Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Department of Measurement and Information Systems

Monitoring Cyberphysical Systems

MSc Thesis

Készítette
Velinszky László

Konzulens
dr. Strausz György

May 1, 2015

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Velinszky László*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, May 1, 2015

Velinszky László
hallgató

Kivonat

Minden ember nap mint nap találkozik olyan szenzorokkal, melyek hálózatba kötve kommunikálnak egymással vagy egy központi egységgel. Ezek lehetnek akár időjárás érzékelők, utcai kamerák, mobiltelefonok. Az eszközök "Internet of things" (dolgok internetje) részévé válnak, hogy aztán a méréseket kiértékelve következtetéseket vonjunk le és automatikusan beavatkozhassunk a környezetünkbe. Az ilyen kiberfizikai rendszerek felett azonban nincsenek szemantikai kapcsolatok, általában a szenzorok információit csak adott feladatokra hazsnálják ki. Gazdaságosabb lehet az egyes szenzorokat több feladatra is felhasználni, ehhez azonban több dologra is szükség van. Egy részről szükséges egy szabványos elérés az adatok eléréséhez, ez a SensorML. Másodsorban szükséges egy értelmezési, szemantikai megközelítés mely alapján a szenzorok kereshetőek lehetnek. Nem utolsó sorban a két dolgot össze kell kapcsolni. A diplomamunkám témája az utóbbi részek megvalósításának jelen helyzete és egy mintaprogram kidolgozása, mellyel ezek a kiberfizikai rendszerek monitorozhatóak.

Abstract

Most of us encounters different sensors that act together while connected into a network every day. These can be either weather sensors or CCTV cameras or smart-phones. These devices become part of the Internet of Things by evaluating their observations and interacting with its environment. Such cyberphisical systems does not contain semantic connections between its sensors, they are only used for one purpose. It would be more efficient to be able to use each sensor for several tasks. Many things are needed for that. Firstly there should be a standard language for accessing information, this is called SensorML. Secondly, there shall exist a semantic knowledge about the sensors to make them easily search-able. In this thesis the latter parts are described and a sample implementation is introduced.

Introduction

Nowadays, most of our devices are connected through the Internet. Our computers, mobile phones, surveillance cameras share their information via the world wide web. Each device has many sensors, such as GPS, gravity, acceleration sensor, imaging devices or just processing powers. These sensors or resource's information are stored individually on each device.

The world is emerging into a state that information needs to be shared between peers and should be stored in an easy to reach independent location called Cloud. The Cloud is a distributed information store which can be reached through the internet. This enables sensor information to be stored and processed for new uses, because the same sensor can be used for different purposes. An outdoor surveillance camera can be used to protect from intruders or to provide weather information. This data can be used for different purposes in various regions. A local measurement can control the local heating system, but a grid of weather stations can be used to provide forecast data.

It is not enough to measure all the data using such sensors. It has to be stored for analytics as it can be a source for predictions. Storing has become easier in the BigData world we live in. There are expectations of the method of storing the data. It should be transparent letting different systems share information with each other. To create such a standardized way of storing sensor measurement data Open Geospatial Consorptium started and maintains the SensorMl format. One of SensorML's couple layers is the SOS (Sensor Observation Service) which provides different ways to reach the observed data. This storage engine makes it available for outer services and inner services (like virtual sensors) to reach the desired data and run analytics and trigger monitoring events on them.

Storing sensor data does not give any semantic knowledge about the system. Such knowledge can be that a wind sensor is also a weather sensor or a camera is a visual sensor. To describe such semantic connections an ontology has to be created. There are many ways to store ontologies. Usually they are described as triples like in the most common RDF format but there are newer formats which has extended capabilities and can describe natively much more things (such as temporal logic) like OWL format. Storing such ontologies are done using special RDF databases. Such databases can analyze connections faster it is even possible gain new knowledge from predefined conditions using reasoning. This way we can describe such things that if an Anemometer is a kind of wind sensors and wind sensors

are weather sensors than without specifying explicitly that the Anemometer is a weather sensor the system already knows the answer.

If we have such an enormous system it can be hard to maintain it. There can be thousands or even more sensors in a network with different capabilities and efficiency. The operation of the sensors should be monitored in an easy to access method, where each sensor's state should be easy to reach via a modern user interface. The purpose of this document is to describe such a large system's architecture and provide a tool with such a monitoring task can be done.

Such a system is described in this paper. The chosen format is a dynamic, interactive web page. The page consists of an engine that is capable of showing the state of the sensors using the RDF database. The sensor's state is displayed on a user friendly page. The sensors can be filtered to show only a group of sensors. A summary page for the error is also included to have a big picture of the state of the system.

In the next two section the basics of the used technologies are described. First there is a detailed introduction to the SOS server and the SensorML standard. After that the semantic description language and its storage engine is introduced. In the third chapter the actual system is shown which the monitoring supports. There will be a detailed introduction to the components and the used sensors and some use case for the system. The final chapter describes the monitoring system in details. It shows its structure, the used third party frameworks and a usage manual. In the end the whole work will be summarized.

Chapter 1

Introduction of SensorML and SensorWeb

1.1 Goals of observation gathering

Most physical quantities are measured with great accuracy on very basic devices. From the most basic sound measurements with microphone, to the nowadays popular inertial measurement units every data can be measured. There are also weather sensors and solar sensors available for everyone.

With the evolution of sensor fusion algorithms these data can be merged together to give an even better accuracy or a general knowledge of our environment. This data can be used to predict weather conditions by merging neighborhood sensors into one. It can be used to make predictions based on the weather conditions in an area and the direction of the wind on the location.

Figure 1.1. *SWE architecture*

The other challenge is to use a sensor's data for multiple purposes. For example a CCTV camera picture can be used to count the number of cars on the road, to get an estimation

of the speedings in a crossing or to get visual weather data. Computer Vision algorithms, regressions and special machine learning algorithms need different computing resources. Those can be outsourced to different computers which must have access to the data. The Sensor Observation Server is used to make this all available.

1.2 Types of sensors

Sensors are all around us. We have sensors in our cell phones or any handheld devices. Most of them are connected to the Internet or a network. The measurements of all of these devices can be stored and queried from a database. The usable data in some devices are:

- Smartphone
  - Camera: Visual image of the phone itself
  - Accelerometer: measurement of the acceleration of a system. Derivate of speed.
  - Weather sensors: Often smartphones has built in temperature sensors, rarely barometer is also installed.
  - Computing resource: ability to run additional softwares by using up its resources
  - Gyroscope: Orientation sensor.
- Weather sensor
  - Temperature sensor: Inner and outer temperature.
  - Humidity sensor: percentage of humidity.
  - Barometer: Air pressure.
- Beagleboard
  - Computing resource.

The examples show that a standard way to retrieve the measurements should include not only the measurement, but the type of the sensor, the measured unit, the location of the sensor and many additional information like this.

Storing and retrieving this information is done using the SensorML format.

1.3 The SensorML

The Open Geospatial Consortium approved the SensorML language to be able to describe all the necessary information about measured data[2]. This is an XML based language that is used to describe the sensors, add, update, delete or retrieve information. The SensorML

is able to solve the above mentioned problems. It is an abstract definition of the sensor information.

To have a general structure for the data, there is an abstract hierarchy that represents sensors and measurements. They have their own concepts which is described in as follows [4]:

- FeatureOfInterest: additional constraints provided for the measurement, e.g. time or location of measurement.
- ObservedProperty: what type of physical property is measured, e.g. speed, temperature.
- Procedure: the virtual or real sensor itself that is providing the data.
- Offering: the property itself that has been measured, e.g. wind speed, air humidity, traffic size.
- Result: the output of the procedure.

This way gives as enough abstraction to separate the measured properties, the physical properties, the actual sensors and the results from each other.

It is able to define the location of the sensor, the timestamp of the measurement and the sensor data itself, with many additional information. The data can be stored in SOS servers that can retrieve the measurements on different interfaces. There are additional other services[6] that are present to add demanded functionalities to the service. These other service models are SAS, SPS and WNS, which are introduced below and SOS will be covered in details.

1.4 Sensor Alerting Service

This service model takes care of the broadcast of the messages. The clients can subscribe to some special messages or alerts and they will be notified if such an event has occurred. SAS is some kind of event notification service.

1.5 Sensos Planning Service

This service provides a standard interface for controlling sensors and simulations. The SPS service implements the following tasks:

- Description of the Tasks.
- Verification of the feasibility of the tasks.

- Status check of the running tasks.
- Update or modification of running tasks.
- Removal of tasks.

1.6 Web Notification Service

This service connects users with the sensor informations. This model provides a standard interface for communication between clients and the service through HTTP, instant messaging, e-mail, SMS, phone or fax.

1.7 Server Observation Service

Server Observation Servers store SensorML data and let others query, manipulate and add sensors to the database. These sensors can be derived sensors. Such derived sensors are called procedures. A procedure can be a traffic information based on the CCTV camera. An SOS server should be able to handle dependencies based on which procedure requires other procedures to provide data. Sometimes derived sensors are also called virtual sensors, because they don't do physical measurement but provide a derived value.

There are many closed and open source implementations of SOS servers. Some open source implementations are introduced here.

The **MapServer** is written in C, C++, however it can be extended in many other languages[3]. It has a built in GUI to view data, however it is not yet available for the SOS implementation. The server can only be reached by one interface. It was originally developed by University of Minnesota. In the time of writing the paper the code is maintained and supported by the Open Source Geospatial Foundation. It is mostly used for mapping not for serving data for derived sensors. It was used in some projects between NASA and the University of Minnesota.

istSOS is another implementation in Python[11]. It runs its scripts from Apache web server just like MapServer. It uses PostgreSQL database backend to store values. It has a nice GUI for administration. Only supports standard SensorML interface to retrieve data. It is a project of the SUPSI University in Switzerland.

OOSThetys is a basic toolkit for enabling SensorML communication. It is written in Perl and Java. It is fairly documentated and seems to be abandoned because there has been no new release since the February of 2012.

52north SOS is a sample implementation written in Java[1]. It runs as a web service from Apache tomcat to serve requests. It also uses PostgreSQL with PostGIS extension. The application is used in the sample implementation and it is covered in details later.

All SOS servers and the whole SensorML standard is missing the semantic information about the sensors meaning that all sensor data are stored in such a database and all properties can be described for the sensors but no connection between the concepts can be described for the sensor.

1.8 52north SOS server

This implementation is done by the non profit organization with the identical name. The software supports many interfaces to query the information needed. The standard SOAP can be used to work with Java Web services. There are KVP and POX to retrieve or add data using standard GET queries. A big advantage is that 52north SOS supports JSON interface. It enables JSON queries to retrieve information efficiently in JavaScript, PHP, Python or other modern scripting languages.

The new 4.2 version is easy to install, it has a graphical user interface to set up the database connection and initialize the database. PostgreSQL with PostGIS is required. The basic usage is described on 52north webpage, but sample queries are shown in the built in test client.

The project is built with Maven and uses Spring framework too. The included unit tests ensure that the software's architecture is still consistent after compilation. The project was chosen to be part of Google's Summer of Code 2014 and the project gained a lot improvements from that.

A request can be sent using the connector of each interface. This used to be done by adding the interface name to the application url but in recent versions the service automatically detects the interface from the HTTP metadata. For example http://152.66.253.152:8080/52n-sos-webapp/service is the url where all interface can be reached. The first part is the host of the server. The tomcat application server is listening on port 8080. 52n-sos-webapp is the default name of the web application, service it the default connector for the webservice. The sent HTTP content-type header marks the interface. A detailed description of the interfaces can be seen on table 1.1.

1.9 SOS commands

There are many commands to query or manipulate the SOS server. In this section some of them will be shown using the JSON interface.

The GetCapabilities command (shown on Listing 1.1) allows the clients to retrieve a list of the reachable sensors of the server and their configuration. This command lists all the available measurements on the server too. This call does not have any required parameters, only if the details of the response should be controlled. Such detail is the section to be displayed. There are 4 different sections:

Interface	Content type	Description
SOAP	application/soap+xml	Standard SOAP request mostly used in JAVA, it is sending the request in XML format using POST request.
POX	application/xml	Very similar to SOAP but has a different XML format.
KVP	application/xml	This is a GET request, meaning that all request parameters are encoded in the URL. The answer is in XML format.
JSON	applicaton/json	This is a POST request which sends and receives information using JSON objects. Most programming languages can easily parse this type of request.

Table 1.1. *Interfaces for 52n SOS server*

- ServiceIdentification: the list of profiles that the server has been using. These profiles describe the format that the data is represented in.
- ServiceProvider: in this section is the contact information for the operator of the SOS service.
- OperationMetadata: this section provides the available options and their possible values.
- FilterCapabilities: what fields are available for filtering. Can be spatial filtering for location or temporal filtering for time range.
- Contents: summary of the type and FOI of each procedure.

Listing 1.1. *JSON getCapabilities POST request*

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

If the procedure is known the DescribeSensor command will describe the measurements of the selected sensor. The response will include a description field which contains the details about the sensor in the standard XML based SensorML format. The sample request code can be seen on Listing 1.2.

Listing 1.2. *JSON DescribeSensor POST request*

```
{
```

```
  "request": "DescribeSensor",
  "service": "SOS",
  "version": "2.0.0",
  "procedure": "http://www.52north.org/test/procedure/1",
  "procedureDescriptionFormat": "http://www.opengis.net/sensorML/1.0.1"
}
```

After knowing the necessary parameters the getObservation property can be called. This command shows the measured values for some given observations. The input can be complex, it can filter based on procedures, offerings, observed properties, on any feature of interest or temporal or spatial filters. A sample can be read from Listing 1.3.

Listing 1.3. *JSON GetObservation POST request with all filters*

```
{
  "request": "GetObservation",
  "service": "SOS",
  "version": "2.0.0",
  "procedure": [
    "http://www.52north.org/test/procedure/6",
    "http://www.52north.org/test/procedure/1"
  ],
  "offering": [
    "http://www.52north.org/test/offering/6",
    "http://www.52north.org/test/offering/1"
  ],
  "observedProperty": [
    "http://www.52north.org/test/observableProperty/1",
    "http://www.52north.org/test/observableProperty/6"
  ],
  "featureOfInterest": [
    "http://www.52north.org/test/featureOfInterest/6",
    "http://www.52north.org/test/featureOfInterest/1"
  ],
  "spatialFilter": {
    "bbox": {
      "ref": "om:featureOfInterest/sams:SF_SpatialSamplingFeature/sams:shape",
      "value": {
        "type": "Polygon",
        "coordinates": [
          [ [ 50, 7 ],
            [ 53, 7 ],
            [ 53, 10],
            [ 50, 10],
            [ 50, 7 ]
          ]
        ]
      }
    }
  },
  "temporalFilter": [
    {
      "during": {
        "ref": "om:phenomenonTime",
        "value": [
          "2012-11-19T14:00:00+01:00",
          "2012-11-19T14:05:00+01:00"
        ]
      }
```

```
    },
    {
      "equals": {
        "ref": "om:phenomenonTime",
        "value": "2012-11-19T14:08:00+01:00"
      }
    }
  ]
}
```

The response (on Listing 1.4) will contain the usual header which is contained by all the responses. It tells the requester the version of the response. After that comes the list of the observations that fit the conditions given. In the observation objects there are the procedures that measured the result, the feature of interests and the observable property that has been measured. The result object contains the uom field which is an abbreviation for the unit of measurement and the specific value. It can be an integer value, floating point or some encoded binary data.

Listing 1.4. *JSON GetObservation response*

```
{
    "request": "GetObservation",
    "version": "2.0.0",
    "service": "SOS",
    "observations": [
        {
            "type":
                "http://www.og.net/def/observationType/OGC-OM/2.0/OM_Measurement",
            "procedure": "http://www.52north.org/test/procedure/6",
            "offering": "http://www.52north.org/test/offering/6",
            "observableProperty":
                "http://www.52north.org/test/observableProperty/6",
            "featureOfInterest": {
                "identifier": {
                    "codespace": "http://www.opengis.net/def/nil/OGC/0/unknown",
                    "value": "http://www.52north.org/test/featureOfInterest/6"
                },
                "sampledFeature":
                    "http://www.52north.org/test/featureOfInterest/world",
                "geometry": {
                    "type": "Point",
                    "coordinates": [
                        51.447722,
                        7.270806
                    ]
                }
            },
            "phenomenonTime": "2012-11-19T13:09:00.000Z",
            "resultTime": "2012-11-19T13:09:00.000Z",
            "result": {
                "uom": "test_unit_6",
                "value": 2.9
            }
        },
        {
            "type":
                "http://www.og.net/def/observationType/OGC-OM/2.0/OM_Measurement",
            "procedure": "http://www.52north.org/test/procedure/6",
```

```
            "offering": "http://www.52north.org/test/offering/6",
            "observableProperty":
                "http://www.52north.org/test/observableProperty/6",
            "featureOfInterest": {
                "identifier": {
                    "codespace": "http://www.opengis.net/def/nil/OGC/0/unknown",
                    "value": "http://www.52north.org/test/featureOfInterest/6"
                },
                "sampledFeature":
                    "http://www.52north.org/test/featureOfInterest/world",
                "geometry": {
                    "type": "Point",
                    "coordinates": [
                        51.447722,
                        7.270806
                    ]
                }
            },
            "phenomenonTime": "2012-11-19T13:03:00.000Z",
            "resultTime": "2012-11-19T13:03:00.000Z",
            "result": {
                "uom": "test_unit_6",
                "value": 2.3
            }
        }
    ]
}
```

If only the result fields are important to the user the shorter GetResult command can be used. It is shown on Figure 1.5.

Listing 1.5. *JSON minimal GetResult POST request*

```
{
  "request": "GetResult",
  "service": "SOS",
  "version": "2.0.0",
  "offering": "http://www.52north.org/test/offering/6",
  "observedProperty": "http://www.52north.org/test/observableProperty/6"
}
```

The response will be a list of values of the last measurement for the queried properties. The results will be given as a concatenated string where each measurement is separated by a comma and in each measurement the value and its timestamp is separated by a hash mark. It can be seen on Figure 1.6.

Listing 1.6. *JSON GetResult response*

```
{
    "request": "GetResult",
    "version": "2.0.0",
    "service": "SOS",
    "resultValues": "10#2012-11-19T13:00:00.000Z,2012-11-19T13:00:00.000Z,
    2.0#2012-11-19T13:01:00.000Z,2012-11-19T13:01:00.000Z,
    2.1#2012-11-19T13:02:00.000Z,2012-11-19T13:02:00.000Z,
    2.2#2012-11-19T13:03:00.000Z,2012-11-19T13:03:00.000Z,
    2.3#2012-11-19T13:04:00.000Z,2012-11-19T13:04:00.000Z,
    2.4#2012-11-19T13:05:00.000Z,2012-11-19T13:05:00.000Z,
```

```
    2.5#2012-11-19T13:06:00.000Z,2012-11-19T13:06:00.000Z,
    2.6#2012-11-19T13:07:00.000Z,2012-11-19T13:07:00.000Z,
    2.7#2012-11-19T13:08:00.000Z,2012-11-19T13:08:00.000Z,
    2.8#2012-11-19T13:09:00.000Z,2012-11-19T13:09:00.000Z,
    2.9"
}
```

There are other commands that manipulate the database, like InsertSensor, DeleteSensor to add or remove sensors. There are commands to add observable properties, and add results to the database too.

Client applications use these services to communicate with the servers. These applications show the user a user friendly interface to read the measured data or their locations on a map. Such systems are described in the following sections.

1.10 SOS client applications

As the format introduced before is hard to decode for human there are various tools to display stored data in a user friendly way. With these tools measurements can be plotted in time, compared with each other and sensor locations can be displayed using a map. Even measurements can be plotted to a map giving a nice visual aid for processing the measured data.

52north has an official client application for the SOS server called 52n Sensorweb Client. This is a Java web application that can be configured to connect to multiple hosts and display their sensor data on maps and charts. It is developed using the same tools as the SOS server. This means that the application can be installed next to the SOS server without any extra needs. This client can be opened in a web browser, but some parts run on the backend. This is a relatively complex application that does not yet understand semantic connections between the procedures. A screenshot can be seen on Figure 1.2.

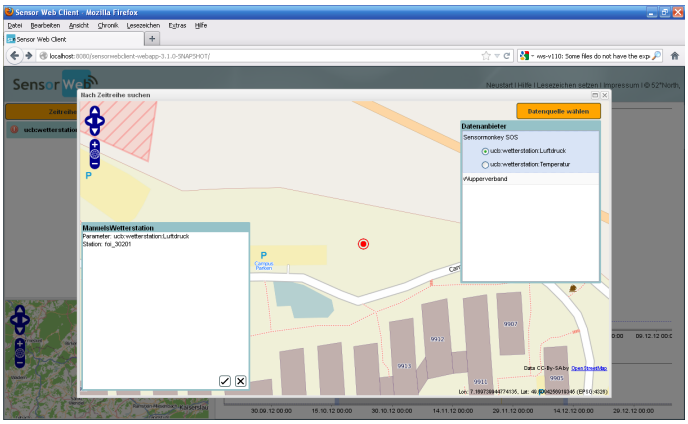

Figure 1.2. *Screenshot of SWE Client 3*

There is a JavaScript framework that enables users to use only client side tools to connect to SOS servers and display data called SOS.js. Unfortunately, because of security restrictions

on JavaScript cross-site request the data cannot be retrieved from the SOS server directly, the client has to be served from the same domain as the server. This is often not doable. That is why this application needs a proxy that forwards the data to the same domain to bypass this restriction. If the proxy is working, the framework can display everything just using the browser. Screenshot can be seen on Figure 1.3.

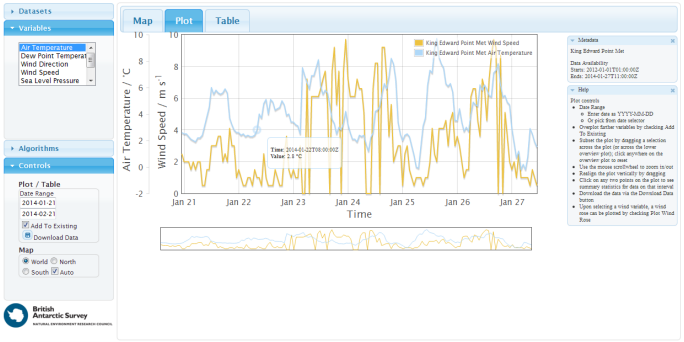

Figure 1.3. *Screenshot of sos.js*

There are external softwares that can display their own measurements but not the SensorML standard. To make them usable with 52north SOS the developers created tools to extend such existing softwares to be able to import data from SOS. Such extension is the ArcGIS extension which makes SOS data available to ArcGIS server. This is also available to other programs such as μDig.

There are other tools to export data to R language which is used to process large dataset for scientific purposes or to use with other Geographic Information Systems (GIS) softwares. However, the problem is that no client software has the ability to make search available by semantic connections. A client has to be extended with such information to enable convenient filtering when monitoring a cyberphisical system.

Fortunately there are standards to describe such systems and convert knowledge from one way to the other. This is described in the next chapter.

Chapter 2

Semantic Connection

2.1 Advantages of semantic information

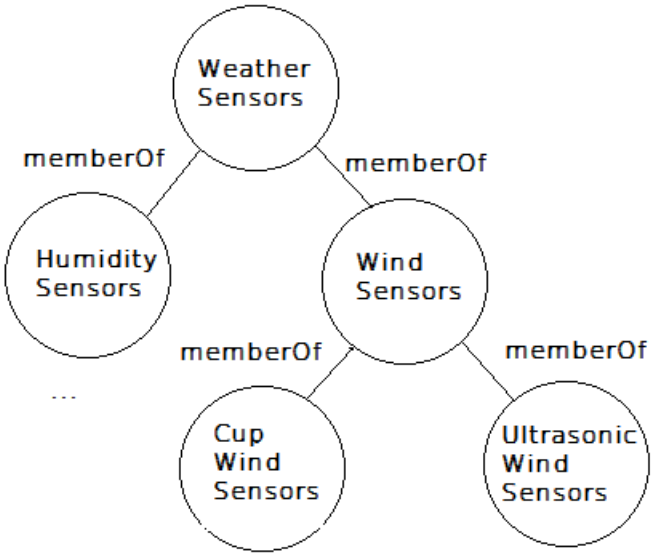

Figure 2.1. *Using triples in the example*

In a cyberphisical system stored in an SOS every sensor has a name and a type. These are identifiers that correspond to a sensor and there may be other descriptors that are not informative for human reader. Sensors can be called on different names like Anemometer, Wind instrument, wind meter that means the same, measures the speed of the wind. Filtering for name is not a good approach for filtering the information. To make monitoring simpler. A semantic information is needed that organizes the data in a way that is convenient for the human reader to read. The sensors should be separated into different groups and subgroups and these groups should have an easily distinguishable name. Such groups can be Phyisical sensors -> Weather Sensors -> Wind sensors -> Wind speed sensors. This information yet can not be stored on the SOS server a separate ontology database shall be created. The usual way to store such information is in an ontology. An ontology is a description where objects, their properties and the connections between objects are described. This is very

```
// classes
class Style {
 String neve;
}
class MusicalStyle
extends Style {
}
class Author {
 MusicalType musicalType;
}
// instances
MusicalType classic
 = new MusicalType();
Author Mozart =
 new Author();
Mozart.style = classic;
```

```
<!-- classes -->
<rdfs:Class rdf:id="Style"/>
<rdfs:Class rdf:id="MusicalStyle">
 <rdfs:subClassOf rdf:resource="#Style"/>
</rdfs:Class>
<rdfs:Class rdf:id="Author"/>
<!-- attributes -->
<rdf:Property rdf:id="StyleName">
 <rdfs:domain rdf:resource="#Style"/>
 <rdfs:range STRING/>
</rdf:Property>
<rdf:Property rdf:id="musicalStyle">
 <rdfs:domain rdf:resource="#Author"/>
 <rdfs:range rdf:resource="#MusicalStyle"/>
</rdf:Property>
<!-- instances -->
<MusicalStyle rdf:id="classic"/>
<Author rdf:id="Mozart">
 <MusicalStyle rdf:resource="#classic"/>
</Author>
```

Figure 2.2. *Java classes and their RDF representation side by side*

similar to classes, instances and relations in object oriented programming. An example of Java classes translated to an RDF ontology can be seen on Figure 2.2[5].

The most widespread ontology storage standards are the RDF databases.

2.2 Resource Definition Framework

The RDF standard[10] is created to be used with the Semantic Web approach to expand web pages with additional meanings that makes machines capable of understanding and reasoning about a web page. For example a web store can be easily understood by a customer: it has items, prices, shipping information, etc. However, for machines without saying explicitly that the value in one field is the price in USD it can be easily confused by the dimensions or the performance. Although nowadays these problems can be solved by machine learning, it is still a resource intensive process. To solve this problem an XML based standard has been introduced. In each page the data is explained using metadata information which is stored righ next to the HTML page itself. The pages can be directly referenced using their URL-s and anchors. The description is very simple, this is done by using triples.

The triple describes *which* page or entity (subject) is connected on *what* property(predicate) to *which* other page or entity(object). These subject, predicate and object triples are separate three unique values and it is called a statement. Each value could have another triple describing it. The first (subject) part is the resource which we make statements from. The

second parameter (predicate) is the property of the resource that represents the attribute, aspect or connection. The third parameter is the value(object) which can refer to other resources or it can be a value from a specific domain like a number. This is very similar to statements in real life, like:

- Anemometer measures wind-speed
- Anemometer is a wind-sensor
- Wind-sensor is a sensor

From the above example direct connections between wind-sensors and Anemometer can be described. Using reasoning indirect connections can be deduced. This makes RDF databases a powerful schema to describe real world examples. To be able to describe a big ontology which were put together using multiple ontologies each unique resource identifier is a fully qualified domain name and a hash tag and a unique name, like a URL with an anchor on a web page. The original idea behind the fully qualified domain name is that the approach was designed to give semantic meanings for web pages. Pages could refer to each other and give a global knowledge that is understandable by computers too.

Such recursive data can be represented in graph databases, where reasoning is only walking in the database. There are graph databases to store these triples and also dedicated RDF databases. The standard way to query RDF databases is using SPARQL queries. Although RDF can represent data and connections it can not describe the rules how the reasoning, the walks in the graph should be done. These are represented by OWL or SWRL which are an extension to the RDF. However, most RDF databases also support such rules.

Raw RDF descriptor files are stored in XML format. A comparison beween classes and their RDF representation can be seen on 2.2.

2.3 Some open source RDF databases

Apache Jena is an open source RDF datastore supported by the Apache foundation. It is written in Java[7]. It has many interfaces and supports many database backends. It can be used with in memory databases, SQL RDBs, triplestores. There is a built-in reasoner in the datastore, however it can be changed to other external reasoners too. The architecture of the software is shown on Figure 2.3.

OpenRDF Sesame is another tool for storing RDF triples. It is also written in Java. It has three different interfaces for communication: the SAIL API, the RIO interface and an HTTP client. The whole application runs from a Java container like Tomcat.

Apache Foundation also supports another RDF and Graph related project called Apache Marmotta. It was started in 2012. It supports a wide range of datastore engines from

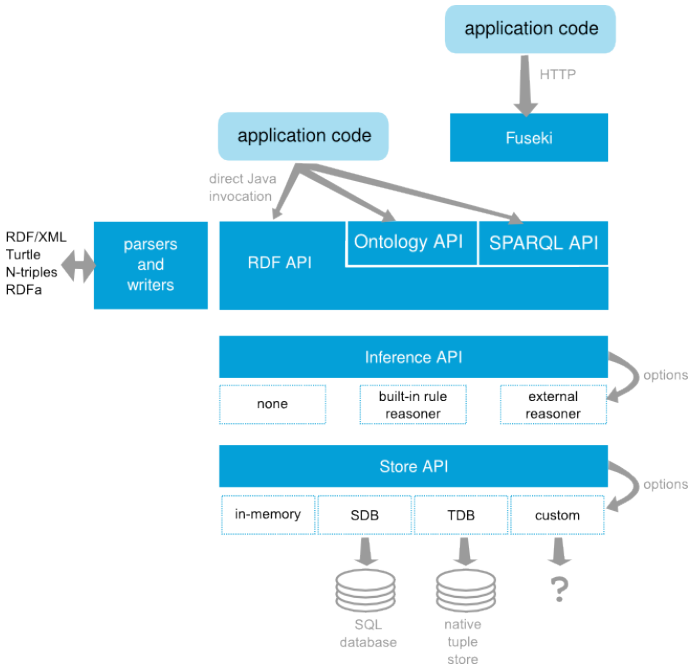

Figure 2.3. *Architecture of Apache Jena*

in memory databases through SQL servers to large distributed column bases stores like Apache Cassandra.

There is an out of the box tool that contains better reasoners, has a basic GUI and accepts different data types. This is Stardog database. It is a commercial application. However, it has a community version with a few restrictions. The software is written in Java and run from a web container, like Tomcat. It supports many interfaces such as HTTP and SNARL, it has a built in reasoner with integrated constraint validation. Connectors for different programming languages are ready to use. It can be easily queried using SPARQL. It has support for OWL 2 rules. Because it is an easy to use, out of the box tool this database is used in the sample application.

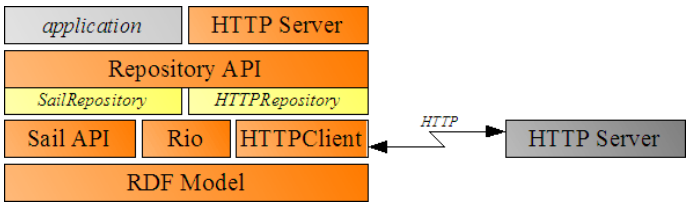

Figure 2.4. *Components of Sesame*

2.4 SPARQL for the queries

To retrieve the necessary metadata the SPARQL query language is used in the RDF databases. These queries are less human readable than standard SQL queries, although they look similar. The language builds on the subject-predicate-object triples. In a result all matching data is responded.

Listing 2.1. *Sample SPARQL that queries all filterable objects*

```
select ?s { <uri#filterable> <uri#subPropertyOf> ?s }
```

A SPARQL query can be quite complex. A simple query can be seen on listing 2.1. A query can start with a BASE attribute, which defines the default prefix for each attribute. When this base is defined, it can be neglected from the query itself. As these are usually long URL-s, this can have a big gain on query length. The second given attributes are the prefixes. If external nodes from external sources are referenced, it can be shortened by using a prefix. This also results a shorter query. The next given parameter can be after a SELECT keyword a list separated by commas of the fields that should be displayed in the output. These are the variables that have been used in the condition. The condition part starts with the WHERE keyword. After that comes a multiple list of triples, where one keyword is either displayed in the result or have a reference in the query chain. Two triple can be either connected with a comma, a semi-colon or a dot. If it is a dot it means that the statement is finished the subject, predicate, object values are given. Having a semi-colon as a separator means that the subject should be repeated, only the predicate and the object is needed. The comma repeats the subject and the predicate, this way only the other object is needed. The prefix is separated by a colon from these values. Filtering functions can be also given in a query. For example the FILTER function can have one parameter and a regular expression and it filters only those values which fulfill the given regular expression. It can also compare values. Results can be grouped, ordered, just like in standard SQL languages. Aggregation of results are also possible.

Predicates can be also used for recursive search. Listing every node of a subtree or leafs can be easier done in SPARQL. For selecting all the nodes, including the original, an asterisk has to be added to the end of the predicate. A plus sign will list every child element of the subject. These are called property paths and they are in the standard since version 1.1.

A SPARQL query can also provide other answers than returning a list of the answers. SELECT can be replaced by CONSTRUCT for creating new triples or by ASK which will check if the value exists in the repository.

Listing 2.2. *Sample complex SPARQL which shows top 5 countries based on population using DBpedia*

```
PREFIX type: <http://dbpedia.org/class/yago/>
PREFIX prop: <http://dbpedia.org/property/>
SELECT ?country_name ?population
WHERE {
?country a type:LandlockedCountries ;
rdfs:label ?country_name ;
prop:populationEstimate ?population .
FILTER (?population > 15000000 && langMatches(lang(?country_name), "en")) .
} ORDER BY DESC(?population) LIMIT 5
```

With such technology the triples can be stored and retrieved. However, the ontology behind the measurement is still missing. For that there is a standard ontology for sensors maintained by the Semantic Sensor Network Incubator Group. The underlying system which will be introduced in the next chapter is using this ontology in a somewhat customized

way[4]. The ontology is described in the next section.

2.5 The Semantic Sensor Network Ontology

The Semantic Sensor Network Incubator Group is a W3C incubator project. These projects are ran for one year to work on an area, this project ran from March of 2009 to September of 2010. It has developed an ontology for the semantic annotation of the Sensor Web Enablement standard. The ontology is available at http://www.w3.org/2005/Incubator/ssn/ssnx/ssn website, and it is conceptually organized into 10 modules. This is only a conceptual representation, the implementation consists of 41 concepts and 39 object properties. It is built upon a lightweight ontology, the DOLCE-UltraLite. This underlying ontology helps the new one to connect with other semantic databases and to have a better understanding of the concepts behind the properties.

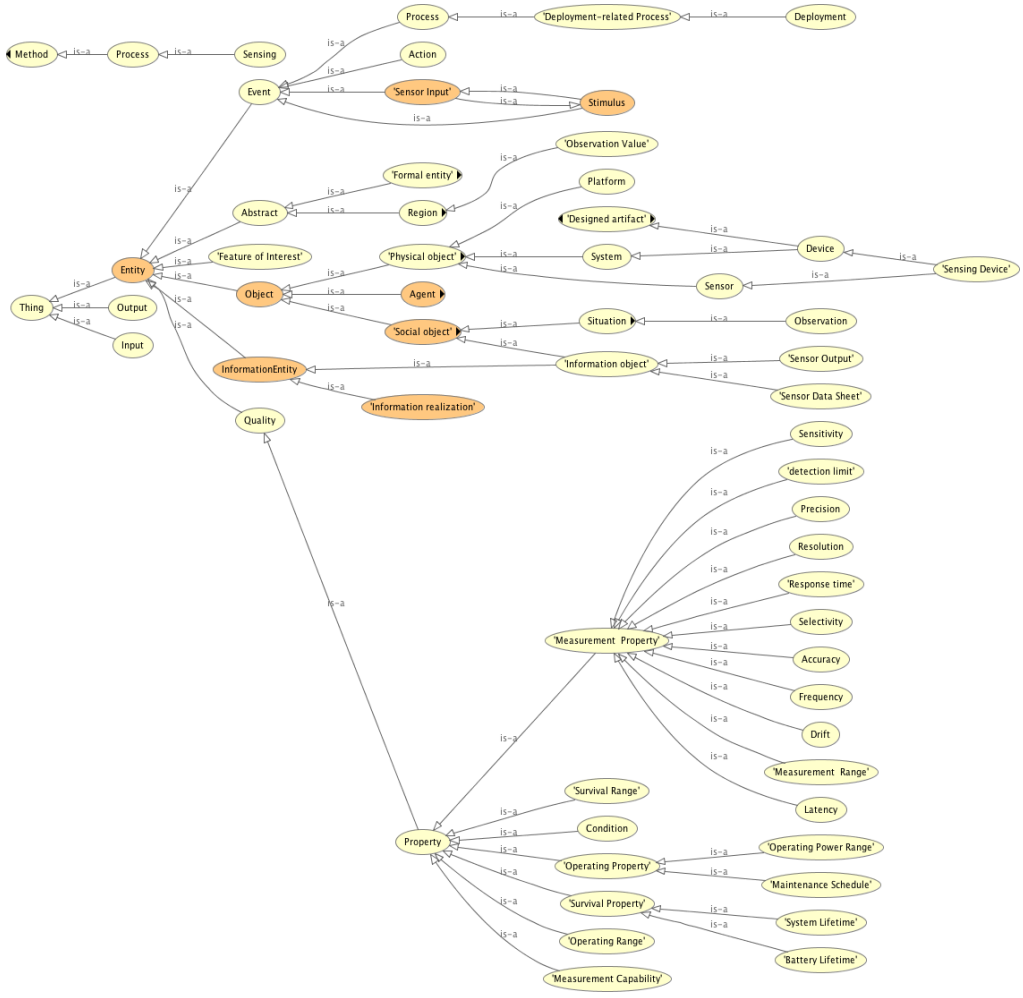

Figure 2.5. *Inferenced connections of the SSNO*

The ontology can be viewed from four different perspectives:

- A sensor perspective with focus on what and how it senses and what it sensed.

- An observation perspective which focuses on the observation and its environment.
- A system perspective in focus on the network itself and the deployment.
- A feature and property perspective which focuses on the physical properties and the subject of the observation.

This makes it possible not just to deduct that for example wind sensor is a kind of sensor but that wind speed is a speed measurement. The connections between each object of the SSN ontology using reasoning can be seen on Figure 2.5. In every ontology the ancestor is the Thing object. Its descendants are the Entities and the Input and Output objects. An entity can be an Event, like a measurement (sensor input), a process or an action. A sensor input is triggered by a Stimulus (stimulation of the sensor). The properties of the sensors are also Entities. There are two kind of properties, one is the property of the measurement and the other is the property of the sensor itself.

Chapter 3

The used cyberphysical system

3.1 Usage of the designed software

The designed monitoring software is made to be part of the system created for the Future Internet Research, Services and Technology project started by ETIK organization. The system is a prototype of a sensor network where the output of sensors can be used to create so called virtual sensors and store the data in a central data store. The system can reason using the ontology built on the sensors. A detailed introduction can be found in this chapter.

3.2 Goal of the project

The project's goal is to show a prototype of a cyberphisical system. It should contain sample sensors, a central database and also a planner, which plans the deployment of the designed new processes or virtual sensors. The system should prove using small example use cases that such a system is feasible to build. The prototype should not need to scale for large amount of sensors.

3.3 System architecture

The system was designed in the following way: There are computing or sensing resources called nodes which can provide computational capacity for the new application or sensor output using its sensors. The measurements are sent to the central SOS server and the available free resource information to the RDF store. The SOS server stores the measurements and provide the data to the other nodes. There is a translation module, which can create the RDF representation of the sensor database for the ontology. The RDF store contains the overall architecture of the deployed applications and resource nodes. It also stores the load of each node. There is a sensor browser module which users can use to search in the ontology. It is connected to a measurement browser which will display the

observations of the selected sensors. Users can deploy new application through the planner system. A package shall be uploaded and the planner should automatically deploy it to the chosen node. The status of the system can be seen on the implemented monitoring system.

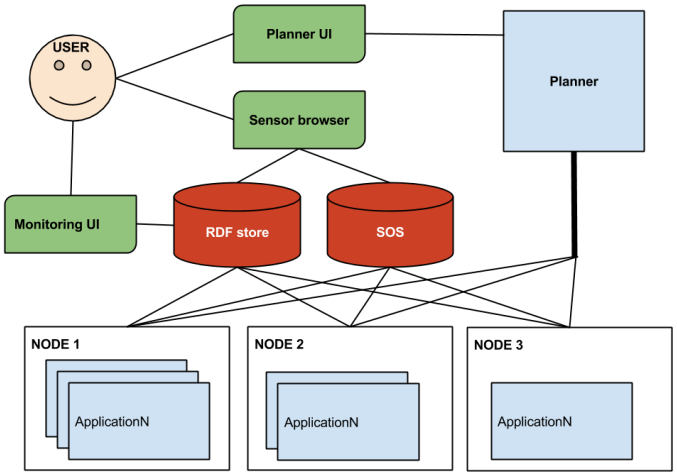

Figure 3.1. *System overview*

3.4 Sensor devices

The system is using different devices with different capabilities. There are simple microcontrollers and high performance mobile phones integrated in the system.

The most simple solution is the Arduino Uno board with Ethernet controller. The card has an AVR based microcontroller working inside. It is good for simple measurement. It contains 14 general purpose inputs and outputs from which 6 can be PWM output. It's computing speed is 16Mhz.

The next in computational power is the Raspberry PI computer. This computer was designed to be cheap, reliable and compact, so poor people in developing countries can afford it. There are two models, both of them are fully working desktop computer with 700Mhz ARM cpu. It has:

- 3 USB port
- 1 CSI port for raw camera input
- 1 Composite video output
- 1 HDMI output
- 1 SD/MMC/SDIO card reader
- 8 GPIO port (for UART, SPI, I2C, I2S audio)

It can run multiple operating systems, like Arch Linux, Raspbian OS, Debian, Slackware, etc. The more expensive model which costs around 35 dollars has a built in Ethernet port.

The most expensive embedded computer is the BeagleBone in the project. This computer was developed by Texas instruments. It has a 600Mhz ARM Cortex 8 processor with 128 MB of RAM. It has built in features for sound and video processing. It costs around 150 USD.

3.5 The Sensor Observation Service

The sensors communicate with the 52n SOS server using POX GET requests. To reach the service the sensors have to be in the same network or VPN to make the connection secure. The service has its own client installed next to it.

3.6 RDF store

The ontology is stored in an RDF database. It is based on the ontology introduced in the previous Chapter. The SSN Ontology has been extended for the custom use cases. The introduced ontology is called SISRO. It extends the original ontology in two ways:

- It describes concepts and relations present in SensorML (and missing from SSN)
- It contains hardware details of the sensor devices

For easy communication with the sensors and the SOS server, the system can be reached using a custom service. This Sensor Instance Semantic Registry Service contains the following interfaces:

- Collection of Sensor Metadata. This interface contains the metadata for discovering sensors.
- System Discovery Interface. This interface is responsible for discovering sensors, searching in them using semantic connections.
- Sensor Status Interface. This interface manages the status information of the sensors. This interface makes it possible to search based on Status Information and change states.
- Sensor and hardware interconnection interface. Sensors does not have hardware description in SensorML. This interface supports application installation information and reconfiguration.
- SPARQL interface. This interface lets users to interact with the database on a low level.

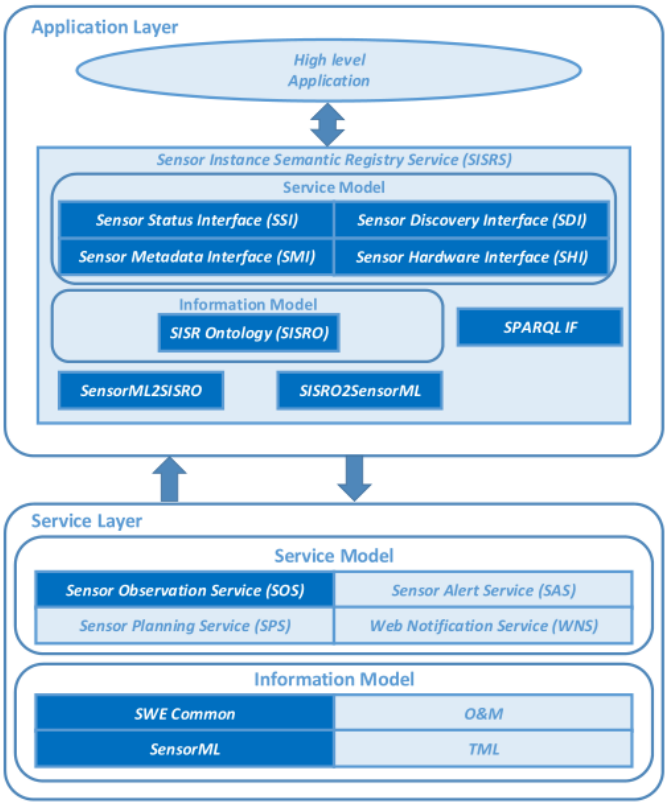

Figure 3.2. *SISR service layers*

The layer architecture can be seen on Figure 3.2.

The custom API has the following functions:

- GetCapabilities(): This function returns the sensor metadata from the database.
- SearchSensor(): This function makes it possible to query the sensors based on temporal, spatial or topical properties.
- DescribeSensor(): This function returns the SISRO ontology of a chosen sensor.
- UpdateSensorInfo(): This function enables modification of sensor metadata.
- GetSensorStatus(): This function retrieves sensor statuses.
- SetSensorStatus(): This function lets the users modify sensor statuses.
- AssignSensor(): This function assigns a sensor to a specific device.
- RemoveSensor(): This function removes a sensor from the device.

However, these services does not contain the necessary features to build the semantic connection between the sensors and their categories. That is why the implemented software uses the RDF databese directly.

3.7 Planner

The planer is a separate module that is capable of deploying different applications to the different resources. It is not used in the monitoring only its results, the deployed application.

3.8 Image processing example application

In the following example a a real life use case of the built cyberphisical system is solved. This example has been implemented in the project itself[12].

The goal was to check visual signals in a server room. There was a switch in the room what had blinking LED lights for indicating network connection. In this scenario we would like to monitor these connections only by using the LED indicators. The application is capturing a live video stream from the server room. In the video stream, the LEDs are seen. The user can specify the position of the monitored LEDs. The observations are categorized into 3 different categories:

- Switched off - The LED hasn't been turned on since 5 seconds
- Blinking
- Turned on - The LED hasn't been turned off in the past 5 second

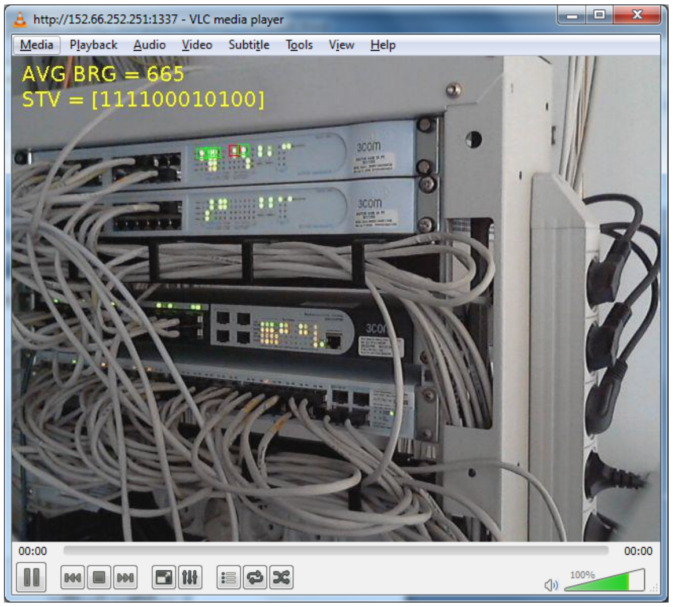

Figure 3.3. *Input image and corresponding output vectors*

The measurements are accumulated into a vector which projects each LED's state. Then the state vector is forwarded into the SOS database. The system also recognizes human interaction. If high movement is detected the system does not send the state vector but a marker signal that tells the system that human movement is detected. If there has been

no movement for a long time, the system should automatically recover and show the state of the LEDs. The sensor input is stored in a plain text file.

The video capturing is done by a Raspberry PI device. The stream is sent to a Beagleboard computer which is running the the image processing application. It has the text file with the configuration. The text file contain 5 different lines:

server the URL of the video stream.

username username for password protected streams.

password password for the protected stream.

sos URL for the POX endpoint of the SOS database.

point there can be multiple of this descriptor, which sets the X and Y coordinates of the LEDs

The input image and the output state vector can be seen on Figure 3.3.

3.9 The building monitoring example application

The use case is a typical infrastructure monitoring system. It consists of such devices that can be used for intruder detection or measuring usage of certain rooms. It is a demo environment which have been built up in room IE322 and the third floor corridor at the University. Detailed description of the devices follows.

Microphone

The microphone is used for sound analyzing it is used in virtual sensors. It has a streaming output that can be captured or processed further by other sensors. The device's output in SOS is a streaming URL which can be used to listen to the signal.

It has only one observable property, the stream URL, which is given as a text result by name "observableProperty/stream_url"

It is placed in IE322 room.

Eco sensor

Eco sensors are also physical sensors like the microphone device. They are motion sensors of intruder alert system. For that they have measure several physical properties which ensemble would be used in the output. These observable properties are:

- Temperature: measures temperature near the sensor. It is measured in degrees Celsius. It has the name observableProperty/temperature
- Motion: when motion is detected a boolean signal will be sent, which will have true value if motion is detected. The name of this property is observableProperty/motion.
- Luminance: Brightness in the room measured by the sensor. It is given in an inner unit as a floating point number. It is named observableProperty/luminance in the database.
- Battery voltage: the voltage of the used batteries. It is given in Volts. Using the measurement unexpected power losses can be prevented. Its name in the database is observableProperty/battery_voltage

There are 13 eco sensors in the system. 9 of them is in the corridor and 4 is in room IE322.

Door state sensor

These sensors monitor the closeness of windows and doors. They have a boolean property which represent if the door or windows is closed or not. Its only measured property is named observableProperty/door_state It is placed in IE322 room.

Speech sensor

The speech sensor is the only virtual sensor in the system. It listens to the microphone stream and tries to decode the sentences that has been said in the monitored room. It gives the decoded text as answer.

Chapter 4

The monitoring system

4.1 Goal of the implementation

After introducing the data formats and the cyberphisical system implementation, in this part a sample application is shown that can handle connection to both data types.

The goal was to show how a connection can be created from third party applications. In further chapters the integration of the different types will be shown. The meaning of this phase is to seek for new ways to implement a web application using the connectors provided and measure the needs for such a web application. A whole test system was created to enable this.

4.2 The test environment

The system has been tested on multiple platforms. There was an Ubuntu Linux based virtual machine, which had a limited 512 MB of RAM, and a max. 10 GB storage. It's network card was hidden behind a NAT provided by the host computer. Java Runtime Environment was installed on the virtual machine for Stardog and SOS. For the web application NodeJS has been set up.

The other platform was a Windows based computer. It had the same services (Stardog and SOS) running. The monitoring application requires NodeJs, which is a platform independent solution, so this did not mean any problem.

The chosen RDF database is Stardog Community edition. At first the software preallocates too much memory. It might be required for heavy weight applications but in our case it was not necessary. The startup script had to be changed to make the software start. Changing this parameter does not decreases the performance significantly, compared to another computer with more memory.

The chosen SOS server is 52north SOS 4.1 This is a recent release of the software. The used development version enables JSON communication. The server requires PostgreSQL database backend and Tomcat application server.

PostgreSQL 9.1.12 is used as the database backend for SOS. PostGIS environment had to be added. The database can be administrated using pgAdmin III from the host computer using port forward. For the Linux test environment no new users has been added, the SOS server uses the admin user to connect to the database. The database had to be created manually but tables and configuration is added during the installation automatically.

Tomcat 7 is installed to support SOS. To keep the system safe, on the Linux machine a new user is created to run the container and the application. The Stardog database is also running in a Tomcat like environment that is why it is started by the same user as the SOS server.

4.3 NodeJS: The backend of the monitoring application

The connector application is written in JavaScript. This way the frontend and the backend application can be written using the same programming language. Since the V8 engine exists JavaScript can be compiled and run significantly faster[9] than before. The engine was introduced in 2008 and it caused a breakthrough in Javascript language. It has some new solutions that made V8 more effective, these are:

- **Hidden classes**: In V8 every object is a class. They are stored in a tree based structure and can be recalled faster.
- **Dynamic Machine Code Generation**: V8 generates machine code from source without any intermediate bytecode language, like in Java.
- **Inline cache**: It has a built in inline cache for resolving method names quicker.
- **Efficient Garbage Collection**:Garbage collection freezes code execution and has a detailed map of the variable, however each garbage collection phase would only clear some part of the memory, it does not run full garbage collection every time.

NodeJS builds upon this V8 engine and lets JavaScript run on backend. Another advantage is that NodeJS has a single threaded, event driven core. This enables running applications faster, without worrying about thread safety. The architecture of the NodeJS environment can be seen on figure 4.1. To keep the integrator separated a different user is added to run server side code on the Linux system. No web application container or web server is needed to run a NodeJS application. The software itself handles TCP connections and other modules help do it similar to Java Servlets. Because it has smaller overhead and dependencies, the created application should run faster than a rich Java Web application. NodeJS has an easy to use packaging system that makes dependency handling simple, like maven for Java. All necessary modules names are added to the related part of the package.json file and required packages are downloaded using the npm install command from a central repository. On the Windows machine a fork of NodeJS, the IOJS engine was tested. It is a community driven alternative that is fully compatible with NodeJS.

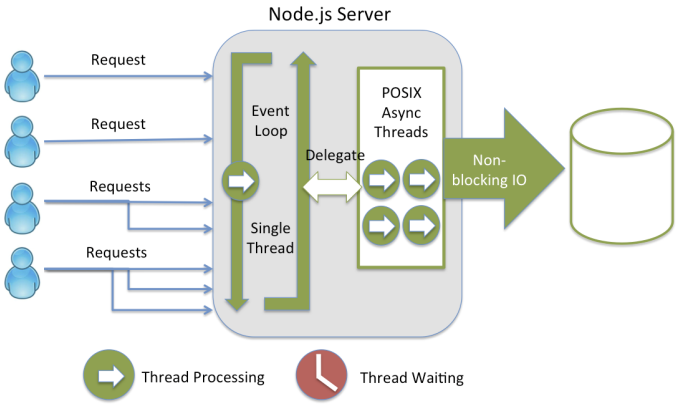

Figure 4.1. *NodeJS's event driven achitecture.*

4.4 Dependent modules

Using the NPM (Node Package Manager) new dependent packages can be easily downloaded from a central repository. NodeJS has a great coverage of packages, people can easily share their own modules. It contained 143 300 packages at the time of writing. In the implementation of the monitoring application there was a need for an interface for the database communication, interface for generating the web services and serving web pages. The most important packages were the following:

- The ExpressJS module acts as the Servler engine for JavaScript. It handles incomming connections and parsed HTTP request are passed to a callback function which generates the responses. Different urls can have different callback functions to do routing. Express also supports many templating engines, like EJS or JADE.
- Restler is an easy to use module that can asynchronously read a web page and parse it as JSON data and return it to the callback function. This can be used to communicate with the SOS server as it has an interface that is using JSON post requests to receive queries.
- Stardog.JS is Stardog database servers own connector to get SPARQL queries. It is in early stages, however all the necessary functions are working.
- Nodemon, Forever or PM2 are utilities that run NodeJS codes automatically, restarts them on error or code changes. There is a support for development, that on any sourcecode change the changes are pushed to the browser without the need to refresh the browser. They can be configured to restart on error.

4.5 AngularJS: the frontend of the monitoring application

The ExpressJS module is serving the static files from the backend. The webpage itself is a single page web application which means that it consists of one HTML page and some

partial pages that are loaded on the client side by a JavaScript framework. The framework that loads the different modules is called AngularJS. It was acquired by Google and now the project is maintained by them[8]. It's main advantage is that it implements a clear Model-View-Controller pattern. It dynamically changes the views, the HTML page based on the changes in the model, without additional coding. It is using dependency injection to be easily extensible.

For manipulating the user interface some external packages has been added to Angular. The most important of those are the following:

- **Chart.js**: This is a module that can be used to draw line charts or other kind of charts. This has an Angular integration and can be used to display the measured data in a time series plot. It can automatically smooth the curves. This is an open source project using HTML5.
- **Angular Bootstrap DateTime picker**: This module has been chosen to select the time period to display. This is an open source project too.
- **NG Tags Input**: This module can display tags in a search box and can handle autocomplete.

For formatting the page the Bootstrap template is used. This library has some built in styles that are displayed the same in every browser. It has support for mobile devices.

There are different building blocks for AngularJS.

- **Module**
- **Controller**
- **Directive**
- **Filter**

4.6 RDF representation of the SOS data

The general SISRO representation was missing some data of the example use case at the time of writing. To bridge this problem the monitoring application is using a different ontology that has been created to enable sensor browsing. This ontology contains only the necessary information to query sensor information. The elements can be seen on Figure 4.2. Some parts of the ontology hasn't been used in the monitoring system, the key components are:

- **Sensor**: This object is the ancestor of all sensor types. It has the Weather and Other sensors and their children. The instances of the child classes are the actual sensors.

Figure 4.2. *The Ontology used in the monitoring system.*

Figure 4.2 shows that a final sensor can have multiple measurement, thus it can have multiple parents. Eco sensors are instances of the CPSCorridorEcoSensor and door state sensors are Instances of CPSDoorStateSensors.

- **loc**: This object stores location information, like URI of the feature of interest property in SOS and the name of the location.
- **ObservedProperty**: This is the observed feature of a sensor. There can be multiple observed properties for one sensor, like Battey voltage, Motion sensors, Speech text. The object instance contains the URI for the ObservedProperty in SOS and the type of the observed property.
- **ObservedPopertyType**: This is the type of the observed property it has 3 instances in the example application: quantity, text, boolean.

The sensors of the use case are introduced in the end of the previous chapter, so those won't be covered here in details.

4.7 Architecture of the monitoring system

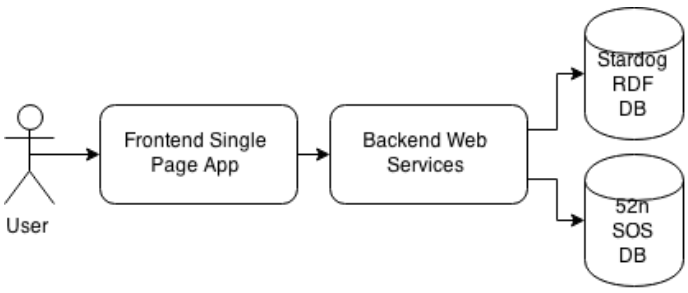

Figure 4.3. *Overall architecture of the monitoring system.*

The overall architecture can be seen on Figure 4.3. The user communicates through the single page web application. The web application dynamically loads the elements by sending requests to the backend service. The backend service connects to the two different databases and returns the data to the frontend. Initially the client side pages are also served by the backend service.

4.8 The backend web service

The backend services are called using HTTP POST and GET request. The POST body is always a JSON object. The detailed architecture of the backend services can be seen on Figure 4.4.

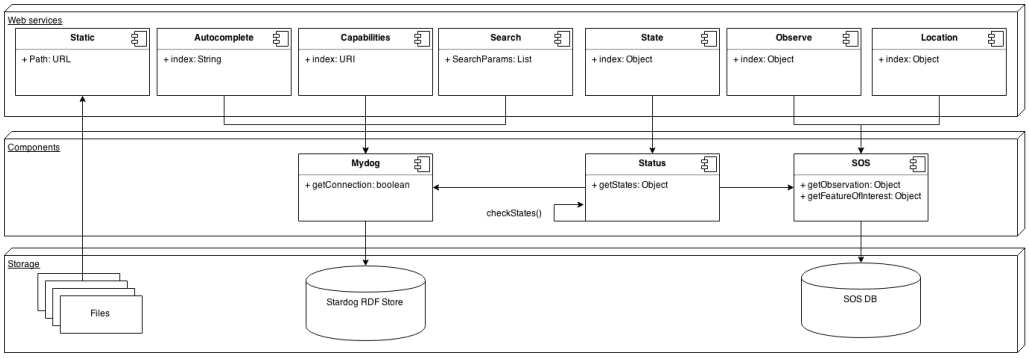

Figure 4.4. *Detailed architecture of the backend services.*

4.8.1 Serving static files

As the backend service is also providing the basic web server features, it has a built in static module, which in case there is no other web service by the provided location, it is serving a file from a given folder. If the file specified in the location does not exist the default index.html is returned. This ensures that Angular's custom routes are redirected to the same page. This feature is provided by ExpressJS.

4.8.2 Autocomplete service

In the sensor browser page, there is a search module where you can start typing in the name of the wanted category. After 3 letters a query is sent to the Autocomplete backend service which tries to find the matching categories and returns a list of the result. The Autocomplete service is using the Mydog module to get the data from Stardog. The query searches for every descendant of the sensor class and returns the result if found. The SPARQL query for that can be seen on Listing 4.1.

Listing 4.1. *SPARQL query for Autocomplete on the "wea" string*

```
SELECT ?name ?label {?name rdfs:subClassOf+ mit:sensor ; rdfs:label ?label. FILTER
    regex(?label, "wea","i")}';
```

4.8.3 Capabilities service

The Capabilities service receives a procedure and returns all the information that describe the sensor in some way. This is similar to what SOS getCapabilities would return but with additional information. It is done using SPARQL in the RDF store. The result of this service call is used in other parts of the application. The SPARQL query for this can be seen on Listing 4.2. The result will be given as a JSON object. It will contain the sensor name, the procedure name, the offering of the procedure, the feature of interest of the procedure, the label of the feature of interest, the observed property URI, its name, and the name of the type of the observed property.

Listing 4.2. *SPARQL query for Capabilities*

```
SELECT ?label ?procedure ?offering ?foi ?flabel ?observable ?obs_name ?tlabel
WHERE { <procedure/name> demo:procedure ?procedure ;
 rdfs:label ?label;
 demo:offering ?offering;
 demo:observedProperty ?obs ;
 mit:hasLoc ?loc.
?loc demo:foi ?foi;
 rdfs:label ?flabel.
?obs demo:observedPropertyURI ?observable ;
 rdfs:label ?obs_name ; mit:hasObservedPropertyType ?type .
?type rdfs:label ?tlabel}
```

4.8.4 Search service

The search service filters the sensor list. By default it will return the list of all sensors in the system. If there is an additional parameter list given in a given format the system will modify the query accordingly. This format is an array of JSON objects which have three fields:

text The displayed name of the given parameter, it is not used in the backend.

value The class URI which identifies the class itself. This is added to the query.

type The type of the parameter. Yet it can have the value of "sensortype" only. This may contain additional values if new features provide more conditions for filtering.

The SPARQL query for retrieving the objects can be seen on Listing 4.3 The example SPARQL is using two categories for filtering. The result is always the union of the given categories.

Listing 4.3. *SPARQL query for Search*

```
SELECT ?procedure ?name ?offering
    { ?class rdfs:subClassOf+ mit:sensor  .
     ?procedure rdf:type  ?class ; rdfs:label ?name ; demo:offering ?offering
     { ?class rdfs:subClassOf* <category_one> } UNION { ?class rdfs:subClassOf*
         <category_two> }
    }
```

4.8.5 Observe service

The Observe service queries sensor observations and returns them to the frontend. The time range can be given to the query allowing the frontend to narrow down the data to a manageable level. The service uses the SOS component. It is expecting an object similar to the result of the Capabilities service as an input parameter, and also two datetime strings in ISO 8601 format, which are the start and end times of the observations. The service automatically decreases the number of results to a displayable number of 50 results if more has been found. The result will be two array and some metadata: one array for the date (x field) and another array for the measurements (y field). Boolean results are automatically converted to a series of zeros and ones. The text fields are returned as string values in the y array.

4.8.6 Location service

The Location service queries only the positions of the given object. The object should be given in a format similar to the result of the Capabilities service. The service uses the SOS component to query the featureOfInterest's position and return it as an object. Yet, it can only retrieve point like sensors.

4.8.7 State service

This service retrieves the current status of the system. The status is queried by the Status component from time to time. The service is simply returning the result in a response and some additional metadata about showable fields.

4.9 The backend components

There are three main components for the monitoring system. They all provide needed functionalities the the web services describe before to shae some common tasks.

4.9.1 SOS component

This component provides connectivity to the SOS server. Connection parameters are given in the global config file. The component has two main functions:

- **getObservation** queries the SOS database for the result objects in a given timerange for the given sensor.
- **getFeatureOfInterest** gets the location of a given sensor and sends it back as an object.

The component handles connections using the Restler Node module, which has JSON POST request capabilities.

4.9.2 MyDog component

This component gives helper functions for communicating with the Stardog RDF store. It has one important method, the getConnectin function, which returns a connection to the Stardog database. It has one boolean parameter which is set to true if reasoning functionality is needed in Stardog. The queries are put together in the services themselves. There are some additional helper variables, like the list of prefixes for the SPARQL queries and the database name that is used for the semantic store. Stardog database related settings are also stored in the global configuration.

4.9.3 Status component

This component handles the status updates for the sensors. Yet it is only prepared to handle small number of sensors for the special use case. The module starts when the whole system starts. In a given interval specified in the configuration file it will query all the sensors in the system using the MyDog RDF store and the query on Listing 4.4. After that it is going through all the possible devices and reads observations using the SOS component in the past given period. This period can be set in the global configuration too. If there is no data in the given period the sensor is said to be down, otherwise the sensor is marked as up. The current state is maintained in the modules states array. If there has been a change an event can be triggered.

Listing 4.4. *SPARQL query for querying all sensors*

```
SELECT ?procedure ?name ?offering ?foi ?observable ?obs_name ?obs
    { ?class rdfs:subClassOf+ mit:sensor  .
     ?procuri rdf:type  ?class;demo:procedure ?procedure ; rdfs:label ?name ;
         demo:offering ?offering;  demo:observedProperty ?obs ; mit:hasLoc ?loc.
     ?loc demo:foi ?foi. ?obs demo:observedPropertyURI ?observable ; rdfs:label
         ?obs_name ; mit:hasObservedPropertyType ?type .
     ?type rdfs:label ?type_label}
```

4.10 Components of the frontend

The frontend is using the mentioned AngularJS framework to provide responsive, dynamically changing web pages to the user. The initial entry point is the index.html file which is a standard HTML page extended with Angular functionalities. Used modules, stylesheets and javascripts are partly added dynamically during the build process done by Grunt build system. Angular has a built in routing system which can call the needed modules depending on the URL given. Each module has a controller which loads the template of the module (the View). The frontend components can be seen on Figure 4.5.

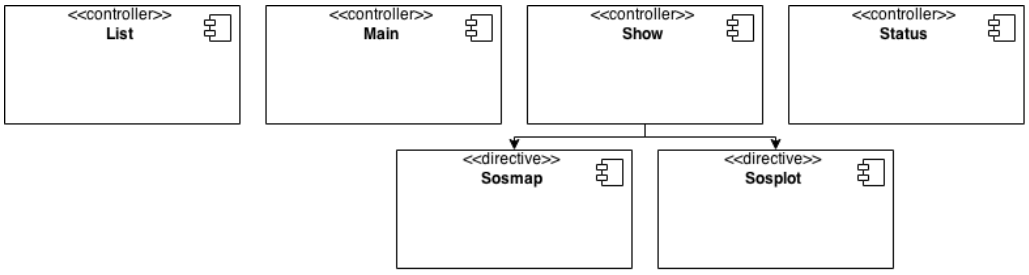

Figure 4.5. *Detailed architecture of the frontend components.*

4.10.1 Main component

The main component renders the opening page with the application help. This page does not have any dynamic property. This is the default route.

4.10.2 List component

This component contains the template to list all the sensors and filter the elements. The template contains a search box directive called ng-taginput which was mentioned before. The filter part uses the backend autocomplete service. The results are loaded using the search backend service. The list is updated every time the filter button is clicked. The sensor names are clickable and they redirect the users to the show component with the additional procedure parameter. This component can be found on /list route.

4.10.3 Status component

This component lists the result of the periodic status checks. It is communicating with the status backend service to get the list and state of the sensors. There are built in Angular directives that make listing and ordering possible. Some parameters were modified to make custom ordering possible. This component is under the route /status.

4.10.4 Show component

This component accepts one parameter, the identifier of the procedure and shows all the related measurements. It queries the necessary parameters using the capabilities backend service. The service returns all the possible measurements and as a second step it invokes the measurement plotter directive called sosplot for all the possible observable properties. It also displays the position of the sensor using Google Maps. The data date and time range can be selected by a third party datetime AngularJS plugin. After selecting the time the charts are automatically updated. This component highly depends on the two directives Sosplot and Sosmap.

4.10.5 Sosplot component

This component is an Angular directive. These directives act as new custom designed HTML elements, which Angular will transform into standard web page elements. The Sosplot directive is used to display measurement data. It needs three parameters, an object which contains the details of the observed property and the start and end time. Using this information it uses the backend observe service to query the measurements and then display the results using the previously mentioned Chart.JS third party chart plotter module. If the data is not plottable, like if we have text property type, the directive automatically loads a different template that will render a list of the text messages.

4.10.6 Sosmap component

The Sosmap component is an Angular directive which is using the Google Maps API to display the position of the sensor. The directive is using the location backend service. Now it is capable of displaying single points on a map.

Chapter 5

User Manual, conclusion and future plans

5.1 User Manual for the Monitoring system

5.2 Conclusion

During the thesis project the author could see the steps of implementing a test environment to a cyberphisical system. Standard description format for sensor data storage of such systems got known. Databases for storing those data was investigated. The one with the most features and plarform independent implementation was chosen. The standard way to describe semantic description for such data and possible database solutions for such systems were also explained. A semantic database with great number of features and Javascript interface has been chosen. After the basics were introduced an actual use case for such a system was shown which was part of the BUTE ETIK project. The implementation of the monitoring system was based on this project. The used technologies were shared and a description of the monitoring system has been given. These technologies were NodeJS with AngularJS frontend using the previously introduced databases.

The implementation showed that NodeJS is capable of serving the purpose of a monitoring system of Cyberphisical systems. Its great number of tools made it easy to add new features of the system. It is fast and it has great support for many interfaces. It can run with small overhead and it can be installed on most platforms. AngularJS is a great tool for Single page application, it is mobile friendly and supports most of the browsers. As web based technologies arise, this solution is a great way to write monitoring application.

5.3 Future plans

Acknowledgement

The author is thankful for the help and support of Dr. György Strausz and Dr. András Förhécz.

Bibliography

[1] 52north org. Sensor observation service.

[2] Mike Botts. Ogc sensorml: Model and xml encoding standard.

[3] MapServer Community. Mapserver webpage.

[4] Erdos Csanád Dr. Dabóczi Tamás, Dr. Tóth Csaba. Szenzorvirtualizáció részletes elemzése.

[5] Förhécz András Engedy István, Eredics Péter. Adatelemzés és tudásmodellezés lehet?ségei.

[6] Förhécz András Sz?ke Ákos Eredics Péter, Györke Péter. Adatintegrációs séma tervezése virtuális szenzorhálózatokhoz.

[7] Apache Foundation. Getting started.

[8] Google. Angularjs.

[9] Google. Chrome v8.

[10] RDF Working Group. Resource description framework (rdf).

[11] istSOS Community. istsos webpage.

[12] Erd?s Csanád Dr. Dabóczi Tamás Wacha Gábor Kovács Kristóf András Szarvas Attila, Dr. Kovácsházy Tamás. Mintaalkalmazás fejlesztése kiberfizikai rendszerekhez ii.