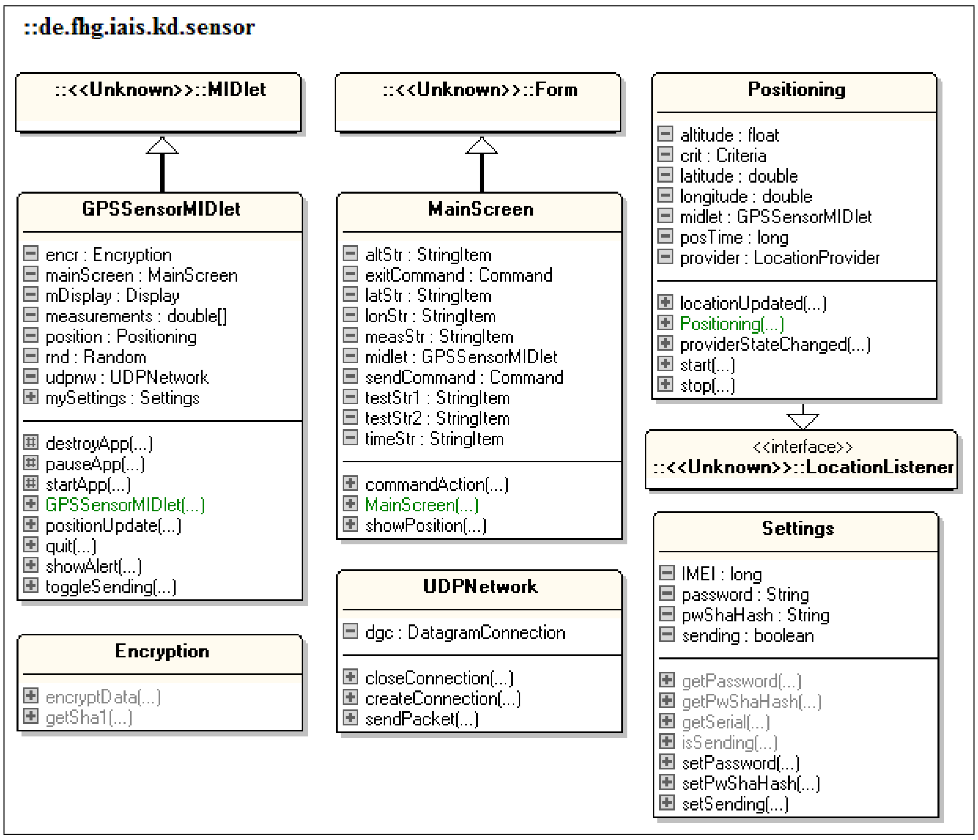


Figure 6.3.: Mobile client UML diagram

`getSha1` function calculates the SHA-1 hash value from the user's custom password by using the *MessageDigest* from the JSR 177 (SATSA-CRYPTO). The `EncryptData` method takes the whole data and the password hash and performs the DES encryption. From our password hash, only the first 8 characters are used because the encryption key must have this fixed length (56 bit DES key + parity). As the encryption function `doFinal` only takes byte arrays as the input, our data values have first to be converted and concatenated into *one* byte array. After the encryption, the result is returned to the main class.

Here, the network transmission function of the *UDPNetwork* class is invoked that needs the encrypted byte array, the number of measurements and the phone's serial number (IMEI) as input parameters. Normally the IMEI should be obtainable by invoking a `System.getProperty` call, but this did not work on our Nokia phones. Therefore it is currently stored as a fixed value and has later on to be entered manually by the user. The three values are concatenated into one byte array again, and then sent as an UDP datagram to the server.

Very similar classes exist for the other two transmission paths, namely TCP and HTTP, which are not shown in the UML diagram for brevity. The difference of the HTTP class is that it does not use encryption. It stores the static XML code of an *InsertObservation* request as a *String* and only completes the dynamic parts with the content of the appropriate variables.

The *MainScreen* class handles the visualization of the mobile phone display. It currently rather serves as a debug screen, as it displays the current position of the phone and the random measurement values. Further it is possible to toggle the network transmission, and of course, to exit the program.

6.4. Java TCP/UDP server

The structure of the server side application for the raw data transmission is depicted in figure 6.4.

In the *ServerRunner* class the main function is contained, which first processes the command line parameters. These are the names of the phenomena, which the the server requires to map the measurements from the mobile phone to SOS phenomena. This enables us to store them in the same database as the SOS uses. For saving database queries, the definition of the phenomena is implemented on a per-server basis, not per-client. This means that one server instance is only able to receive measurements from clients that measure the same phenomena.

Next, the database connection is initiated. The `storePosition` function of the

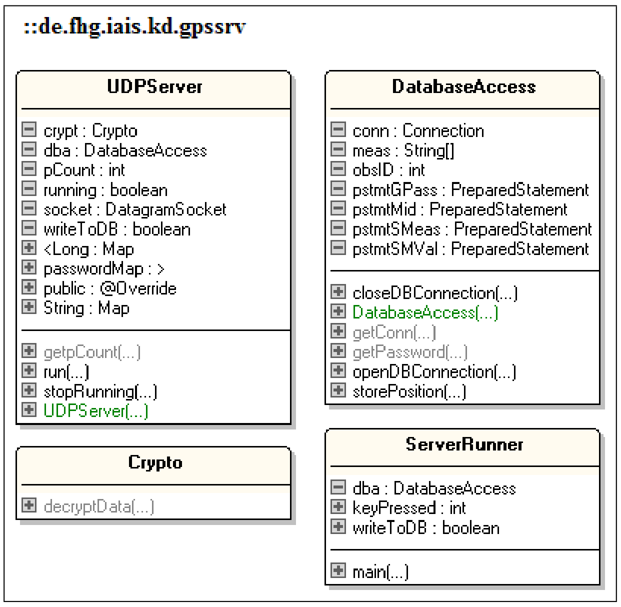


Figure 6.4.: TCP/UDP server UML diagram

DatabaseAccess class is very similar to the `InsertObservation` method from the *OracleSensorFactory* of the SOS implementation. It also uses *PreparedStatements* and batching, so that it is possible to do a fair performance comparison with the SOS later on. An additional function is needed here to fetch the user's password hash from the database, which is dependent on the IMEI sent with the measurements. To avoid the need of querying the database again and again for the same hash with each incoming data packet, the IMEI-to-hash mapping is cached in a Java `HashMap`.

Finally, the actual server thread – an instance of the *UDPServer* class – starts and listens for incoming network data. The received datagram packets are handled inversely to the mobile client, by first separating the plain data (IMEI and number of measurements) from the encrypted part. After decrypting the data by employing the *Crypto* class, the resulting byte array is split and casted into the proper data types, which then are written to the database. If the client used the wrong password for encryption, this results in a crypto exception and the packet will be discarded. The same happens, if the database does not know the specified IMEI.

An additional `writeToDB` Boolean setting in the server application allows to disable the call of the `storePosition` function. This is later on useful for doing white box tests, where we want to benchmark the server application independently of the database. For knowing how many UDP packets the server was actually able to handle, their quantity is counted and printed on the screen on server shutdown. This can then be compared with the number of packets which one or more clients have sent.

Initially the server was implemented to only work with UDP networking. Subsequently, it was extended to also support TCP by cloning the *UDPServer* class to *TCPServer* and modifying the appropriate network functions. An additional command line parameter was inserted for selecting the server type, which must be specified prior to the phenomena.

6.5. Java test client

As the Java ME emulators simulate whole mobile phones, they have a long start-up time and need a lot of resources (memory and processing power). This is especially a drawback if we want to simulate many mobile clients in parallel for determining the server's limits. Therefore we extracted the network and encryption code from the Java ME client to build a standalone Java testing application (figure 6.5), which is also capable of starting multiple threads in parallel to simulate a large number of clients.

The configuration of the test client is done via command line parameters for the main class *CommandLineTest*. First, the type of connection is specified: `http` for connecting to our SOS, `udp` or `tcp` for testing the raw data transmission, and `http52n` for executing a comparison with the 52° North implementation. It requires a different test as our SOS, since the 52° North server only allows the insertion of *one* measurement per second. Therefore we have to generate unique timestamps.

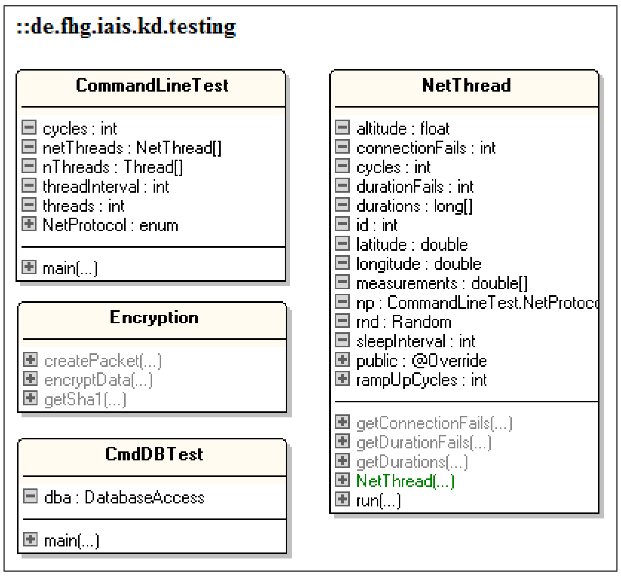


Figure 6.5.: Test client UML diagram

Further parameters are the number of threads that should run in parallel, from which each of them simulates a client. Next, the number of cycles per thread is specified, which indicates how many requests every thread sends to the server. The last parameter defines the maximum duration that a cycle may take. This can be compared to a soft real-time system, where we do not want to exceed a certain time range. If such an exceeding occurs, we are logging this as a failure.

In the *NetThread* class, the actual requests are performed. For the location we use fixed values, the timestamp is equal to the current system time and the measurements contain the number of concurrent threads. Latter is useful for determining, how many data packets of each test run actually reached the database, as package losses in the UDP transmission are not detectable otherwise. For the UDP/TCP testing, the encryption method is directly taken from the Java ME mobile client. The HTTP test uses static XML of the *InsertObservation* request which is directly embedded into the source code, and only replaces the measurements and timestamp with variables.

For each cycle, the whole time from the request until the response is measured. If it is below the specified maximum duration, a `Thread.sleep` is performed for the remaining time. As we often had duration exceedings during the first cycle of each thread, a ramp-up period was implemented that does 5 requests before the actual measurements begin. In the UDP case there is no response from the server, so we always execute a sleep of the maximum duration.

If the test client is finished, the result is printed to the console. It outputs the number of threads that were used, the average duration of each request, the number of total requests, the number of duration exceedings and the number of connection failures, where we got an exception from the connection object.

For doing multiple tests with varying parameters, a shell script for Linux and a batch file for Windows were written, which call the Java test client repeatedly.

To perform a JDBC performance test, which is useful to evaluate the different JDBC drivers and the benefits of batching, the small *CmdDBTest* class was written. It instantiates the *DatabaseAccess* class from the UDP server application, and does a configurable number of `storePosition` calls for inserting observations into the database. Afterwards, the number of insertions and the time that was needed for them is printed out to the console.

6.6. CSV import tool

During the current proof of concept phase of the ESS project, the data which the sensors produce is stored in CSV files. For later on being able to access this data using our SOS, an import tool was written. In a neighboring project, the locations of geo-tagged Flickr photos should be visualized, with the appropriate data already available in one big CSV file. So this file was used as testing input for the import tool. One entry of the file consists of location, timestamp, picture description and picture URL.

The CSV file name is specified as a command line parameter, and it is then read with the Super Csv library[2] line by line. As in the Java test client, the InsertObservation XML request is contained in the source code, and the dynamic values are read from variables which the CSV reader has filled with content. The description and the URL of the photo are handled as textual measurement values. Finally, the request is sent via HTTP to the SOS.

[2]http://supercsv.sourceforge.net/

7. Tests & results

In this section, the tests that evaluate the functionality and the performance of our implementation are described, together with their results.

7.1. Test environment

The server on which the SOS and alternately the Java server applications were running consists of an Intel Core2Duo E8400 (3.00GHz) CPU, with 4GB of RAM and a 500GB hard disk drive. On the software side, we employed Ubuntu Linux 9.10 in the 64 bit edition as the operating system, with Linux Kernel version 2.6.31. The servlet container was an Apache Tomcat 6.0.26 with APR enabled and TC native 1.1.20. Being based on the Tomcat's standard configuration, we modified its Java runtime settings, allowing it to use more memory by setting the environment variable `CATALINA_OPTS="-Xms1024m -Xmx3072m"`. Also, the maximum number of threads was increased in the *server.xml* file (`maxThreads="2000"`).

On a different machine, the Oracle 11g R2 database was running, with the exact version string being "Oracle Database 11g Release 11.2.0.1.0 - 64bit". Its operating system was Debian Linux 5.0.4 (64 bit, Kernel 2.6.26) on a machine that consists of an Intel Core2Duo E8400 with 8GB RAM and 250GB HDD. In the Java server applications, the Oracle JDBC Thin driver 11.2.0.1.0 was used to connect to the database.

A third machine was used to execute the test clients. It was an Intel Pentium D CPU with 3.00GHz, with 2GB RAM and 250GB HDD, running Debian Linux 4.1.1 (64 bit, Kernel 2.6.19).

7.2. Functionality test

For testing and verifying the functionality of our SOS, an already existent client from the ESS project was used, which initially was made for testing the IMEGO sensor templates with the 52° North implementation. It first publishes a new sensor using the *RegisterSensor* operation, and then checks its availability via *GetCapabilities*, followed by a *DescribeSensor* request. Then an *InsertObservation* is made for this sensor, which afterwards is verified by invoking the *GetObservation* operation.

The test client is either runnable as a JUnit test, or as a stand-alone command line application. Its requests are read from pre-made XML files, in which only changing parts like the sensor and observation IDs are replaced. Unfortunately, most of these files did not pass the XML schema validation, for which reason they were corrected to be fully OGC-compliant.

A run of the test client was successful for all aforementioned operations but the *GetObservation* request. This is due to the client expecting an *om:ObservationCollection* as the response, but our SOS sends the observation in *om:Observation* format, with the more compact *swe:DataArray* representation of the results.

7.3. Performance tests

The performance tests are dividable into two groups. In a black box test, the whole system is benchmarked, without a focus on its individual components. The white box tests in contrast analyze the system's elements seperately, being in our case the performance of the web server and that of the database.

7.3.1. Black box

For doing HTTP performance tests, there are tools like the *Apache JMeter*[1] which is written in Java and the *Apache HTTP server benchmarking tool*[2] (abbreviated *ab*) as a C application. According to [5, p. 129], *ab* is able to perform more requests per second than *JMeter*, for which reason the former should be preferred. The same source also advises to start the load testing tool on a different machine than the web server is running on.

As these tools are only suitable for performing HTTP load tests, we cannot use them to compare the performance of our SOS implementation to the raw data transmission, but for benchmarking the SOS separately. For making a comparison, the own Java test client was implemented.

Apache Benchmark

The *ab* tool was used in version 2.0.40-dev for performing *InsertObservation* requests on our SOS implementation, with each observation containing two phenomena. It was configured to do 100 concurrent requests and 10000 requests in total (the corresponding command line parameters are `-c 100` and `-n 10000`). The server was started in varying configurations by toggling the schema validation and the batching, which was set to 500 queries. Each test configuration was benchmarked 5 times, from which we picked the best result. An overview of the configurations and the test results is shown in table 7.1.

From the results we can see that the schema validation has a huge impact on the performance: with batching disabled, turning off the validation improved the throughput by a factor of 4.5, with batching enabled we even got an improvement by factor 88.

Looking at the batching option on activated schema validation, we get a small slow-

[1]`http://jakarta.apache.org/jmeter/`

[2]`http://httpd.apache.org/docs/2.2/programs/ab.html`

validation	batching	requests/s
on	off	32.86
off	off	150.47
on	on	23.19
off	on	2041.67

Table 7.1.: Apache HTTP server benchmarking tool black box test

down if batching is enabled. At this low throughput rate, this is presumably due to the higher administrative effort that the JDBC driver needs to cache the queries. But if the schema validation is disabled, the activation of batching causes a speedup of factor 13.5.

Regrettably we were not able to use the Apache benchmark tool on the 52° North implementation, as the server only accepts one observation per second for each sensor, and the *ab* tool cannot modify the content of the input XML file, allowing us to specify dynamic sensor IDs and timestamps.

Java test client

The Java test client was started multiple times with varying parameters, by using a shell script as it was introduced in the implementation chapter. In the first run, the number of threads was increased in steps of 50 until reaching 2000 concurrent threads. Every thread did 10 *InsertObservation* requests one after another, from which each of them was allowed to have a maximum duration of one second, as this is the common interval in which GPS devices update the position. This test was done for all the 4 SOS configuration combinations (enabling/disabling batching and schema validation) and for the TCP and UDP raw data transmissions, which always have batching enabled on the server side. The average duration of a request for each of the transmission options is depicted in figure 7.1.

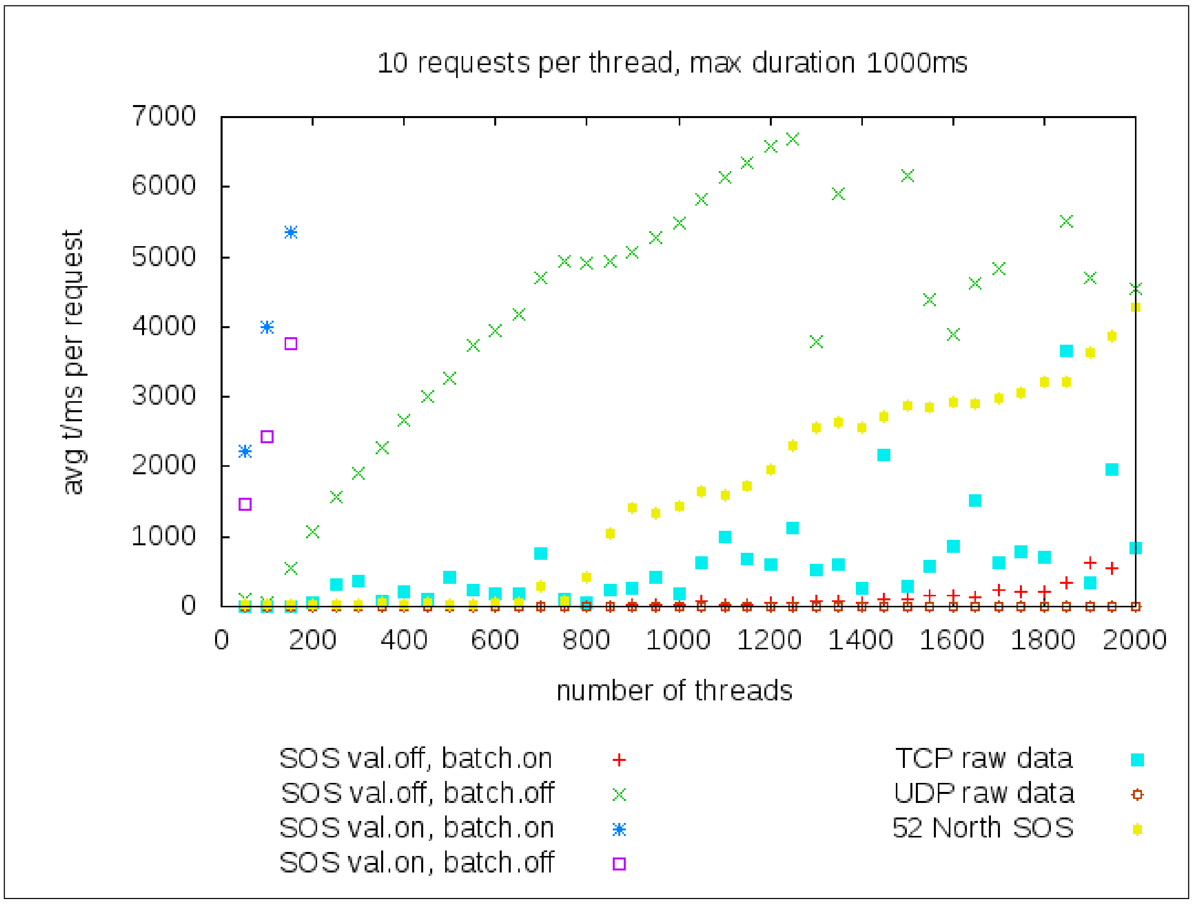


Figure 7.1.: Java test client throughput results

In the case that a single request needed more than one second in one of the connection-oriented protocols (HTTP & TCP), violating our "real-time" constraints, this was counted as a *duration failure* (figure 7.2). This means that the data has correctly been received and stored in the database, but not in time. As the test client does not receive a response from a sent UDP datagram, this connection type cannot be incorporated here.

If the server could not handle the request at all, resulting in an exception or in the case of UDP, if the datagram content was afterwards not findable in the database, this was counted as a *connection failure*. In opposite to the duration failures, here the data is lost. The count of these failures is shown in figure 7.3.

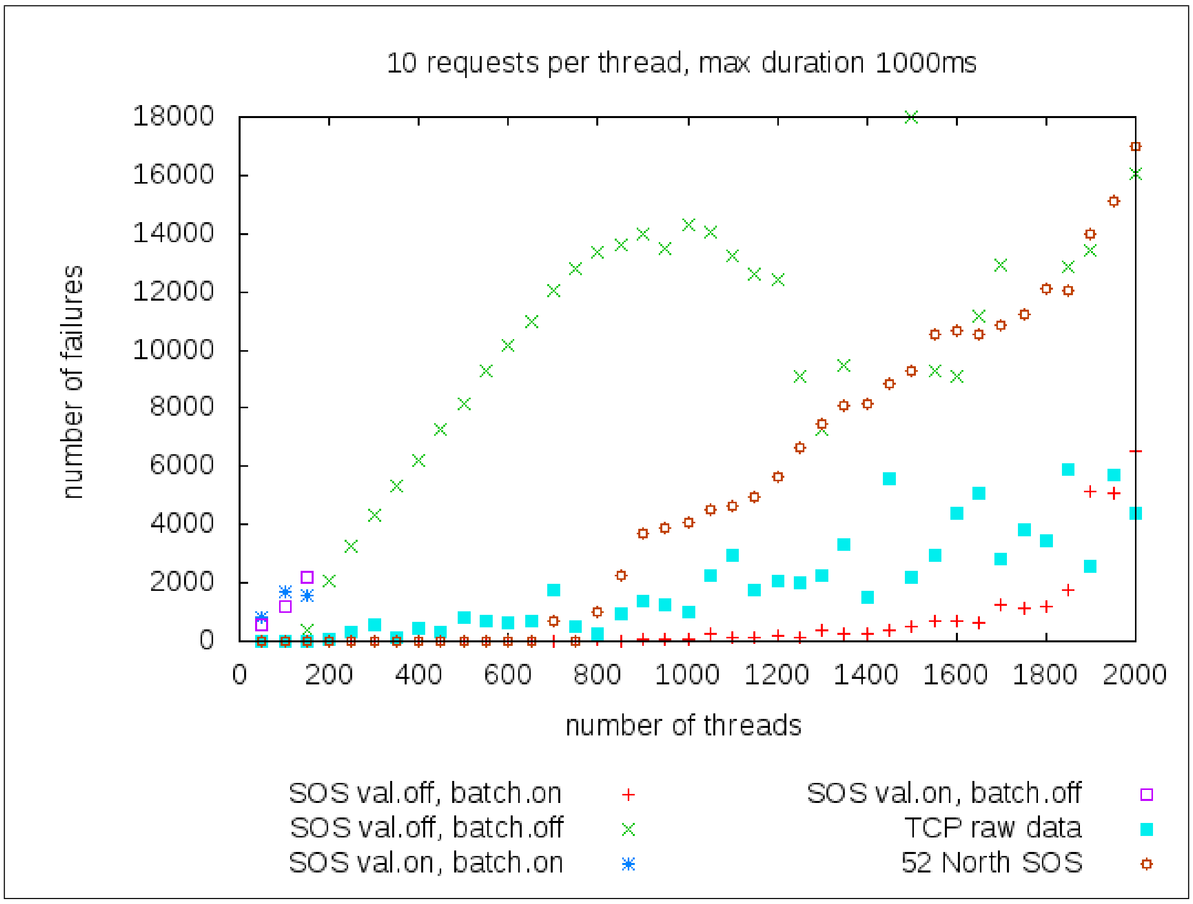


Figure 7.2.: Java test client duration failures

Looking at the different SOS configurations, the test confirms the results of the previous Apache benchmarking tool. With schema validation activated and the batching option not significantly causing a difference, the server completely fails at doing more than 150 threads in parallel. In the Tomcat log file there were out of memory errors, as the server cache fills up with more incoming requests than it is able to process.

With schema validation and batching both turned off, the first timing constraint failures also occur at 150 threads, but there were no memory errors and the server still kept up at 700 threads. Then the first connection failures begin to occur.

The most performant SOS configuration is the one with schema validation turned off

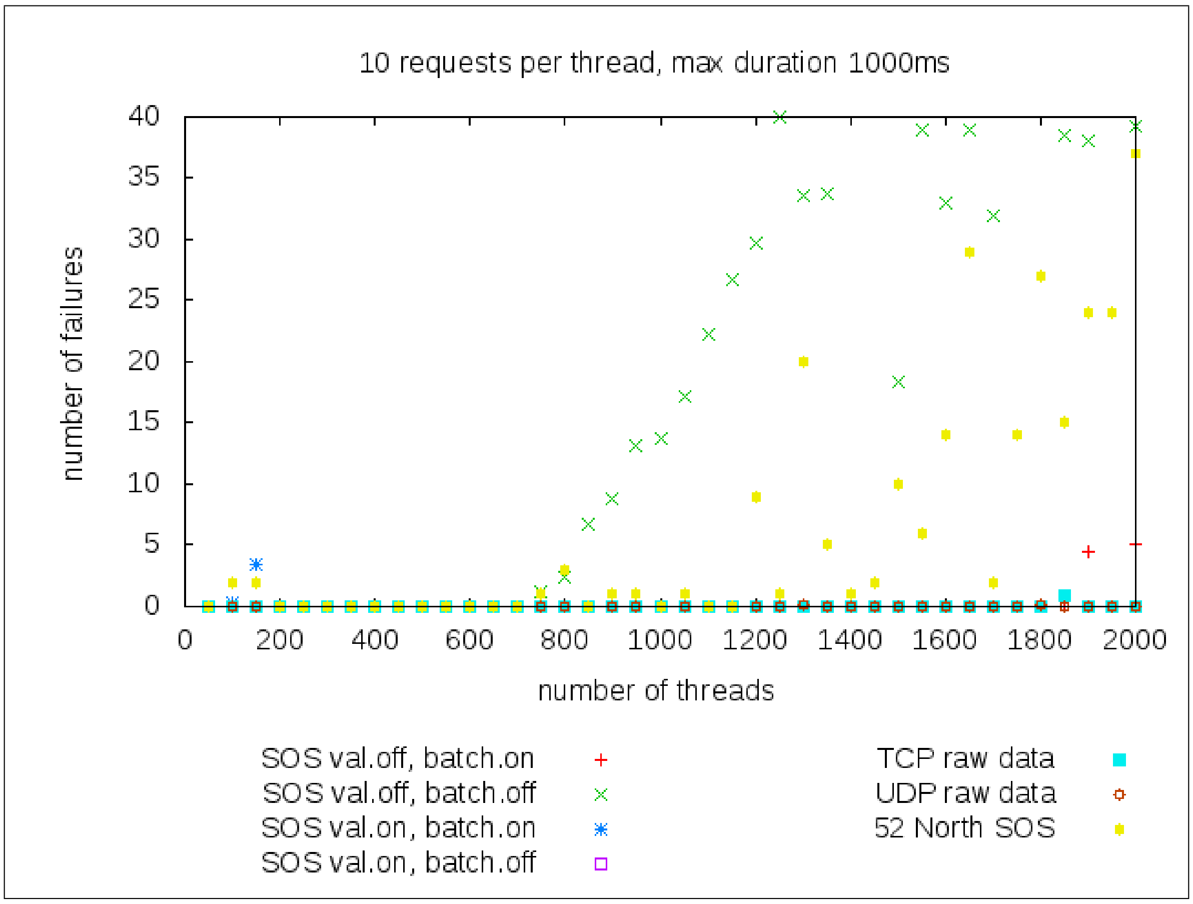


Figure 7.3.: Java test client connection failures

and batching enabled. First duration failures occur at about 1000 concurrent threads and slowly start to raise until 1900 threads, where we have an increasing slope and also the first connection failures. This number is also not far distanced from the maximum of 2000 requests per second from the Apache benchmark.

A comparison with the 52° North SOS shows that its first duration failures occur at about 800 threads in parallel, and a high number of connection failures starts at 1200 threads. This lies half the way in between the two options of batching enabled and disabled of our implementation, with both having schema validation deactivated.

Switching over to the raw data transmission, the first TCP duration failures already occur very early, but raise with a low slope and there are only marginal connection failures up to 2000 threads.

The UDP transmission has the first connection failures at a number of 1900 concurrent threads, having the disadvantage in opposite to the TCP protocol that lost packages are not resent.

To answer the initial question on the overhead that the SOS *InsertObservation* XML request via HTTP causes over a raw data transmission, we can conclude that depending on the server configuration, it does not make a big difference. Since with UDP a packet loss is not detectable for the server, only TCP would be a reliable option. In this case, the response time was even higher than with the fastest configuration of our SOS implementation. This is presumably due to the high optimization level of the Tomcat implementation and its multi-threading architecture, that our raw data Java server is missing.

7.3.2. White box

Already during the implementation phase, some of the white box tests were done to detect performance bottlenecks. The Eclipse IDE offers an extension called "Eclipse Test & Performance Tools Platform Project" (TPTP) which includes a *profiler*. It measures execution time and memory consumption of an application and afterwards visualizes the resulting values top-down from the project's Java classes to its functions. The profiler led us to the insight that the schema validation and the JDBC functions needed the most processing time, for which reason the option to disable the validation was implemented and further database tests and optimizations (batching) were done.

Web server

For separately measuring the performance of the SOS implementation, the database functionality was deactivated and the corresponding classes were replaced by a dummy database class, as it was mentioned in the implementation chapter. As we do not need a comparison with the raw data transmission here, only the Apache benchmark was performed. With schema validation enabled, a throughput of about 30 requests per second was reached. Switching the validation off, the performance resulted in 2280 requests per second. These values are very close-by to the previously executed black box tests, showing that not database is the bottleneck in the current setup, but the web server or rather our SOS implementation.

Database

The database and JDBC driver performance was tested by employing the *CmdDBTest* test application. It was configured to use batching and to insert 100000 measurements into the database, each consisting of one entry in the *measurements* table and two into the *mvalues* table. This simulates the same amount of data as we used for the previously made *InsertObservation* requests, each containing two phenomena.

Two different Oracle JDBC drivers were tested, namely the *Thin* driver and the *OCI* (Oracle Call Interface) driver. The most significant difference between the drivers is that the former is completely written in Java while the latter needs an additional library that is dependent on the operating system which the Java application runs on. This should in exchange yield to a higher performance.

While the *Thin* driver needed 32.02s to do the 100000 inserts, the *OCI* driver did the same task in 32.69s, being slightly slower. But both results correspond to about 3100 requests per second – a value that outperforms the web server performance, and which is also higher than all the previous black box test results.

If an even higher database insertion performance is needed, this can be achieved by disabling the spatial index of the *position* column from the *measurements* table. This has the drawback of a worse performance on spatial queries that retrieve data from the database, so it could be configured for only being used temporarily if there is a high insertion load. Without the index, we achieved 12795 requests per second with the *Thin* driver and 12285 when using the *OCI*.

This shows that in the second case, the *OCI* driver was even slower than the *Thin* driver, resulting in our decision to stay with the *Thin* driver, as it is also easier to deploy since it does not need the OS-dependent libraries. Further we learned that the Oracle Spatial extension – more precisely the spatial index – slowed down the insertion performance by a factor of 4. Depending on the application scenario of the SOS, if no spatial filters are needed in the *GetObservation* request or if a bounding box filter, which is implemented as a size comparison of latitude and longitude, is sufficient, the spatial index can be dropped.

7.4. Outdoor test

For being able to perform an outdoor test with the mobile phone sending data to the SOS, we setup a Tomcat server which is accessible through the Internet. The phone then sends its position and measurement values via UMTS or GPRS to the SOS servlet. Having recorded the bus drive to the ESS field test location, the resulting track is shown in figure 7.4.

The transmission nearly worked without problems, only if there was a dead spot in the mobile phone network, packets got lost. In the track they are not visible, as the visualization software interconnects consecutively received locations. To avoid packets being lost, the implementation could be extended to cache and resend them, which is of course not possible on UDP raw data transmission.

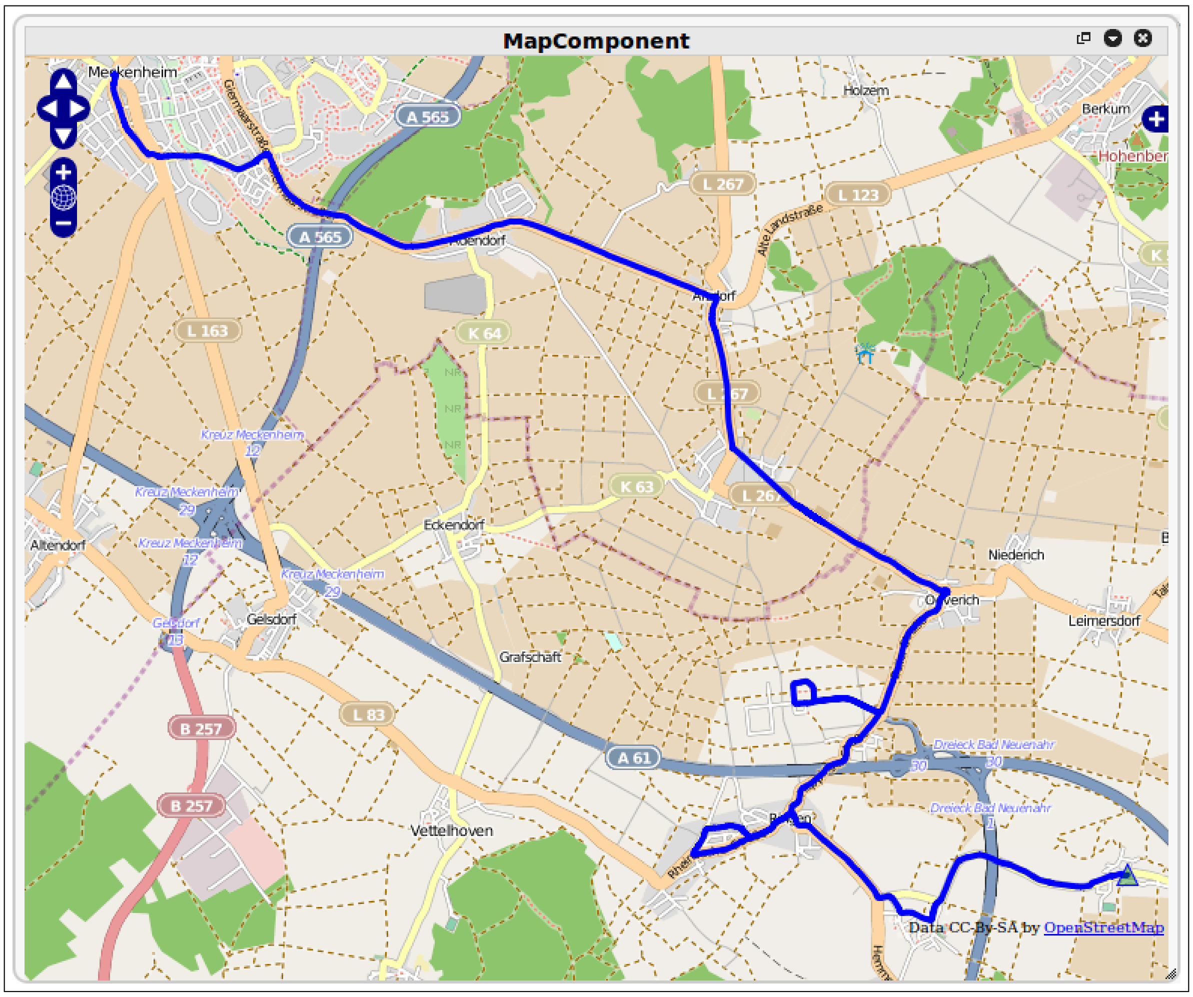


Figure 7.4.: Track created during outdoor test

8. Summary & Outlook

The implementation of our Sensor Observation Service has shown the possibility of enabling mobile sensors in an OGC-compliant manner. The mobility is reached by handling the positions as measurements and by manipulating the returned data content (being the current sensor position in the *DescribeSensor* SensorML response) without violating or extending the OGC standards. Especially mobile phones were successful enabled to act as mobile sensors, but also the integration of existing data in CSV format was possible.

By giving the server administrator the choice of toggling the XML schema validation and database batching options, we enable him to customize the server configuration for more performance or more reliability. In the most performant setting, our implementation even beats the 52° North SOS which is often treated as the reference SOS implementation.

Having setup this fundamental SOS implementation, there is still further space for improvements. As the implementation was initially made with relying on the schema validation, it can get weak if the validation option is deactivated and malformed XML requests are sent by a client.

Further the batching option increases the performance of inserting new observations to the server, but destroys its real-time capability. To cut down this drawback, a monitoring thread could be implemented which induces the JDBC driver to commit the current batch queue at a certain time interval.

As we have also seen to occur in other SOS implementations, the servers stopped responding if a client requested a very large number of observations. A workaround could be a limitation of the measurements in the *GetObservation* response (e.g. 1000 entries) to avoid the server freeze.

For the upcoming SOS 2.0 specification, the XML requests and responses will be put into SOAP envelopes. There will also be new functions to better cope with mobile sensors, which may be useful to be contained in our SOS.

Bibliography

[1] Algosystems S.A. Press release at the Start of ESS. `http://www.ess-project.eu/downloads/category/1-press.html`, August 2009.

[2] Dirk Ammelburger. *XML mit Java.* Markt+Technik, 2002.

[3] Donald Bales. *Java Programming with Oracle JDBC.* O'Reilly Media, 2001.

[4] Mike Botts and Alex Robin. Bringing the sensor web together. *Géosciences*, 6:46–53, 2007.

[5] Jason Brittain and Ian F. Darwin. *Tomcat: The Definitive Guide.* O'Reilly Media, 2007.

[6] Carsten Czarski. Auf den Ort kommt es an: Geodaten-Untersttzung im RDBMS Oracle. *Datenbank-Spektrum*, 21:22–29, 2007.

[7] Johannes Echterhoff and Ingo Simonis. ESS WP3 - SWE Template for IMEGO Sensors, July 2009.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[9] Kurt A. Gabrick and David B. Weiss. *J2EE and XML Development.* Manning Publications Co., 2002.

[10] Rick Greenwald, Robert Stackowiak, and Jonathan Stern. *Oracle Essentials - Oracle Database 11g*. O'Reilly Media, 4th edition, 2007.

[11] Brett D. McLaughlin and Justin Edelson. *Java & XML*. O'Reilly Media, 3rd edition, 2006.

[12] R.M. Menon. *Expert Oracle JDBC Programming*. Apress, 2005.

[13] Nokia Corporation. *MIDP: SATSA Crypto API Developer's Guide Version 2.0*, 2006.

[14] Open Geospatial Consortium Inc. OpenGIS Simple Features Implementation Specification for SQL. OGC 99-049, May 1999.

[15] Open Geospatial Consortium Inc. Observations and Measurements - Part 1 - Observation schema. OGC 07-022r1, December 2007.

[16] Open Geospatial Consortium Inc. OGC Web Services Common Specification. OGC 06-121r3, February 2007.

[17] Open Geospatial Consortium Inc. OpenGIS Geography Markup Language (GML) Encoding Standard. OGC 07-036, August 2007.

[18] Open Geospatial Consortium Inc. OpenGIS Sensor Model Language (SensorML) Implementation Specification. OGC 07-000, July 2007.

[19] Open Geospatial Consortium Inc. Sensor Observation Service. OGC 06-009r6, October 2007.

[20] Open Geospatial Consortium Inc. Implementations by Specification. `http://www.opengeospatial.org/resource/products/byspec/?specid=289`, 2009.

[21] Dr. Santiago Pericas-Geertsen. DOM vs. JAXB Performance. `http://weblogs.java.net/blog/2005/12/01/dom-vs-jaxb-performance`, December 2005.

[22] Jason Price. *Oracle Database 11g SQL*. McGraw-Hill Osborne Media, 2007.

[23] Christoph Stasch, Arne Bröring, and Alexander C. Walkowski. Providing Mobile Sensor Data in a Standardized Way – The SOSmobile Web Service Interface. 2008.

[24] Ajay Vohra and Deepak Vohra. *Pro XML Development with Java Technology*. Apress, 2006.

A. SOS XML example documents

GetCapabilities

Request

```
<?xml version="1.0" encoding="UTF-8"?>
<GetCapabilities xmlns="http://www.opengis.net/sos/1.0"
    xmlns:swe="http://www.opengis.net/swe/1.0.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opengis.net/sos/1.0
     http://schemas.opengis.net/sos/1.0.0/sosAll.xsd"
    service="SOS">
</GetCapabilities>
```

Response

```
<?xml version="1.0" encoding="UTF-8"?>
<Capabilities xmlns="http://www.opengis.net/sos/1.0"
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:ogc="http://www.opengis.net/ogc"
    xmlns:om="http://www.opengis.net/om/1.0"
    xmlns:ows="http://www.opengis.net/ows/1.1"
    xmlns:sml="http://www.opengis.net/sensorML/1.0.1"
    xmlns:sos="http://www.opengis.net/sos/1.0"
    xmlns:swe="http://www.opengis.net/swe/1.0.1"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opengis.net/sos/1.0
     http://schemas.opengis.net/sos/1.0.0/sosGetCapabilities.xsd"
    version="1.0.0">
```

```
<ows:ServiceIdentification>
 <ows:Title>ESS SOS</ows:Title>
 <ows:Abstract>ESS Sensor Observation Service test</ows:Abstract>
 <ows:Keywords>
  <ows:Keyword>ESS</ows:Keyword>
 </ows:Keywords>
 <ows:ServiceType codeSpace="http://opengeospatial.net">
  OGC:SOS
 </ows:ServiceType>
 <ows:ServiceTypeVersion>1.0.0</ows:ServiceTypeVersion>
 <ows:Fees>NONE</ows:Fees>
 <ows:AccessConstraints>NONE</ows:AccessConstraints>
</ows:ServiceIdentification>
<ows:ServiceProvider>
 <ows:ProviderName>IAIS</ows:ProviderName>
 <ows:ProviderSite xlink:href="http://www.iais.fraunhofer.de/"/>
 <ows:ServiceContact>
  <ows:IndividualName>Roland Mueller</ows:IndividualName>
  <ows:PositionName>MSc student</ows:PositionName>
 </ows:ServiceContact>
</ows:ServiceProvider>
<ows:OperationsMetadata>
 <ows:Operation name="GetCapabilities">
  <ows:DCP>
   <ows:HTTP>
    <ows:Get xlink:href="http://cherry.iais.fraunhofer.de:8080/ESSSoS/sos?"/>
    <ows:Post xlink:href="http://cherry.iais.fraunhofer.de:8080/ESSSoS/sos"/>
   </ows:HTTP>
  </ows:DCP>
  <ows:Parameter name="service">
   <ows:AllowedValues>
    <ows:Value>SOS</ows:Value>
   </ows:AllowedValues>
  </ows:Parameter>
  <ows:Parameter name="updateSequence">
   <ows:AnyValue/>
  </ows:Parameter>
  <ows:Parameter name="AcceptVersions">
   <ows:AllowedValues>
    <ows:Value>1.0.0</ows:Value>
```

```
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="Sections">
      <ows:AllowedValues>
        <ows:Value>ServiceIdentification</ows:Value>
        <ows:Value>ServiceProvider</ows:Value>
        <ows:Value>OperationsMetadata</ows:Value>
        <ows:Value>Contents</ows:Value>
        <ows:Value>All</ows:Value>
        <ows:Value>Filter_Capabilities</ows:Value>
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="AcceptFormats">
      <ows:AllowedValues>
        <ows:Value>text/xml</ows:Value>
      </ows:AllowedValues>
    </ows:Parameter>
  </ows:Operation>
  <ows:Operation name="GetObservation">
    <ows:DCP>
      <ows:HTTP>
        <ows:Post xlink:href="http://cherry.iais.fraunhofer.de:8080/ESSSoS/sos"/>
      </ows:HTTP>
    </ows:DCP>
    <ows:Parameter name="version">
      <ows:AllowedValues>
        <ows:Value>1.0.0</ows:Value>
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="service">
      <ows:AllowedValues>
        <ows:Value>SOS</ows:Value>
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="srsName">
      <ows:AnyValue/>
    </ows:Parameter>
    <ows:Parameter name="offering">
      <ows:AllowedValues>
        <ows:Value>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</ows:Value>
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="eventTime">
      <ows:AllowedValues>
        <ows:Range>
          <ows:MinimumValue>2010-05-05T11:26:35+02:00</ows:MinimumValue>
        </ows:Range>
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="procedure">
      <ows:AllowedValues>
        <ows:Value>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</ows:Value>
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="observedProperty">
      <ows:AllowedValues>
        <ows:Value>urn:ogc:def:property:OGC::AirTemperature</ows:Value>
        <ows:Value>urn:ogc:def:property:OGC::AtmosphericPressure</ows:Value>
        <ows:Value>urn:ogc:def:property:OGC::RelativeHumidity</ows:Value>
        <ows:Value>urn:ogc:def:property:OGC::WindDirection</ows:Value>
        <ows:Value>urn:ogc:def:property:OGC::WindSpeed</ows:Value>
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="featureOfInterest">
      <ows:AnyValue/>
    </ows:Parameter>
    <ows:Parameter name="result">
      <ows:AnyValue/>
    </ows:Parameter>
    <ows:Parameter name="responseFormat">
      <ows:AllowedValues>
        <ows:Value>text/xml;subtype=&amp;quot;OM/1.0.0&amp;quot;</ows:Value>
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="resultModel">
      <ows:AllowedValues>
        <ows:Value>om:Observation</ows:Value>
      </ows:AllowedValues>
    </ows:Parameter>
    <ows:Parameter name="responseMode">
```

```
    <ows:AllowedValues>
      <ows:Value>inline</ows:Value>
    </ows:AllowedValues>
  </ows:Parameter>
</ows:Operation>
<ows:Operation name="DescribeSensor">
  <ows:DCP>
    <ows:HTTP>
      <ows:Post xlink:href="http://cherry.iais.fraunhofer.de:8080/ESSSoS/sos"/>
    </ows:HTTP>
  </ows:DCP>
  <ows:Parameter name="version">
    <ows:AllowedValues>
      <ows:Value>1.0.0</ows:Value>
    </ows:AllowedValues>
  </ows:Parameter>
  <ows:Parameter name="service">
    <ows:AllowedValues>
      <ows:Value>SOS</ows:Value>
    </ows:AllowedValues>
  </ows:Parameter>
  <ows:Parameter name="outputFormat">
    <ows:AllowedValues>
      <ows:Value>text/xml;subtype=&amp;quot;sensorML/1.0.1&amp;quot;</ows:Value>
    </ows:AllowedValues>
  </ows:Parameter>
  <ows:Parameter name="procedure">
    <ows:AllowedValues>
      <ows:Value>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</ows:Value>
    </ows:AllowedValues>
  </ows:Parameter>
</ows:Operation>
<ows:Operation name="GetFeatureOfInterest">
  <ows:DCP>
    <ows:HTTP>
      <ows:Post xlink:href="http://cherry.iais.fraunhofer.de:8080/ESSSoS/sos"/>
    </ows:HTTP>
  </ows:DCP>
  <ows:Parameter name="service">
    <ows:AllowedValues>
```

```
     <ows:Value>SOS</ows:Value>
    </ows:AllowedValues>
   </ows:Parameter>
   <ows:Parameter name="version">
    <ows:AllowedValues>
     <ows:Value>1.0.0</ows:Value>
    </ows:AllowedValues>
   </ows:Parameter>
   <ows:Parameter name="featureOfInterestId">
    <ows:AllowedValues>
     <ows:Value>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</ows:Value>
    </ows:AllowedValues>
   </ows:Parameter>
   <ows:Parameter name="location">
    <ows:AnyValue/>
   </ows:Parameter>
  </ows:Operation>
 </ows:OperationsMetadata>
 <sos:Filter_Capabilities>
  <ogc:Spatial_Capabilities>
   <ogc:GeometryOperands>
    <ogc:GeometryOperand>gml:Envelope</ogc:GeometryOperand>
    <ogc:GeometryOperand>gml:Polygon</ogc:GeometryOperand>
    <ogc:GeometryOperand>gml:Point</ogc:GeometryOperand>
    <ogc:GeometryOperand>gml:LineString</ogc:GeometryOperand>
   </ogc:GeometryOperands>
   <ogc:SpatialOperators>
    <ogc:SpatialOperator name="BBOX"/>
    <ogc:SpatialOperator name="Contains"/>
    <ogc:SpatialOperator name="Intersects"/>
    <ogc:SpatialOperator name="Overlaps"/>
   </ogc:SpatialOperators>
  </ogc:Spatial_Capabilities>
  <ogc:Temporal_Capabilities>
   <ogc:TemporalOperands>
    <ogc:TemporalOperand>gml:TimeInstant</ogc:TemporalOperand>
    <ogc:TemporalOperand>gml:TimePeriod</ogc:TemporalOperand>
   </ogc:TemporalOperands>
   <ogc:TemporalOperators>
    <ogc:TemporalOperator name="TM_During"/>
```

```
      <ogc:TemporalOperator name="TM_Equals"/>
      <ogc:TemporalOperator name="TM_After"/>
      <ogc:TemporalOperator name="TM_Before"/>
     </ogc:TemporalOperators>
    </ogc:Temporal_Capabilities>
    <ogc:Scalar_Capabilities>
     <ogc:ComparisonOperators>
      <ogc:ComparisonOperator>Between</ogc:ComparisonOperator>
      <ogc:ComparisonOperator>EqualTo</ogc:ComparisonOperator>
      <ogc:ComparisonOperator>NotEqualTo</ogc:ComparisonOperator>
      <ogc:ComparisonOperator>LessThan</ogc:ComparisonOperator>
      <ogc:ComparisonOperator>LessThanEqualTo</ogc:ComparisonOperator>
      <ogc:ComparisonOperator>GreaterThan</ogc:ComparisonOperator>
      <ogc:ComparisonOperator>GreaterThanEqualTo</ogc:ComparisonOperator>
      <ogc:ComparisonOperator>Like</ogc:ComparisonOperator>
     </ogc:ComparisonOperators>
    </ogc:Scalar_Capabilities>
    <ogc:Id_Capabilities>
     <ogc:FID/>
     <ogc:EID/>
    </ogc:Id_Capabilities>
   </sos:Filter_Capabilities>
   <Contents>
    <ObservationOfferingList>
     <ObservationOffering gml:id="936DA01F-9ABD-4d9d-80C7-02AF85C822A8">
      <gml:description>Combines the observations produced by
       IMEGO Medium Sized Sensor Module
      </gml:description>
      <gml:boundedBy>
       <gml:Envelope srsName="urn:ogc:def:crs:EPSG:6.14:4979">
        <gml:lowerCorner>50.73822 7.11034 50.0</gml:lowerCorner>
        <gml:upperCorner>50.73822 7.11034 50.0</gml:upperCorner>
       </gml:Envelope>
      </gml:boundedBy>
      <intendedApplication>weather monitoring</intendedApplication>
      <time>
       <gml:TimePeriod>
        <gml:beginPosition>2010-05-05T11:26:35+02:00</gml:beginPosition>
        <gml:endPosition>2010-05-05T11:26:44+02:00</gml:endPosition>
       </gml:TimePeriod>
      </time>
      <procedure xlink:href="936DA01F-9ABD-4d9d-80C7-02AF85C822A8"/>
      <observedProperty>
       <swe:CompositePhenomenon dimension="5" gml:id="adHocPhenomenon">
        <gml:description>Ad hoc phenomenon - best practice when measuring
         multiple properties at the same time for a given feature of interest.
        </gml:description>
        <gml:name codeSpace="urn:ogc:tc:arch:doc-bp(xx-xxx)">
         319C201F-9000-47dd-3258-835169543B9
        </gml:name>
        <gml:name>ad hoc compound phenomenon</gml:name>
        <swe:component xlink:href="urn:ogc:def:property:OGC::AirTemperature"/>
        <swe:component xlink:href="urn:ogc:def:property:OGC::AtmosphericPressure"/>
        <swe:component xlink:href="urn:ogc:def:property:OGC::RelativeHumidity"/>
        <swe:component xlink:href="urn:ogc:def:property:OGC::WindSpeed"/>
        <swe:component xlink:href="urn:ogc:def:property:OGC::WindDirection"/>
       </swe:CompositePhenomenon>
      </observedProperty>
      <observedProperty xlink:href="urn:ogc:def:property:OGC::AirTemperature"/>
      <observedProperty xlink:href="urn:ogc:def:property:OGC::AtmosphericPressure"/>
      <observedProperty xlink:href="urn:ogc:def:property:OGC::RelativeHumidity"/>
      <observedProperty xlink:href="urn:ogc:def:property:OGC::WindSpeed"/>
      <observedProperty xlink:href="urn:ogc:def:property:OGC::WindDirection"/>
      <featureOfInterest xlink:href="936DA01F-9ABD-4d9d-80C7-02AF85C822A8"/>
      <responseFormat>text/xml;subtype=&amp;quot;OM/1.0.0&amp;quot;
      </responseFormat>
      <resultModel>om:Observation</resultModel>
      <responseMode>inline</responseMode>
     </ObservationOffering>
    </ObservationOfferingList>
   </Contents>
  </Capabilities>
```

DescribeSensor

Request

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<DescribeSensor xmlns="http://www.opengis.net/sos/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opengis.net/sos/1.0
     http://schemas.opengis.net/sos/1.0.0/sosAll.xsd"
    outputFormat="text/xml;subtype=&quot;SensorML/1.0.1&quot;"
    service="SOS" version="1.0.0">
 <procedure>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</procedure>
</DescribeSensor>
```

Response

```
<?xml version="1.0" encoding="UTF-8"?>
<sml:Component xmlns="http://www.opengis.net/sos/1.0"
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:ogc="http://www.opengis.net/ogc"
    xmlns:om="http://www.opengis.net/om/1.0"
    xmlns:ows="http://www.opengeospatial.net/ows"
    xmlns:sml="http://www.opengis.net/sensorML/1.0.1"
    xmlns:swe="http://www.opengis.net/swe/1.0.1"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opengis.net/sensorML/1.0.1
     http://schemas.opengis.net/sensorML/1.0.1/sensorML.xsd">
 <sml:identification>
  <sml:IdentifierList>
   <sml:identifier name="uniqueID">
    <sml:Term definition="urn:ogc:def:identifier:OGC::uniqueID">
     <sml:value>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</sml:value>
    </sml:Term>
   </sml:identifier>
   <sml:identifier name="longName">
    <sml:Term>
     <sml:value>IMEGO Medium Sized Sensor Module</sml:value>
    </sml:Term>
   </sml:identifier>
   <sml:identifier name="shortName">
    <sml:Term>
     <sml:value>imego_mssm</sml:value>
    </sml:Term>
   </sml:identifier>
   <sml:identifier name="manufacturer">
    <sml:Term>
     <sml:value>IMEGO</sml:value>
    </sml:Term>
   </sml:identifier>
   <sml:identifier name="operator">
    <sml:Term>
     <sml:value>rmueller</sml:value>
    </sml:Term>
   </sml:identifier>
  </sml:IdentifierList>
 </sml:identification>
 <sml:contact>
  <sml:ResponsibleParty>
   <sml:organizationName>IMEGO</sml:organizationName>
   <sml:contactInfo>
    <sml:onlineResource xlink:href="http://www.imego.com"/>
   </sml:contactInfo>
  </sml:ResponsibleParty>
 </sml:contact>
 <sml:location>
  <gml:Point srsName="urn:ogc:def:crs:EPSG:6.14:4979">
   <gml:pos>52.0 8.67 50.0</gml:pos>
  </gml:Point>
 </sml:location>
 <sml:inputs>
  <sml:InputList>
   <sml:input name="time">
    <swe:ObservableProperty definition="urn:ogc:def:property:OGC::Time"/>
   </sml:input>
   <sml:input name="temperature celsius">
    <swe:ObservableProperty definition="urn:ogc:def:property:OGC::AirTemperature"/>
   </sml:input>
   <sml:input name="pressure">
```

```
      <swe:ObservableProperty
       definition="urn:ogc:def:property:OGC::AtmosphericPressure"/>
     </sml:input>
     <sml:input name="percentage">
      <swe:ObservableProperty
       definition="urn:ogc:def:property:OGC::RelativeHumidity"/>
     </sml:input>
     <sml:input name="speed">
      <swe:ObservableProperty definition="urn:ogc:def:property:OGC::WindSpeed"/>
     </sml:input>
     <sml:input name="angle">
      <swe:ObservableProperty definition="urn:ogc:def:property:OGC::WindDirection"/>
     </sml:input>
    </sml:InputList>
   </sml:inputs>
   <sml:outputs>
    <sml:OutputList>
     <sml:output name="outputData">
      <swe:DataRecord>
       <swe:field name="samplingTime">
        <swe:Time definition="urn:ogc:def:property:OGC::SamplingTime"
         referenceFrame="urn:ogc:def:crs:OGC:UTC">
         <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
        </swe:Time>
       </swe:field>
       <swe:field name="temperature celsius">
        <swe:Quantity definition="urn:ogc:def:property:OGC::AirTemperature">
         <swe:uom code="Cel"/>
        </swe:Quantity>
       </swe:field>
       <swe:field name="pressure">
        <swe:Quantity definition="urn:ogc:def:property:OGC::AtmosphericPressure">
         <swe:uom code="Mbar"/>
        </swe:Quantity>
       </swe:field>
       <swe:field name="percentage">
        <swe:Quantity definition="urn:ogc:def:property:OGC::RelativeHumidity">
         <swe:uom code="%"/>
        </swe:Quantity>
       </swe:field>
       <swe:field name="speed">
        <swe:Quantity definition="urn:ogc:def:property:OGC::WindSpeed">
         <swe:uom code="m/s"/>
        </swe:Quantity>
       </swe:field>
       <swe:field name="angle">
        <swe:Quantity definition="urn:ogc:def:property:OGC::WindDirection">
         <swe:uom code="deg"/>
        </swe:Quantity>
       </swe:field>
      </swe:DataRecord>
     </sml:output>
    </sml:OutputList>
   </sml:outputs>
   <sml:parameters>
    <sml:ParameterList>
     <sml:parameter name="samplingFrequency">
      <swe:Quantity definition="urn:ogc:def:property:OGC::SamplingFrequency">
       <gml:description>
        Set the sensor to measure with the given frequency.
       </gml:description>
       <swe:uom code="Hz"/>
      </swe:Quantity>
     </sml:parameter>
    </sml:ParameterList>
   </sml:parameters>
   <sml:method xlink:href="urn:ogc:def:process:OGC::detector"/>
  </sml:Component>
```

GetObservation

Request

```
<?xml version="1.0" encoding="UTF-8"?>
<GetObservation xmlns:ogc="http://www.opengis.net/ogc"
    xmlns="http://www.opengis.net/sos/1.0"
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opengis.net/sos/1.0
     http://schemas.opengis.net/sos/1.0.0/sosAll.xsd"
    service="SOS" version="1.0.0">
 <offering>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</offering>
 <eventTime>
  <ogc:TM_After>
   <ogc:PropertyName>om:samplingTime</ogc:PropertyName>
   <gml:TimeInstant>
    <gml:timePosition>2009-08-20T11:39:50+01:00</gml:timePosition>
   </gml:TimeInstant>
  </ogc:TM_After>
 </eventTime>
 <procedure>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</procedure>
 <observedProperty>319C201F-9000-47dd-3258-835169543B9</observedProperty>
 <responseFormat>text/xml;subtype=&quot;om/1.0.0&quot;</responseFormat>
</GetObservation>
```

Response

```
<?xml version="1.0" encoding="UTF-8"?>
<om:Observation xmlns:gml="http://www.opengis.net/gml"
    xmlns:om="http://www.opengis.net/om/1.0"
    xmlns:sa="http://www.opengis.net/sampling/1.0"
    xmlns:swe="http://www.opengis.net/swe/1.0.1"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opengis.net/om/1.0
     http://schemas.opengis.net/om/1.0.0/om.xsd
     http://www.opengis.net/sampling/1.0
     http://schemas.opengis.net/sampling/1.0.0/sampling.xsd
     http://www.opengis.net/swe/1.0.1
     http://schemas.opengis.net/sweCommon/1.0.1/swe.xsd">
 <om:samplingTime>
  <gml:TimePeriod xsi:type="gml:TimePeriodType">
   <gml:beginPosition>2009-08-20T12:40:00+02:00</gml:beginPosition>
   <gml:endPosition>2009-08-20T12:40:00+02:00</gml:endPosition>
  </gml:TimePeriod>
 </om:samplingTime>
```

```xml
<om:procedure xlink:href="936DA01F-9ABD-4d9d-80C7-02AF85C822A8"/>
<om:observedProperty>
 <swe:CompositePhenomenon dimension="5" gml:id="adHocPhenomenon">
  <gml:name>resultComponents</gml:name>
  <swe:component xlink:href="urn:ogc:def:property:OGC::AirTemperature"/>
  <swe:component xlink:href="urn:ogc:def:property:OGC::RelativeHumidity"/>
  <swe:component xlink:href="urn:ogc:def:property:OGC::AtmosphericPressure"/>
  <swe:component xlink:href="urn:ogc:def:property:OGC::WindSpeed"/>
  <swe:component xlink:href="urn:ogc:def:property:OGC::WindDirection"/>
 </swe:CompositePhenomenon>
</om:observedProperty>
<om:featureOfInterest>
 <sa:SamplingPoint>
  <gml:name>SF_P1</gml:name>
  <sa:sampledFeature xlink:href="urn:ogc:def:nil:OGC:unknown"/>
  <sa:position>
   <gml:Point>
    <gml:pos srsName="urn:ogc:def:crs:EPSG:6.14:4979">52.0 8.67 50.0</gml:pos>
   </gml:Point>
  </sa:position>
 </sa:SamplingPoint>
</om:featureOfInterest>
<om:result>
 <swe:DataArray>
  <swe:elementCount>
   <swe:Count>
    <swe:value>3</swe:value>
   </swe:Count>
  </swe:elementCount>
  <swe:elementType name="Components">
   <swe:SimpleDataRecord>
    <swe:field name="Time">
     <swe:Time definition="urn:ogc:data:time:iso8601"/>
    </swe:field>
    <swe:field name="Latitude">
     <swe:Quantity definition="urn:ogc:phenomenon:latitude:wgs84">
      <swe:uom code="deg"/>
     </swe:Quantity>
    </swe:field>
    <swe:field name="Longitude">
```

```
      <swe:Quantity definition="urn:ogc:phenomenon:longitude:wgs84">
        <swe:uom code="deg"/>
      </swe:Quantity>
    </swe:field>
    <swe:field name="Altitude">
      <swe:Quantity definition="urn:ogc:phenomenon:altitude:wgs84">
        <swe:uom code="m"/>
      </swe:Quantity>
    </swe:field>
    <swe:field name="airTemperature">
      <swe:Quantity definition="urn:ogc:def:property:OGC::AirTemperature">
        <swe:uom code="Cel"/>
      </swe:Quantity>
    </swe:field>
    <swe:field name="humidity">
      <swe:Quantity definition="urn:ogc:def:property:OGC::RelativeHumidity">
        <swe:uom code="%"/>
      </swe:Quantity>
    </swe:field>
    <swe:field name="atmosphericPressure">
      <swe:Quantity definition="urn:ogc:def:property:OGC::AtmosphericPressure">
        <swe:uom code="Mbar"/>
      </swe:Quantity>
    </swe:field>
    <swe:field name="windSpeed">
      <swe:Quantity definition="urn:ogc:def:property:OGC::WindSpeed">
        <swe:uom code="m/s"/>
      </swe:Quantity>
    </swe:field>
    <swe:field name="windDirection">
      <swe:Quantity definition="urn:ogc:def:property:OGC::WindDirection">
        <swe:uom code="deg"/>
      </swe:Quantity>
    </swe:field>
  </swe:SimpleDataRecord>
</swe:elementType>
<swe:encoding>
  <swe:TextBlock blockSeparator=";" decimalSeparator="." tokenSeparator=","/>
</swe:encoding>
<swe:values>2009-08-20T12:40:00+02:00,52.01,8.67,50.01,23.5,40,1013,3.4,180;
```

```
    2009-08-20T12:40:01+02:00,52.04,8.69,50.01,23.5,40,1013,3.4,180;
    2009-08-20T12:40:02+02:00,52.07,8.71,50.01,23.5,40,1013,3.4,180;
   </swe:values>
  </swe:DataArray>
 </om:result>
</om:Observation>
```

RegisterSensor

Request

```
<?xml version="1.0" encoding="UTF-8"?>
<RegisterSensor xmlns="http://www.opengis.net/sos/1.0"
    xmlns:om="http://www.opengis.net/om/1.0"
    xmlns:swe="http://www.opengis.net/swe/1.0.1"
    xmlns:sa="http://www.opengis.net/sampling/1.0"
    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"
    xsi:schemaLocation="http://www.opengis.net/sos/1.0
     http://schemas.opengis.net/sos/1.0.0/sosAll.xsd
     http://www.opengis.net/sensorML/1.0.1
     http://schemas.opengis.net/sensorML/1.0.1/sensorML.xsd
     http://www.opengis.net/om/1.0
     http://schemas.opengis.net/om/1.0.0/om.xsd
     http://www.opengis.net/sampling/1.0
     http://schemas.opengis.net/sampling/1.0.0/sampling.xsd
     http://www.opengis.net/swe/1.0.1
     http://schemas.opengis.net/sweCommon/1.0.1/swe.xsd"
    service="SOS" version="1.0.0">
 <SensorDescription>
  <sml:Component>
   <sml:identification>
    <sml:IdentifierList>
     <sml:identifier name="uniqueID">
      <sml:Term definition="urn:ogc:def:identifier:OGC::uniqueID">
       <sml:value>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</sml:value>
   </sml:Term>
  </sml:identifier>
  <sml:identifier name="longName">
   <sml:Term>
    <sml:value>IMEGO Medium Sized Sensor Module</sml:value>
   </sml:Term>
  </sml:identifier>
  <sml:identifier name="shortName">
   <sml:Term>
    <sml:value>imego_mssm</sml:value>
   </sml:Term>
  </sml:identifier>
  <sml:identifier name="manufacturer">
   <sml:Term>
    <sml:value>IMEGO</sml:value>
   </sml:Term>
  </sml:identifier>
  <sml:identifier name="operator">
   <sml:Term>
    <sml:value>Roland Mueller</sml:value>
   </sml:Term>
  </sml:identifier>
 </sml:IdentifierList>
</sml:identification>
<sml:contact>
 <sml:ResponsibleParty>
  <sml:organizationName>IMEGO</sml:organizationName>
  <sml:contactInfo>
   <sml:onlineResource xlink:href="http://www.imego.com/"/>
  </sml:contactInfo>
 </sml:ResponsibleParty>
</sml:contact>
<sml:location>
 <gml:Point srsName="urn:ogc:def:crs:EPSG:6.14:4979">
  <gml:pos>52 8.67 50</gml:pos>
 </gml:Point>
</sml:location>
<sml:inputs>
 <sml:InputList>
  <sml:input name="time">
```

```
      <swe:ObservableProperty definition="urn:ogc:def:property:OGC::Time"/>
     </sml:input>
     <sml:input name="atmosphericTemperature">
      <swe:ObservableProperty
       definition="urn:ogc:def:property:OGC::AirTemperature"/>
     </sml:input>
     <sml:input name="atmosphericPressure">
      <swe:ObservableProperty
       definition="urn:ogc:def:property:OGC::AtmosphericPressure"/>
     </sml:input>
     <sml:input name="humidity">
      <swe:ObservableProperty
       definition="urn:ogc:def:property:OGC::RelativeHumidity"/>
     </sml:input>
     <sml:input name="windSpeed">
      <swe:ObservableProperty definition="urn:ogc:def:property:OGC::WindSpeed"/>
     </sml:input>
     <sml:input name="windDirection">
      <swe:ObservableProperty definition="urn:ogc:def:property:OGC::WindDirection"/>
     </sml:input>
    </sml:InputList>
   </sml:inputs>
   <sml:outputs>
    <sml:OutputList>
     <sml:output name="outputData">
      <swe:DataRecord>
       <swe:field name="samplingTime">
        <swe:Time definition="urn:ogc:def:property:OGC::SamplingTime"
         referenceFrame="urn:ogc:def:crs:OGC:UTC">
         <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
        </swe:Time>
       </swe:field>
       <swe:field name="airTemperature">
        <swe:Quantity definition="urn:ogc:def:property:OGC::AirTemperature">
         <swe:uom code="Cel"/>
        </swe:Quantity>
       </swe:field>
       <swe:field name="humidity">
        <swe:Quantity definition="urn:ogc:def:property:OGC::RelativeHumidity">
         <swe:uom code="%"/>
        </swe:Quantity>
       </swe:field>
       <swe:field name="atmosphericPressure">
        <swe:Quantity definition="urn:ogc:def:property:OGC::AtmosphericPressure">
         <swe:uom code="Mbar"/>
        </swe:Quantity>
       </swe:field>
       <swe:field name="windSpeed">
        <swe:Quantity definition="urn:ogc:def:property:OGC::WindSpeed">
         <swe:uom code="m/s"/>
        </swe:Quantity>
       </swe:field>
       <swe:field name="windDirection">
        <swe:Quantity definition="urn:ogc:def:property:OGC::WindDirection">
         <swe:uom code="deg"/>
        </swe:Quantity>
       </swe:field>
      </swe:DataRecord>
     </sml:output>
    </sml:OutputList>
   </sml:outputs>
   <sml:parameters>
    <sml:ParameterList>
     <sml:parameter name="time">
      <swe:Time definition="urn:ogc:def:property:OGC::Time"
       referenceFrame="urn:ogc:def:crs:OGC:UTC">
       <gml:description>Resets the sensors internal clock
        to the given date and time.</gml:description>
       <swe:uom xlink:href="urn:ogc:def:unit:ISO:8601"/>
      </swe:Time>
     </sml:parameter>
     <sml:parameter name="samplingFrequency">
      <swe:Quantity definition="urn:ogc:def:property:OGC::SamplingFrequency">
       <gml:description>Set the sensor to measure with the
        given frequency.</gml:description>
       <swe:uom code="Hz"/>
      </swe:Quantity>
     </sml:parameter>
    </sml:ParameterList>
   </sml:parameters>
   <sml:method xlink:href="urn:ogc:def:process:OGC::detector"/>
  </sml:Component>
 </SensorDescription>
 <ObservationTemplate>
  <om:Observation>
   <om:samplingTime/>
   <om:procedure/>
   <om:observedProperty>
    <swe:CompositePhenomenon gml:id="adHocPhenomenon" dimension="5">
     <gml:description>Ad hoc phenomenon - best practice when measuring multiple
      properties at the same time for a given feature of interest.</gml:description>
     <gml:name codeSpace="urn:ogc:tc:arch:doc-bp(xx-xxx)">
      319C201F-9000-47dd-3258-835169543B9</gml:name>
     <gml:name>ad hoc compound phenomenon</gml:name>
     <swe:component xlink:href="urn:ogc:def:property:OGC::AirTemperature"/>
     <swe:component xlink:href="urn:ogc:def:property:OGC::RelativeHumidity"/>
     <swe:component xlink:href="urn:ogc:def:property:OGC::AtmosphericPressure"/>
     <swe:component xlink:href="urn:ogc:def:property:OGC::WindSpeed"/>
     <swe:component xlink:href="urn:ogc:def:property:OGC::WindDirection"/>
    </swe:CompositePhenomenon>
   </om:observedProperty>
   <om:featureOfInterest>
    <sa:SamplingPoint>
     <sa:sampledFeature xlink:href="urn:ogc:def:nil:OGC:unknown"/>
     <sa:position>
      <gml:Point>
       <gml:pos srsName="urn:ogc:def:crs:EPSG:6.14:4979"/>
      </gml:Point>
     </sa:position>
    </sa:SamplingPoint>
   </om:featureOfInterest>
   <om:result xsi:type="swe:DataRecordPropertyType">
    <swe:DataRecord>
     <swe:field name="airTemperature">
      <swe:Quantity definition="urn:ogc:def:property:OGC::AirTemperature">
       <swe:uom code="Cel"/>
      </swe:Quantity>
     </swe:field>
     <swe:field name="humidity">
      <swe:Quantity definition="urn:ogc:def:property:OGC::RelativeHumidity">
```

```
        <swe:uom code="%"/>
      </swe:Quantity>
    </swe:field>
    <swe:field name="atmosphericPressure">
      <swe:Quantity definition="urn:ogc:def:property:OGC::AtmosphericPressure">
        <swe:uom code="Mbar"/>
      </swe:Quantity>
    </swe:field>
    <swe:field name="windSpeed">
      <swe:Quantity definition="urn:ogc:def:property:OGC::WindSpeed">
        <swe:uom code="m/s"/>
      </swe:Quantity>
    </swe:field>
    <swe:field name="windDirection">
      <swe:Quantity definition="urn:ogc:def:property:OGC::WindDirection">
        <swe:uom code="deg"/>
      </swe:Quantity>
    </swe:field>
  </swe:DataRecord>
 </om:result>
</om:Observation>
</ObservationTemplate>
</RegisterSensor>
```

Response

```
<?xml version="1.0" encoding="UTF-8"?>
<RegisterSensorResponse xmlns="http://www.opengis.net/sos/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opengis.net/sos/1.0
     http://schemas.opengis.net/sos/1.0.0/sosAll.xsd">
 <AssignedSensorId>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</AssignedSensorId>
</RegisterSensorResponse>
```

InsertObservation

Request

```
<?xml version="1.0" encoding="UTF-8"?>
<InsertObservation xmlns="http://www.opengis.net/sos/1.0"
    xmlns:om="http://www.opengis.net/om/1.0"
    xmlns:sa="http://www.opengis.net/sampling/1.0"
    xmlns:swe="http://www.opengis.net/swe/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"
    xsi:schemaLocation="http://www.opengis.net/sos/1.0
     http://schemas.opengis.net/sos/1.0.0/sosAll.xsd
     http://www.opengis.net/sampling/1.0
     http://schemas.opengis.net/sampling/1.0.0/sampling.xsd
     http://www.opengis.net/swe/1.0.1
     http://schemas.opengis.net/sweCommon/1.0.1/swe.xsd"
    service="SOS" version="1.0.0">
 <AssignedSensorId>936DA01F-9ABD-4d9d-80C7-02AF85C822A8</AssignedSensorId>
 <om:Observation gml:id="Obs1">
  <om:samplingTime>
   <gml:TimeInstant>
    <gml:timePosition>2009-08-20T11:40:00+01:00</gml:timePosition>
   </gml:TimeInstant>
  </om:samplingTime>
  <om:procedure xlink:href="936DA01F-9ABD-4d9d-80C7-02AF85C822A8"/>
  <om:observedProperty xlink:href="319C201F-9000-47dd-3258-835169543B9"/>
  <om:featureOfInterest>
   <sa:SamplingPoint>
    <gml:name>SF_P1</gml:name>
    <sa:sampledFeature xlink:href="urn:ogc:def:nil:OGC:unknown"/>
    <sa:relatedObservation xlink:href="#Obs1"/>
    <sa:position>
     <gml:Point>
      <gml:pos srsName="urn:ogc:def:crs:EPSG:6.14:4979">52 8.67 50</gml:pos>
     </gml:Point>
    </sa:position>
   </sa:SamplingPoint>
  </om:featureOfInterest>
  <om:result xsi:type="swe:DataRecordPropertyType">
   <swe:DataRecord>
    <swe:field name="airTemperature">
     <swe:Quantity definition="urn:ogc:def:property:OGC::AirTemperature">
      <swe:uom code="Cel"/>
      <swe:value>23.5</swe:value>
     </swe:Quantity>
    </swe:field>
    <swe:field name="humidity">
     <swe:Quantity definition="urn:ogc:def:property:OGC::RelativeHumidity">
      <swe:uom code="%"/>
      <swe:value>40</swe:value>
     </swe:Quantity>
    </swe:field>
    <swe:field name="atmosphericPressure">
     <swe:Quantity definition="urn:ogc:def:property:OGC::AtmosphericPressure">
      <swe:uom code="Mbar"/>
      <swe:value>1013</swe:value>
     </swe:Quantity>
    </swe:field>
    <swe:field name="windSpeed">
     <swe:Quantity definition="urn:ogc:def:property:OGC::WindSpeed">
      <swe:uom code="m/s"/>
      <swe:value>3.4</swe:value>
     </swe:Quantity>
    </swe:field>
    <swe:field name="windDirection">
     <swe:Quantity definition="urn:ogc:def:property:OGC::WindDirection">
      <swe:uom code="deg"/>
      <swe:value>180</swe:value>
     </swe:Quantity>
    </swe:field>
   </swe:DataRecord>
  </om:result>
 </om:Observation>
</InsertObservation>
```

Response

```xml
<?xml version="1.0" encoding="UTF-8"?>
<InsertObservationResponse xmlns="http://www.opengis.net/sos/1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.opengis.net/sos/1.0
     http://schemas.opengis.net/sos/1.0.0/sosAll.xsd">
 <AssignedObservationId>o_2</AssignedObservationId>
</InsertObservationResponse>
```