SEVENTH FRAMEWORK PROGRAMME

Specific Targeted Research Project

Call Identifier:	FP7–ICT–2011–7
Project Number:	287305
Project Acronym:	OpenIoT
Project Title:	Open source blueprint for large scale self-organising cloud environments for IoT applications

D3.3.1 Intelligent OpenIoT Edge Server a

Document Id:	OpenIoT-D331-130910-Draft
File Name:	OpenIoT-D331-130910-Draft.doc
Document reference:	Deliverable 3.3.1
Version:	Draft
Editor:	Christian von der Weth
Organisation:	DERI/NUIG
Date:	2013 / 09 /10
Document type:	Deliverable (Prototype)
Dissemination level:	PU (Public)

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V01	Christian von der Weth	DERI	2013/04/28	Initial ToC
V02	Christian von der Weth	DERI	2013/05/20	Initial content
V03	Panos Dimitropoulos	SENSAP	2013/06/20	Low-Level Filtering
V04	Christian von der Weth	DERI	2013/05/20	Additional content (DERI, Sec. 4) Integration of SENSAP contributions
V05	Sofiane Sarni	EPFL	2013/07/08	Contribution to 3.1.1, 3.2.2, 3.3.2
V06	Hylke van der Schaaf	IOSB	2013/08/29	Technical Review
	Sofiane Sarni	EPFL	2013/08/30	GSN Comments addressed
	Christian von der Weth	DERI	2013/08/30	Addressed all TR comments not related to GSN.
V07	Johan E. Bengtsson	AL	2013/08/30	Quality Review
V08	Martin Serrano	DERI	2013/09/05	Circulation for Approval
V09	Martin Serrano	DERI	2013/09/09	Approved
Draft	Martin Serrano	DERI	2013/09/10	EC Submitted

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TERMS AND ACRONYMS

ALE	Application Level Events
AO	Association Ontology
BGP	Basic Graph Pattern
CQELS	Continuous Query Evaluation over Linked Streams
DnS	Descriptions and Solutions
DOLCE	Descriptive Ontology for Linguistic and Cognitive Engineering
DUL	DOLCE+DnS Ultralite
EPC	Electronic Product Code
GSN	Global Sensor Network
ICO	Internet Connected Objects
ITSO	Ontology for IT services delivered via the Cloud
LLRP	Low Level Reader Protocol
LSM	Linked Sensor Middleware
PROV	Provenance Ontology
RDF	Resource Description Framework
SIOC	Semantically-Interlinked Online Communities
SPARQL	SPARQL Protocol and RDF query Language
SSN	Semantic Sensor Network (Ontology)
XML	eXtensible Markup Language
XMPP	eXtensible Messaging and Presence Protocol

1 INTRODUCTION

1.1 Scope

This deliverable introduces and describes the *Intelligent OpenIoT Edge Server*. Generally speaking, the edge server component acts as gateway between the Internet-connected objects, particularly physical sensors, and the Cloud-based infrastructure of OpenIoT. The two major components of the edge server are the Global Sensor Network (GSN), responsible for providing an interface for the raw and heterogeneous sensor data, and the Linked Sensor Middleware (LSM), responsible organising the GSN data to become Linked Data and to annotate the data to enable sophisticated sensor discovery and service orchestration. Both GSN and LSM provide various means, described in this deliverable, for filtering and aggregating the data before being handed over to overlying applications. From a user perspective, the deliverable also describes in detail (a) how the raw input data can be fed into the system and (b) how the annotated data and/or pre-processed results can be accessed and retrieved.

1.2 Audience

The target audience of this deliverable include:

- **Researchers, developers and integrators within the OpenIoT consortium**: This deliverable illustrates the features and functionalities of the Intelligent OpenIoT Edge Server. Most notably, this includes participants that work on (a) proving sensor data to be fed into the OpenIoT infrastructure and (b) accessing the data for service discovery and orchestration.
- **Researchers within other IERC projects:** The deliverable presents a core component of the OpenIoT middleware platform.
- **Researchers working on IoT**: The deliverable provides insights into the challenges and our proposed solutions for integrating data from heterogeneous sources, i.e. Internet-connected objects, into a Cloud-based infrastructure.
- **Open source community**: OpenIoT intends to build an open source community based on the OpenIoT middleware infrastructure. This deliverable serves as a guide to some of the components and functionalities of OpenIoT.

1.3 Summary

To act as an appropriate middleware platform for Internet-connected objects (ICOs), OpenIoT has to fulfil two major requirements. Firstly, the platform must allow for easy integration of data coming from all kinds of Internet-connected objects, particularly physical sensors. Secondly, the raw input data coming from ICOs need to be semantically annotated as well as transformed to comply with the Linked Data principles. Further requirements are the need to offer dynamic sensor/ICO selection and orchestration functionalities. Mechanisms for all these requirements are implemented within the OpenIoT Edge Server.

The two main building blocks of the edge server are the Global Sensor Network (GSN) and the Linked Sensor Middleware (LSM). Sensors and Internet-connected objects need to be registered at GSN. This registration process essentially requires the definition of a wrapper to transform the incoming sensor data into a common format. GSN introduces the notion of a virtual sensor to abstract from the device-specific data representation. A virtual sensor can be a physical sensors or any other kind of data source. Moreover, a virtual sensor can represent a set of data sources. LSM, in turn, considers virtual sensors as input data sources. Data coming from virtual sensors are transformed into a Linked Data representation, i.e., RDF, and annotated according to the supported ontologies. This transformation is, again, done using wrappers that add annotations to the data.

Both GSN and LSM provide mechanisms for filtering and aggregating data. For example, as mentioned above, GSN allows aggregating the data from multiple sensors into one virtual sensor. Also, a virtual sensor might provide a reading only when the reading of the underlying physical sensor exceeds a specific threshold. Working on annotated RDF data, LSM provides even more sophisticated means to filter and aggregate data. It allows not only the execution of arbitrary complex SPARQL queries over stored sensor data, but also the execution of continuous queries – formulated in CQELS as an extension of the SPARQL language – over the live incoming data streams. For example, a CQELS query might return a result as soon as the sensors in three different rooms report significantly unexpected readings. Programmatic access to the data is also handled by LSM. For this, LSM features a SPARQL endpoint as well as various streaming channels (via WebSockets, XMPP, PubSubHubbub) to forward requested data to applications.

The core features of GSN and LSM are application-agnostic. There are, however, various extensions required to make both systems suitable to function within the OpenIoT Edge Server. Firstly, LSM needs to support the full OpenIoT ontology that, in turn, derives from the requirements of the different OpenIoT use cases (see Deliverables 3.1a and 3.1b). Secondly, LSM must feature mechanisms for quick and easy mash-up development to realise the (near) zero-programming paradigm envisioned for OpenIoT. Lastly, both GSN and LSM were originally designed and developed to run in centralised systems. In order to handle large numbers of sensors, huge data volumes and many concurrent services, LSM has to be extended to delegate particularly resource-intensive tasks, such as archiving and accessing the data, to external processing nodes.

1.4 Structure

Following the introduction, Section 2 outlines related work. It outlines the most popular middleware solutions with similar aims as OpenIoT. For each solution, the section outlines in what way OpenIoT differs from or moves beyond existing solutions. Section 3 presents the core application-agnostic features and functionalities of the OpenIoT Edge Server. The first part of this section (Section 3.1) covers data importing, i.e., how data from heterogeneous sources such as Internet-connected objects are fed into the system. Sections 3.2 and 3.3 cover the filtering capabilities and aggregation capabilities of the edge server. Both sections address these capabilities according to three basic layers: sensor layer, GSN layer, and LSM layer. Section 3.4 finally covers the different means to gain access to the collected and annotated data. Section 4 presents the OpenIoT-specific aspects of the edge server. This includes the data modelling based on the OpenIoT ontology (Section 4.1), the support for rapid mash-up development (Section 4.2) and the extended functionalities to run within a Cloud setting (Section 4.3). Section 0concludes this deliverable.

2 EXISTING IOT MIDDLEWARE PLATFORMS

The demand for easily making the data of sensors public spurred the design and development of IoT middleware platforms. This includes both commercial products and open-source solutions. In this section, we outline some of the most popular platforms, highlight their aims and strengths, and put them in context of the aims of OpenIoT, i.e., we discuss where OpenIoT will move beyond current solutions.

2.1 Everyware

The Eurotech (multinational) company (headquartered) in Italy offers the Everyware Cloud solution[1], which enables integration of wired and wireless devices within a cloud computing solution. The integration empowers data management and storage of data feeds, along with translation between different protocols. Furthermore, thanks to its accompanying ESF software framework, it allows for flexible programming of interactions between different devices/data feeds towards supporting M2M (machine-to-machine) communications and interactions. The ESF allows for the easy integration of the data streams in business applications. **Figure 1** shows the overall architecture of Everyware.

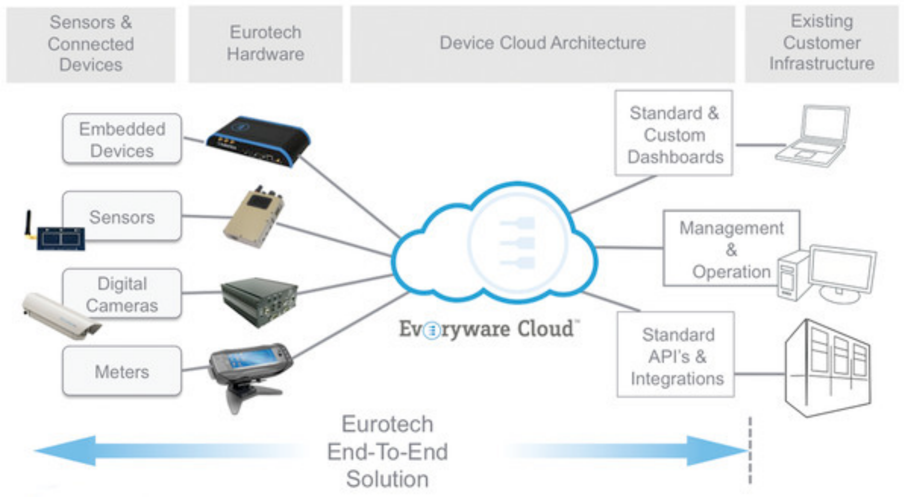

Sensors & Connected Devices
Eurotech Hardware
Device Cloud Architecture
Existing Customer Infrastructure
Embedded Devices
Sensors
Digital Cameras
Meters
Everyware Cloud
Standard & Custom Dashboards
Management & Operation
Standard API's & Integrations
Eurotech End-To-End Solution

Figure 1: Everyware Architecture

OpenIoT supports most of these features based on its background project GSN, which is however based on a research project and is not a commercial product. In the scope of OpenIoT, GSN will be integrated with a cloud infrastructure. Furthermore, OpenIoT will be enhanced with functionalities for on-demand service creation and utility calculation, which are not offered by ESF. However, OpenIoT will initially be a research prototype with limited aims for scalability, performance and availability. In contrast to OpenIoT, Everyware does not support on-demand services and partly relies on non-open standards. Everyware focuses on M2M (machine-to-machine) communication, while OpenIoT is targeting ICO orchestration and scheduling.

[1] http://www.eurotech.com/en/products/software+services/everyware+device+cloud

2.2 UBIDOTS

Ubidots[2] is one of the several platforms that enable service developers to integrate their data streams into a cloud computing environment. It provides tools for managing the data streams and building applications. In this sense the Ubidots programming paradigm can be considered as a Platform-as-a-Service (PaaS) paradigm. The main benefits of the Ubidots platform are:

- Ability to post device data on the basis of a RESTful API.
- Flexibility, which is empowered by a multi-tenant platform allowing its users to mix different data streams and present them in mashup-like dashboards.
- Ubiquitous availability, which is empowered by a cloud-based paradigm
- Security based on in-built of security mechanisms in the Ubidots cloud.

OpenIoT will in principle support all the features of the Ubidots platform. However, the OpenIoT platform will not be primarily offered as an online cloud system according to the PaaS paradigm, though this is in principle possible. OpenIoT will be primarily available as a middleware blueprint for service providers to build and offer on-line IoT/cloud solutions. This is the model already followed by the GSN middleware. From a functional perspective OpenIoT will offer dynamic sensor/ICO selection and orchestration functionalities, which are not supported by Ubidots and other similar platforms.

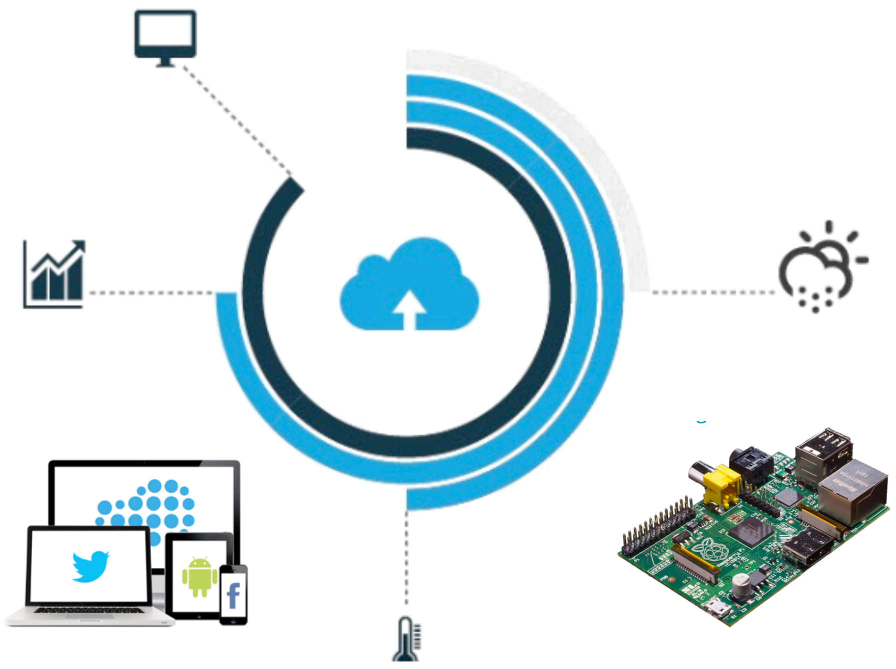


Figure 2: Ubidots Architecture

[2] http://www.ubidots.com/

2.3 Xively

Xively[3] (formerly known as Cosm, formerly known as Pachube) bears many similarities to Ubidots. It allows service developers and integrators to programmatically integrate multiple data feeds into applications. At the same time, it provides tools and techniques for the visualisation of data feeds. Note that Xively leverages data feeds contributed from users all over the world to its public cloud. Xively users contribute data feeds from their programs and online devices. Based on these data, service developers and integrators can connect feeds, devices and applications online.

OpenIoT provides much more sophisticated functionalities in terms of sensor stream integration and orchestration, as well as increased dynamism compared to Xively. OpenIoT comes with a wide range of semantic features, which are based on the rich annotation/representation of the sensors and ICOs. On the other hand, Xively sensors and data feeds are described on the basis of few simple (user-defined) name/value pairs, and this limits the possibilities for intelligent discovery and filtering of resources. However, OpenIoT is not intended to become an online publicly accessible cloud and has a long way to go in order to reach the user base of Xively, as well as the critical mass of related applications. Xively is promoted as an online system for service developers/integrators (public cloud, PaaS model), while OpenIoT is destined to be (primarily) a platform for building/customising sensor clouds. Furthermore, Xively demands that users explicitly develop source code for their applications, while OpenIoT will provide zero programming functionality (e.g., sensor queries without programming).

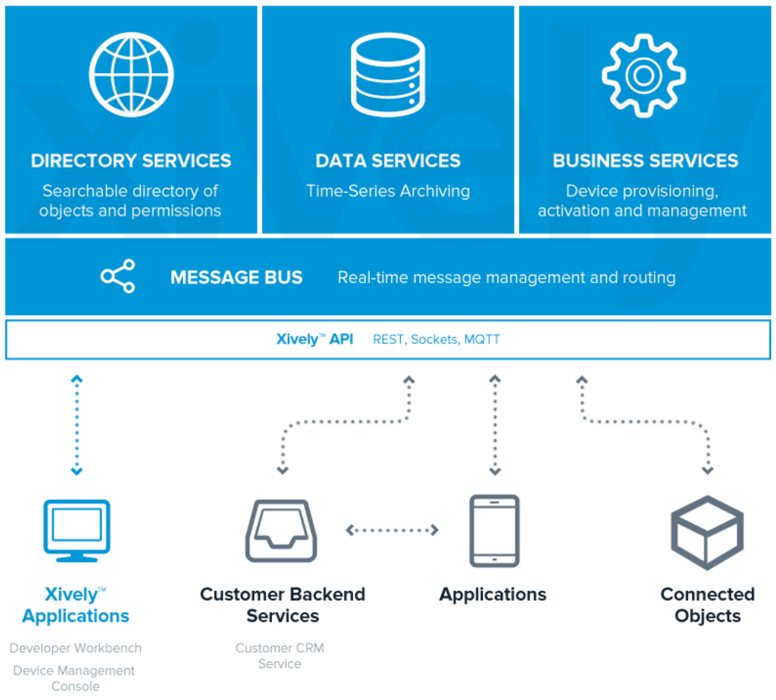


Figure 3: Xively Architecture

[3] https://xively.com/

2.4 ThingSpeak

ThingSpeak[4] is another online cloud platform enabling sensor streams integration and related application development. It is an open platform enabling and easing the connectivity between people and devices. Live data is fed into ThingSpeak via simple HTTP requests containing key-value pairs. The platform also provides data importing mechanisms to import already collected/archived data. The data can be processed (e.g. scaling, summing, averaging) and filtered for their eventual use by an application or service. ThingSpeak also provide various means to visualise the data, and it is fully localised, supporting over 40 locales.

On the whole, ThingSpeak bears a lot of similarities with Ubidots and Xively, yet it also puts emphasis on social networks (e.g., Twitter) as a means to connect people and things. In principle, the platform has similar benefits (like Xively and Ubidots) comparing to OpenIoT, but also similar downsides. Most prominently, ThingSpeak supports simple semantics but does not support dynamic sensor selection and orchestration.

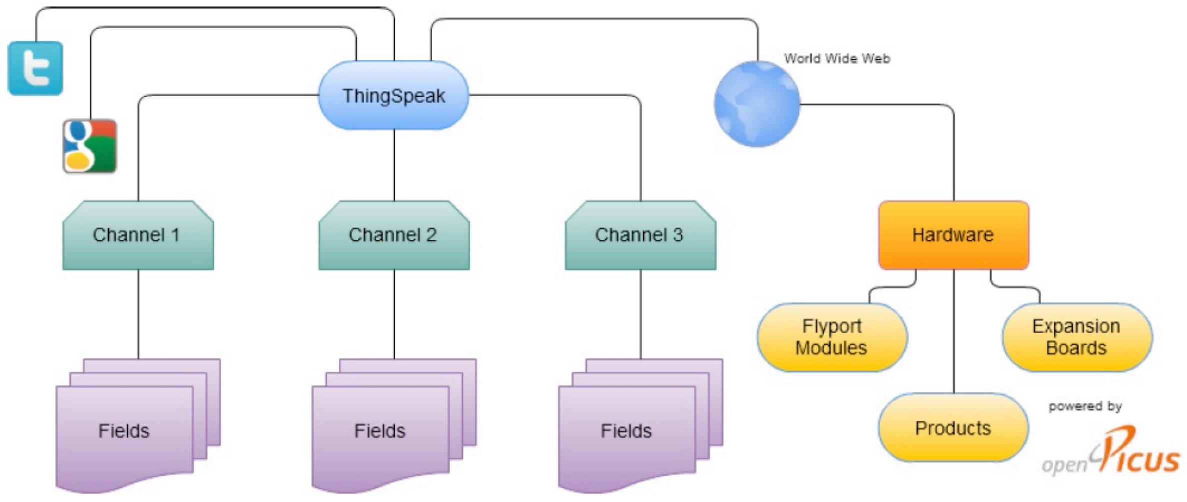


Figure 4: ThingSpeak Architecture

2.5 Open.Sen.se

At the time of writing, Open.Sen.se[5] is a Beta service under development. The following comments may therefore lose their significance in the future, so build your own opinion by trying it yourself and check if it has evolved since the time this article was written. Like ThingSpeak, Open.Sen.se is an online data logging service that can also trigger conditional actions. This is done using Apps, which can be quite sophisticated. It provides various features that are typically beyond the mission of a remote data service. This platform offers different ways to plot readings, process data

[4] https://www.thingspeak.com/

[5] http://open.sen.se/

and even control values. Open.Sen.se also lets users configure their own monitoring/control panels and link to other internet services as Twitter.

Open.Sen.se is based on the notion of channels that send and receive data. Channel endpoints can be physical devices connected to the Internet, web forms through which users enter data manually, or open external data sources. Applications use the data they receive from such channels to perform different types of tasks, such as calculations, aggregations, merging, or comparing the data. Both raw data from channels and processed data from applications can trigger pre-defined actions. Open.Sen.se also provides mechanisms to visualize data and its history in real time.

OpenIoT will provide the same functionality as Open.Sense.se and expand on them, e.g., by allowing users to formulate more complex filtering and aggregation mechanisms over the data. Being on a similar level as, for example, ThingSpeak or Xively, Open.Sen.se also does not focus on the semantic annotation of the sensor data, which on the other hand is a cornerstone within the OpenIoT architecture.

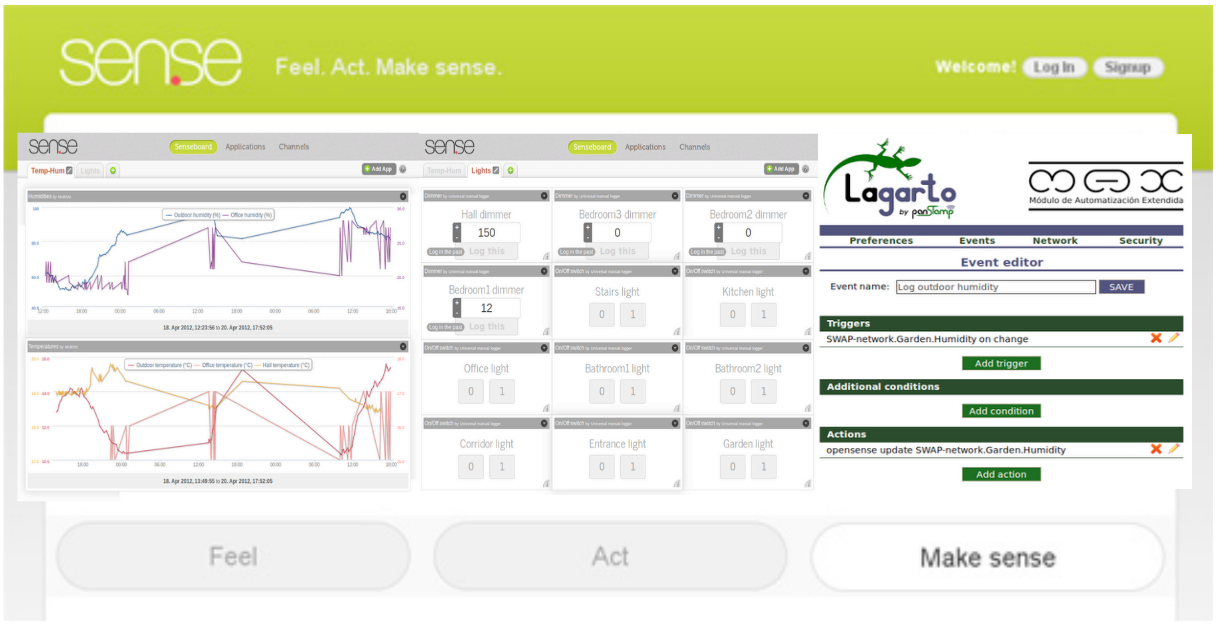


Figure 5: Open.Sen.Se Architecture

2.6 Nimbits

Nimbits a collection of software components designed to log time series data, such as a changing temperature reading from a sensor. As that data is logged, various events can be triggered, such as a calculation or alert. The Nimbits ecosystem consists of users around the world who have downloaded and installed an instance of Nimbits Server on their cloud of choice such as Google App Engine[6], Amazon EC2[7] or their own internal hardware. There is also a central instance of Nimbits[8]

[6] https://cloud.google.com/products/

[7] http://aws.amazon.com/ec2/

which is open to the public and to users of Google Apps for Domains. Users or developers can also download and install a Nimbits instance on their own infrastructure.

Running Nimbits instances around the world, all providing highly redundant and scalable data logging, are able to share data with each other, and can be made searchable so people can find data feeds and connect to them. The Nimbits Cloud platform is an installable server much like any other server like SQL or Oracle. It is a Process Data historian which means it's optimised to store data that comes in as time-stamped streams of values. It makes it easy and efficient to store high frequency changes. Nimbits was built from the ground up to be highly scalable, which means that when one server instance becomes overloaded, another instance can spin up and take on some of the load.

With Nimbits being a collection of software components, the installation and setup phase is typically more complex compared to other service-based middleware platforms, including OpenIoT. Similar to previously mentioned alternatives, Nimbits lacks any support for semantics. It essentially provides an access to individual sensor data, including visualization, but does not allow for elaborate sensor data discovery or for the orchestration of more complex services.

Summing up, existing and open IoT platforms particularly aim to simplify making data from all kinds of sources (physical devices, human input, online data, etc.) publicly available, and hiding the heterogeneity of data behind a common API. All platforms typically provide basic functionalities for filtering and aggregating the data, as well as the formulation of events based on current input data. In this respect, OpenIoT will eventually cover most of these functionalities and will also go beyond that. Most importantly, OpenIoT puts much emphasis on the semantics of the collected, processed and available data (streams). This, in turn, allows for the support of (a) more sophisticated data filtering and aggregation techniques and (b) dynamic sensor/ICO selection and orchestration functionalities. Despite the increased flexibility, OpenIoT aims to provide (near) zero programming functionality to users, be it application developers, service providers or platform providers.

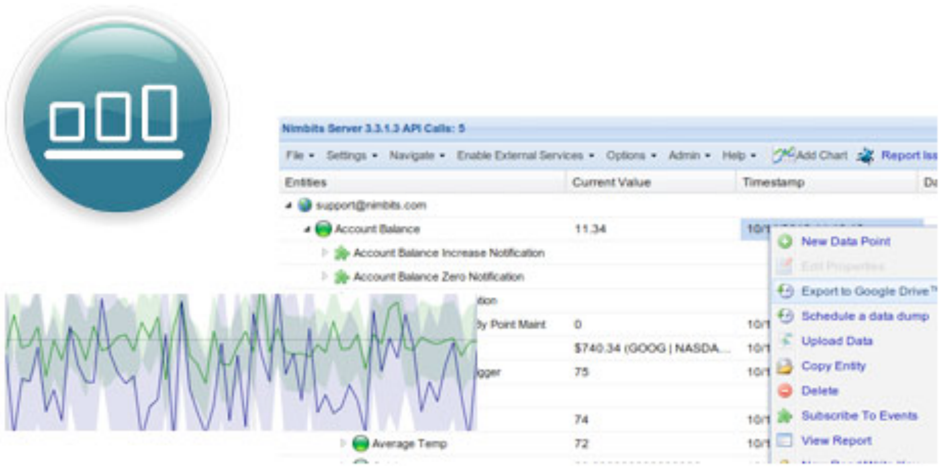


Figure 6: Nimbits Architecture

8 https://cloud.nimbits.com

3 EDGE SERVER – CORE COMPONENTS

3.1 Data Import

GSN is a software middleware designed to facilitate the deployment and programming of sensor networks. GSN runs on one or more computers composing the backbone of the acquisition network (see Figure 2).

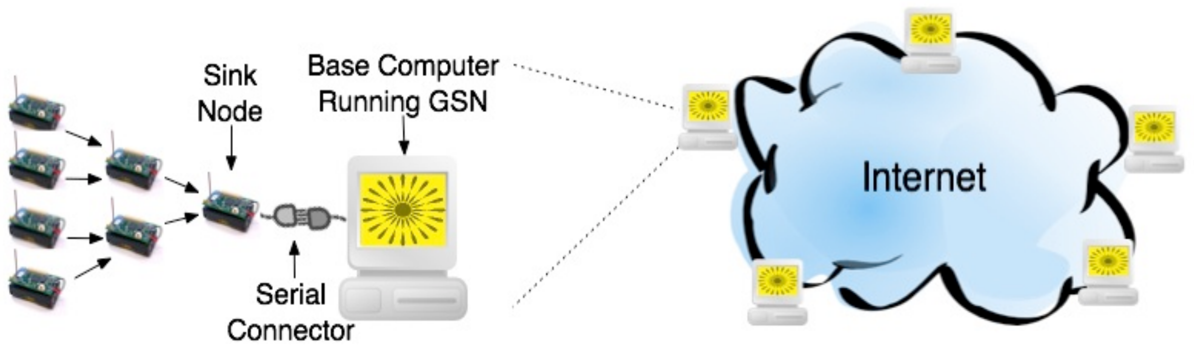


Figure 7: Data acquisition network in GSN

3.1.1 Raw Data into GSN

A set of wrappers allows feeding raw data into the system. Wrappers are used to encapsulate the data received from the data source into the standard GSN data model, called a *Stream Element*. A Stream Element is an object representing a row in the data store of GSN. Each wrapper is implemented as a Java class. Usually, a wrapper initializes a specialized third-party library in its constructor. It also provides a method which is called each time the library receives data from the monitored device. This method will extract the interesting data, optionally parse it, and create one or more Stream Element(s) with one or more columns. From this point on, the received data has been mapped to a SQL data structure with fields that have a name and a type. GSN is then able to filter this using its SQL-like syntax.

Data streams are processed according to XML specification files. The system is built upon a concept of sensors (real sensors or virtual sensors, which are data sources created from live data) that are connected together in order to build the required processing path. For example, one can imagine a thermometer that would send its data into GSN through a wrapper, then that data stream could be sent to an averaging node, the output of this node could then be split and sent to a database for recording and to a web site for displaying the average measured wind in real time. The described example can be realised by editing only a few XML files in order to connect the various nodes together.

Various wrappers are already available and new ones can be written quickly. **Table 1** lists the standard wrappers that ship with GSN.

Table 1: Standard GSN wrappers

Wrapper Name	**Wrapper Id**	**Remarks**
LocalWrapper	local	Local wrapper for combining input from other virtual sensors
RemoteWrapper	remote-rest	Restful Streaming
RemoteWrapperPush	remote	Push-based Streaming
TinyOS	tinyos-mig	TinyOS wrapper
CSV-Wrapper	csv	Reads from local CSV data files
Memory-Usage	memory-usage	Monitors memory usage in the JVM
system-time	system-time	Reads system time
Direct Remote Push	remote-direct	Stateless Push for sporadically connected sensors
HTTP-Get	http-get	HTTP-based wrapper to get images from webcams, can be personalized easily
Serial	serial	Reads packets from serial port
UDP	udp	Reads packets from UDP port
USB-Cam	usb-cam	Reads images from USB camera
RSS-Feed	rss	Gets data from RSS feeds
MultiFormat	multiformat	Custom protocol for analysis
GPSTest	gps-test	
Replay	replay	Replays already saved in an SQL table
Grid data wrapper	grid	Grid data wrapper
Image file wrapper	imagefile	Image file wrapper

3.1.1.1 *Writing new GSN wrappers*

All standard wrappers are subclasses of *gsn.wrapper.AbstractWrapper*. Subclasses must implement the following four (4) methods:

1. boolean initialize()
2. void finalize()
3. String getWrapperName()
4. DataField[] getOutputFormat()

The initialize() function is called after the wrapper object is created. All initializations pertaining to the wrapper should be made there. On the other hand, the finalize() method is called for releasing any resources handled by the wrapper. It is called when the wrapper is about to be deleted. Each wrapper is a thread in the GSN. If you want to do some kind of processing in a fixed time interval, you can override the **run()** method. The run method is useful for time driven wrappers in which a timer triggers the production of a sensor data. Optionally, you may wish to override the method **sendToWrapper**.

3.1.1.1.1 initialize() method

This method is called after the wrapper object creation which happens the first time a virtual sensor referencing it is created. Wrappers can be shared among different virtual sensors if they have exactly the same parameters and producing the same streams.

The complete method prototype is as follows:

```
public boolean initialize();
```

In this method, the wrapper should try to initialize its connection to the actual data producing/receiving device(s) (e.g., wireless sensor networks or cameras). The wrapper should return true if it can successfully initialise the connection, false otherwise. If the wrapper couldn't be initialized, the virtual sensor won't be initialized.

GSN provides access to the wrapper parameters through the following method call:

```
getActiveAddressBean().getPredicateValue("parameter-name");
```

For example, if you have the following fragment in the virtual sensor configuration file:

```
<source ... >
  <address wrapper="x">
    <predicate key="range">100</predicate>
    <predicate key="log">0</predicate>
  </address>
```

You can access the initialization parameter named range with the following code:

```
if(getActiveAddressBean().getPredicateValue("range") != null)
{...}
```

By default GSN assumes that the timestamps of the data produced in a wrapper are local, that is, the wrapper produced them using the system (or GSN) time. If you have cases where this assumption is not valid and GSN should assume remote timestamps for stream elements, add the following line in the ***initialize()*** method:

```
setUsingRemoteTimestamp(true);
```

3.1.1.1.2 finalize() method

The finalize method is called just before the wrapper object is deleted. It has no parameters.

```
public void finalize ( )
```

In the ***finalize()*** method, you should release all the resources you acquired during the initialization procedure or during the life cycle of the wrapper. Note that this is the last chance for the wrapper to release all its reserved resources and after this call the wrapper instance virtually won't exist anymore. For example, if you open a file in the initialization phase, you should close it in the finalization phase.

3.1.1.1.3 getWrapperName() method

This method returns a name for the wrapper. The name identifies the wrapper in a meaningful way in the system. Usually, we use the wrapper class name appended with a static counter like in the example below:

```
static int counter;
...
public String getWrapperName() {
return this.getClass().getName + counter;
}
```

3.1.1.1.4 getOutputFormat() method

The method ***getOutputFormat()*** returns a description of the data structure produced by this wrapper. This description is an array of DataField objects. A DataField object can be created with a call to the constructor

```
public DataField(String name, String type, String description)
```

The name is the field name. The type is one of the basic GSN data types[9], which corresponds to the same types in MySQL. If another database is used with GSN, those types are mapped to the relevant data types.

- *TINYINT*
- *SMALLINT*
- *INTEGER*
- *BIGINT*
- *CHAR*
- *BINARY*
- *VARCHAR*
- *DOUBLE*
- *TIME*

Description is a text describing the field. The following examples should help you get started:

Wireless Sensor Network Example

Assuming that you have a wrapper for a wireless sensor network which produces the average temperature and light value of the nodes in the network, you can implement ***getOutputFormat()*** as follows:

```
public DataField[] getOutputFormat() {
        DataField[] outputFormat = new DataField[2];
        outputFormat[0] = new DataField("Temperature", "double",
                "Average of temperature readings from the sensor network");
        outputFormat[1] = new DataField("light", "double",
                "Average of light readings from the sensor network");
        return outputFormat;
}
```

[9] See ***gsn.beans.DataTypes*** package

Webcam Example

If you have a wrapper producing jpeg images as output (e.g., from wireless camera), the method is similar to, as illustrated below:

```
public DataField[] getOutputFormat() {
        DataField[] outputFormat = new DataField[1];
        outputFormat[0] = new DataField("Picture", "binary:jpeg",
                "Picture from the Camera at room BC143");
        // jpeg is added to the type ino order to properly display
        // the object in the web interface when requested
        return outputFormat;
}
```

3.1.1.1.5 run() method

As described before, the wrapper acts as a bridge between the actual hardware device(s) or other kinds of stream data sources and GSN, thus in order for the wrapper to produce data, it should keep track of the newly produced data items. This method is responsible for forwarding the newly received data to the GSN engine. You should not try to start the thread by yourself: GSN takes care of this. All the data acquisition logic should be implemented within this method (or called from there). As long as data is needed, the wrapper stays active with the *isActive* flag. Once the run() method finishes its execution the thread is ended and no more data acquisition is possible with that wrapper.

The method should be implemented as illustrated below :

```
while(isActive()) {
    {

      // The thread should wait here until arrival of a new data notifies it

      // or the thread should sleep for some finite time before polling the data
source or producing the next data

    }


    //Application dependent processing ...

    StreamElement streamElement = new StreamElement ( ...);

    postStreamElement( streamElement ); // This method in the AbstractWrapper sends
the data to the registered StreamSources

}
```

Webcam example

Assume that we have a wireless camera which runs a HTTP server and provides pictures whenever it receives a **GET** request. In this case we are in a data on demand scenario (most of the network cameras are like this). To get the data at the rate of 1 picture every 5 seconds we can do the following:

```
while(isActive()) {
        byte[] received_image = getPictureFromCamera();
        postStreamElement(System.currentTimeMillis(), new Serializable[]
{received_image});

        Thread.sleep(5*1000); // Sleeping 5 seconds

}
```

Data driven systems

Compared to the previous example, we do sometimes deal with devices that are data driven. This means that we don't have control of either when the data is produced by them (e.g., when they do the capturing) or the rate at which data is received from them. For example, having an alarm system, we don't know when we are going to receive a packet, or how frequently the alarm system will send data packets to GSN. This kind of system is typically implemented using a callback interface. In the callback interface, one needs to set a flag indicating the data reception state of the wrapper and control that flag in the run method to process the received data.

3.1.1.1.6 sendToWrapper()

In GSN, the wrappers can not only receive data from a source, but also send data to it. Thus wrappers are actually two-way bridges between GSN and the data source. In the wrapper interface, the method ***sendToWrapper()*** is called whenever there is a data item which should be send to the source. A data item could be as simple as a command for turning on a sensor inside the sensor network, or it could be as complex as a complete routing table which should be used for routing the packets in the sensor network. The full syntax of ***sendToWrapper()*** is as follows:

```
public  boolean  sendToWrapper(String  action,  String[]  paramNames,  Object[]
paramValues) throws OperationNotSupportedException;
```

The default implementation of the afore-mentioned method throws an ***OperationNotSupportedException*** exception because the wrapper doesn't support this operation. This design choice is justified by the observation that not all kind of devices (sensors) can accept data from a computer. For instance, a typical wireless camera doesn't accept commands from the wrapper. If the sensing device supports this operation, one needs to override this method so that instead of the default action (throwing the exception), the wrapper sends the data to the sensor. You can consult the gsn.wrappers.general.SerialWrapper class for an example.

A fully functional wrapper

```
/*
 * This wrapper presents a MultiFormat protocol in which the data comes from the
 * system clock. Think about a sensor network which produces packets with
 * several different formats. In this example we have 3 different packets
 * produced by three different types of sensors. Here are the packet structures
 * : [temperature:double] , [light:double] , [temperature:double, light:double]
 * The first packet is for sensors which can only measure temperature while the
 * latter is for the sensors equipped with both temperature and light sensors.
 *
 */

public class MultiFormatWrapper extends AbstractWrapper {

  private DataField[] collection = new DataField[] { new DataField("packet_type",
"int", "packet type"),

      new DataField("temperature", "double", "Presents the temperature sensor."),
new DataField("light", "double", "Presents the light sensor.") };

  private final transient Logger logger =
Logger.getLogger(MultiFormatWrapper.class);

  private static int counter;

  private AddressBean params;

  private long rate = 1000;


  public boolean initialize() {

    setName("MultiFormatWrapper" + counter++);


    params = getActiveAddressBean();

    /* setName is the java.lang.Thread function. Threads have to be uniquely named.
       getWrapperName() is internal to GSN. */


    if ( params.getPredicateValue( "rate" ) != null ) {

      rate = (long) Integer.parseInt( params.getPredicateValue( "rate"));


      logger.info("Sampling rate set to " + params.getPredicateValue( "rate") + "
msec.");

    }


    return true;

  }
  public void run() {
    Double light = 0.0, temperature = 0.0;
    int packetType = 0;

    while (isActive()) {
      try {
        // delay
        Thread.sleep(rate);
      } catch (InterruptedException e) {
        logger.error(e.getMessage(), e);
      }

      // create some random readings
      light = ((int) (Math.random() * 10000)) / 10.0;
      temperature = ((int) (Math.random() * 1000)) / 10.0;
      packetType = 2;

      // post the data to GSN
      postStreamElement(new Serializable[] { packetType, temperature, light });
    }
  }

  public DataField[] getOutputFormat() {
    return collection;
  }

  public String getWrapperName() {
    return "MultiFormat Sample Wrapper";
  }

  public void finalize() {
    counter--;
  }
}
```

3.1.2 Data import into LSM

There are two principle ways to import/stream data into LSM – particularly data as output from GSN in the context of OpenIoT: pull-based and push-based. In the pull-based approach, LSM periodically polls a data source similar to data feed. In contrast, the push-based approach enables data sources (e.g., GSN) to actively send data to LSM. Both mechanisms are used within the OpenIoT infrastructure, featuring different advantages and disadvantages depending on the actual use case; we discuss these characteristics in more detail in the following subsection

Similar to the data import from sensor readings into GSN, both the pull-based and push-based methods require the formulation of wrappers. Here, wrappers specify how the collected sensor readings (in case of pulling) or the received sensor readings (in case of pushing) are transformed according the Linked Stream Data layout of LSM. LSM provides three different types of wrapper: (1) *Physical Wrappers* are designed for collecting sensor data from physical devices. (2) *Linked Data* Wrappers expose relational database sensor data into RDF. (3) *Mediate Wrappers* mediate the connections to other sensor middleware platforms by transforming data from a variety of data formats and data feeding protocols into RDF. Among others, LSM provides mediating wrappers for middlewares such as GSN, Xively (formerly Cosm, formerly Pachube), the sensor gateway/Web services from National Oceanic and Atmospheric Administration (NOAA[10]), and the London Transport syndication[11].

Each wrapper is pluggable at runtime so that wrappers can be developed to connect new types of sensors into a live system or when the system is running. The wrappers output the data in a unified format, following the data layout described in the ontologies supported by LSM, which in the context of the project is the OpenIoT ontology in particular. Within the overall OpenIoT architecture, the main source of sensor readings for a data import into LSM is GSN.

3.1.2.1 Pull-based Data Import

In the OpenIoT architecture, GSN serves as sensor middleware to publish sensor data form all kinds of physical devices via a common Web service interface. LSM sits on top of it, fetching the data from GSN via HTTP, transforming it into Linked Data, enriching it with semantic information, and storing the data into an RDF storage system. Adding sensors (here: adding sources of sensor data provided by GSN) and acquiring the data is done by LSM via wrappers. In case of data stemming from GSN, wrappers apply data transformation rules to map the data in a given format into RDF. For example, sensor data in XML can be transformed to RDF using XSLT transformations[12] and the meanings of the sensor readings contained in the XML tags are annotated with concepts in the ontology via an XSLT transformation rule. To give an example for publicly available sensor data in XML, **Figure 8** shows the extract of the output of a weather sensor feed[13].

[10] http://www.noaa.gov

[11] http://www.tfl.gov.uk/businessandpartners/syndication/default.aspx

[12] http://www.w3.org/TR/xslt.

[13] http://api.wunderground.com/weatherstation/WXCurrentObXML.asp?ID=I90580546

```xml
<?xml version="1.0" encoding="utf-8" ?>
  <current_observation>
    <credit>Weather Underground Personal Weather Station</credit>
    <credit_URL>http://wunderground.com/weatherstation/</credit_URL>
    ...
    <location>
      <full>Albany, Auckland, NI</full>
      ...
      <latitude>-36.743698</latitude>
      <longitude>174.654633</longitude>
      <elevation>10 ft</elevation>
    </location>
    <station_id>I90580546</station_id>
    <station_type>WS-3600</station_type>
    <observation_time>Last Updated on May 9, 9:28 AM NZST</observation_time>
    <observation_time_rfc822>Wed, 08 May 2013 21:28:42 GMT</observation_time_rfc822>
    <weather></weather>
    <temperature_string>56.5 F (13.6 C)</temperature_string>
    <temp_f>56.5</temp_f>
    <temp_c>13.6</temp_c>
    <relative_humidity>99</relative_humidity>
    <wind_string>Calm</wind_string>
    <wind_dir>SW</wind_dir>
    <wind_degrees>225</wind_degrees>
    ...
  </current_observation>
```

Figure 8: Example of weather sensor feed.

Writing a new wrapper is a straightforward process and described in the following. Since LSM is implemented in Java, wrappers are also written in this language. All standard wrappers are subclasses of `lsm.wrapper.AbstractWrapper`. Subclasses must implement the following two methods: `boolean initialize()`, and `void run()`. Each instance of a wrapper is a thread in the LSM. If a wrapper developer wants to do some kind of processing in a fixed time interval, the developer can override the `run()` method. The run method is useful for time driven wrappers in which the production of a sensor data is triggered by a timer.

boolean initialize()

This method is called after the wrapper object creation. In this method, the wrapper should try to initialize its connection to the actual data producing/receiving device(s) (e.g., weather sensor or traffic camera). The wrapper should return true if it can successfully initialize the connection, false otherwise.

LSM provides access to the wrapper parameters through the following methods `getPro()` and `setPro(Properties pro)`. The properties variable saves all the configuration parameters which are loaded from wrapper XML configuration file. The XML configuration file has the basic following fragment:

```
<properties>
    <comment>London traffic camera configuration</comment>
    <entry key="wrapper.classname">lsm.wrapper.LondonTrafficCamWrapper</entry>
    <entry key="wrapper.sleeptimevalue">5</entry>
    <entry key="wrapper.sleeptimeunit">minute</entry>
    <entry key="wrapper.type">http</entry>
</properties>
```

Most basically, this configuration file specifies how often the data is being polled from the data source (5 minutes in previous example) and the type of the wrapper, i.e., how the data is fetched (e.g., via HTTP). To access initialization parameter, for example the parameter named class name, is done via `prop.getProperty("wrapper.classname")`. Because the wrapper is also a thread, so that one wrapper is initialized, the thread sleep time unit and the time duration are loaded. For example, in Yahoo Weather wrapper:

```
@Override
public boolean initialize() {
    // TODO : init url here
    url="http://weather.yahooapis.com/forecastrss?w=2344917";
    sleepUnit = prop.getProperty("wrapper.sleeptimeunit");
    sleepDuration = Integer.parseInt(prop.getProperty("wrapper.sleeptimevalue"));
    return true;
}
```

void run()

As described before, the wrapper acts as a bridge between the actual hardware device(s) or other kinds of stream data sources and LSM, thus in order for the wrapper to produce data, it should keep track of the newly produced data items. This method is responsible for forwarding the newly received data to the LSM engine. The method should be implemented as below for the Weather Underground wrapper example:

```
@Override
public void run() {
    System.out.println("-----Yahoo weather Update has started ------");
    for(;;){
        // This method in the AbstractWrapper sends the triples data
        // to the registered message bus
        postRDF(feed(url));
        // the thread should sleep for some finite time before
        // polling the data source or producing the next data
        if(sleepUnit.equals("day"))
            ThreadUtil.sleepForDays(sleepDuration);
        else if(sleepUnit.equals("minute"))
            ThreadUtil.sleepForMinutes(sleepDuration);
        else if(sleepUnit.equals("second"))
            ThreadUtil.sleepForSeconds(sleepDuration);
        else if(sleepUnit.equals("hour"))
            ThreadUtil.sleepForHours(sleepDuration);
    }
}
```

The transformation of the sensor data into N3 triples is done in the `String feed(String url) function`. In the following example, this function fetches input data that are published via a Web service interface and accessible by a unique URL – like it is done within GSN. The incoming data are in XML, thus the function uses XSLT transformations to generate RDF data from the read XML data.

```
private String feed(String url){
  //TODO : load data from URL and convert to Ntriple string here
  String nt = "";
  try{
    String xml = WebServiceURLRetriever.RetrieveFromURL(url);
    Sensor sensor = sensorManager.getSpecifiedSensorWithSource(url);
    if(sensor==null) return null;
    Observation newest = sensorManager.getNewestObservationForOneSensor(sensor.getId());
    if(newest == null || DateUtil.isBefore(newest.getTimes(),
                                 WUnderGroundXMLParser.readerTime)){
      System.setProperty("javax.xml.transform.TransformerFactory",
                                 "net.sf.saxon.TransformerFactoryImpl");
      TransformerFactory tFactory = TransformerFactory.newInstance();
      String xsltPath = XSLTMapFile.sensordata2xslt.get(
                                 SourceType.getSourceType(sensor.getSourceType()));
      try {
        Transformer transformer =
                        tFactory.newTransformer(new StreamSource(new File(xsltPath)));
        String id = sensor.getId().substring(sensor.getId().lastIndexOf("/")+1);
        String foi = "http://lsm.deri.ie/resource/" +
            Double.toString(sensor.getPlace().getLat()).replace(".","").replace("-", "") +
            Double.toString(sensor.getPlace().getLng()).replace(".","").replace("-", "");
        transformer.setParameter("sensorId", id);
        transformer.setParameter("sourceType", sensor.getSourceType());
        transformer.setParameter("sensorType", sensor.getSensorType());
        transformer.setParameter("sourceURL", sensor.getSource());
        transformer.setParameter("foi",foi );
        InputStream inputStream = new ByteArrayInputStream(xml.getBytes("UTF-8"));
        Writer outWriter = new StringWriter();
        StreamResult result = new StreamResult( outWriter );
        transformer.transform(new StreamSource(inputStream),result);
        nt = outWriter.toString().trim();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }catch(Exception e){
    e.printStackTrace();
  }
  return nt;
}
```

From a performance and scalability perspective, a pull-based data import via polling allows for a better control of the overall load of LSM. Although the polling interval for a data source is specified in the wrapper configuration file, it is easy for the server to increase the polling time to avoid any overload in case of large number of data sources initially added with a low polling time.

Polling is a suitable approach if the expected update times of sensor readings are known a priori. For example, for the data feed to a dynamic weather service, sensor readings are needed every few minutes. As such, it is straightforward to set the time for the polling interval within the wrapper configuration file. In case of unknown update intervals, setting the polling interval to a meaningful value is not trivial. While a very frequent polling ensures that no updates are missed it might lead to unnecessary high load in case of unnecessary polling requests due to lack of updates. Vice versa, less frequent polling saves resources but potentially leads to missing updates in the sensor readings. Whether the latter can be tolerated or not typically depends on the requirements of the overlying application or service. Polling is particularly unsuitable for application scenarios including immediate alarms or notifications. Consider, for example, a temperature sensor that only creates an update if the measured value exceeds a specific threshold. To guarantee a close to instant notification using polling, the polling interval needs to be very short. However, given that alarms or notification typically represent exceptions, such a setting would result a huge number of mostly unnecessary polling requests.

3.1.2.2 Push-based Data Import

Besides registering sensors and pulling their data via wrappers, LSM also provides an API for adding and deleting sensors as well as updating, retrieving and deleting GSN sensor data from the LSM triple store. The API is programmatically accessible via the lsmlibs[14] Java library. Further libraries required are dom4j[15] and jena[16]. In the following, this section provides examples for the basic API calls.

[14] http://deri-lsm.googlecode.com/files/lsmlibs.jar

[15] http://sourceforge.net/projects/dom4j/

Adding a new sensor: If a new virtual sensor is defined within GSN, either as a wrapper for a physical device or as a composition of already existing virtual sensors, this new sensor needs to be registered at the LSM store. For this, the class `LSMTripleStore` provides a function `sensorAdd()`. This function takes as parameter an instance of `Sensor`. A sensor is described by various parameters, including the source URL where the sensor data can be retrieved. Other parameters are the name and the type of the sensor, a plain-text description, as well as the location of the sensor. Each sensor must also be assigned to a user who in turn needs to be a registered user of LSM. The following example shows how to add a new virtual sensor.

```
import java.util.Date;
import lsm.beans.Place;
import lsm.beans.Sensor;
import lsm.beans.User;
import lsm.server.LSMTripleStore;

public class TestLSM {

  public static void main(String[] args) {
    try{
      /*
      * add new sensor to lsm store. For example: Air quality sensor from Lausanne
      */
      // 1. Create an instanse of Sensor class and set the sensor metadata
      String sourceURL =
        "http://opensensedata.epfl.ch:22002/gsn?REQUEST=113&name=lausanne_1057";
      Sensor sensor  = new Sensor();
      sensor.setName("lausanne_1057");
      sensor.setAuthor("user_name");
      sensor.setSourceType("lausanne");
      sensor.setInfor("Air Quality Sensors from Lausanne");
      sensor.setSource(sourceURL);
      sensor.setMetaGraph("http://lsm.deri.ie/yourMetaGraphURL");
      sensor.setDataGraph("http://lsm.deri.ie/yourDataGraphURL");
      sensor.setTimes(new Date());

      // Set sensor location information (latitude, longitude, city, country, continent...)
      Place place = new Place(); place.setLat(46.529838); place.setLng(6.596818);
      sensor.setPlace(place);

      // Set sensor's author (has to be a valid account for LSM)
      User user = new User(); user.setUsername("user_name"); user.setPass("password");
      sensor.setUser(user);
```

[16] http://ftp.heanet.ie/mirrors/www.apache.org/dist/jena/

```
      // Create LSMTripleStore instance and add new sensor
      LSMTripleStore lsmStore = new LSMTripleStore();
      lsmStore.setUser(user);
      lsmStore.sensorAdd(sensor);

    } catch (Exception ex) {
       ex.printStackTrace();
       System.out.println("cannot send the data to server");
    }
  }
}
```

Updating sensor data: Updating the data of a sensor is done via an instance of the class `Observation`. For each observation, a sensor existing in LSM needs to be set. Furthermore, an observation can have various properties, such as a temperature value or a CO2 value, represented by instances of the class `ObservationProperty`. To eventually update the sensor data, the class `LSMTripleStore` has the function `sensorDataUpdate(observation)`. The following example illustrates the update process.

```
...
  //create an Observation object
  Observation obs = new Observation();

  // set SensorURL of observation, for example
  String sensorURL = "http://lsm.deri.ie/resource/8a82919d3264f4ac013264f4e14501c0"
  obs.setSensor(sensorURL);

  //set time when the observation was observed.
  //In this example, the time is current local time.
  obs.setTimes(new Date());

  // Relation linking an Observation to the Property that was observed
  ObservedProperty obvTem = new ObservedProperty();
  obvTem.setObservationId(obs.getId());
  obvTem.setPropertyName("Temperature");
  obvTem.setValue(9.58485958485958);
  obvTem.setUnit("C");
  obs.addReading(obvTem);

  ObservedProperty obvCO = new ObservedProperty();
  obvCO.setObservationId(obs.getId());
  obvCO.setPropertyName("CO");
  obvCO.setValue(0.0366300366300366);
  obvCO.setUnit("C");
  obs.addReading(obvCO);

  lsmStore.sensorDataUpdate(obs);
...
```

Retrieve sensor objects by sensor URL id or by sensor source: A sensor can be retrieved by either the unique URL assigned by LSM or by the URL where the data is published/accessible. Depending on the method used, the class `LSMTripleStore` provides the two functions `getSensorById(sourceURL)` and `getSensorBySource(sourceURL)`. The example below illustrates both alternatives.

```
...
  String sensorURL = "http://lsm.deri.ie/resource/8a82919d3264f4ac013264f4e14501c0";
  Sensor sensor1 = lsmStore.getSensorById(sensorURL);

  String sourceURL = "http://opensensedata.epfl.ch:22002/gsn?REQUEST=113&name=lausanne_1057";
  Sensor sensor2 = lsmStore.getSensorBySource(sourceURL);
...
```

Delete sensors and sensor data. Finally, the API allows deleting sensors (`sensorDelete`) and sensor readings (`deleteAllReadings`). The function `deleteAllReadings` of class `LSMTripleStore` allows the specification of additional parameters to filter/limit the set of sensor reading to be deleted. In the following example, the readings of a sensor are deleted depending on a specific timestamp. More specifically, all readings collected before a given timestamp are deleted from the knowledge base.

```
...
  // remove sensor
  lsmStore.sensorDelete("http://lsm.deri.ie/resource/8a82919d3264f4ac013264f4e14501c0");

  // delete all reading data of specific sensor
  lsmStore.deleteAllReadings("http://lsm.deri.ie/resource/8a82919d3264f4ac013264f4e14501c0");

  // delete sensor data at a certain period of time
  Date fromDate = new Date();
  lsmStore.deleteAllReadings("http://lsm.deri.ie/resource/8a82919d3264f4ac013264f4e14501c0",
                           "<=", fromDate, null);
...
```

3.2 Filtering Capabilities

To limit the number of sensors and/or sensor data made available for defining new data services is a fundamental task in OpenIoT. For example, sensors might be filtered according to their location or manufacturer, and sensor data might be filtered according to their minimum level of accuracy. The universal need for such filtering capabilities indicates that it is optimal to implement them within the OpenIoT Edge Server, to make their usage application-agnostic. The filtering can be done on the level of physical sensors, virtual sensors (i.e., within GSN) or within LSM. The following subsections elaborate on this.

3.2.1 Sensor Level

OpenIoT handles data streams that are integrated in the cloud via the sensor middleware (i.e. GSN). GSN allows data collection from different types of sensors (such as physical devices, signal processing algorithms, information fusion algorithms), which are integrated on the basis of GSN's "virtual" sensor concept. Depending on their complexity, virtual sensors can support basic filtering capabilities, that are applied prior to the integration of the virtual sensor data streams into the OpenIoT cloud. This initial filtering layer complements the higher level data filtering capabilities of the OpenIoT platform, which are described in subsequent paragraphs. One may argue that these higher level filtering algorithms subsume the virtual-sensor level filtering capabilities and hence virtual sensor level filtering is not really useful for OpenIoT services integrators. This is not however always the case, since filtering at the virtual sensor level can in several cases provide distinct advantages over higher level mechanisms. In particular, virtual sensor level filtering should be applied in the following cases:

- When there is a need to drive filtering, counting, and other low-level processing functional at the lowest levels of the IoT architecture, with the aim of saving bandwidth resources, while at the same time optimizing the performance of the applications
- When there is a need to minimise the "business logic" implemented in the OpenIoT system for reasons such as performance or independence from particular middleware products or business processes.

While a variety of filtering algorithms can be implemented at the virtual sensor level, OpenIoT supports filters compliant to the EPC Global Architecture (EPCglobal-ARCH 2007) and standards and more specifically filters compliant to the EPC ALE (Application Level Events) (EPCglobal-ALE 2009) and EPC LLRP specifications. OpenIoT's support for GS1 EPC standards lends itself to the integration of EPC compliant platforms with the OpenIoT middleware, and in particular the integration of SENSAP's S-Box product and the AspireRfid platform. An exhaustive presentation of the respective filtering capabilities is beyond the scope of this document and can be found in the respective standards. In a nutshell, the EPC-based virtual sensor level filtering of OpenIoT supports:

- Filtering of specific data fields of EPC tags, along with guidelines on the particular formatting of the tag data that will be consumed by higher layers in the OpenIoT architecture.
- Filtering on the basis of object classes, product types or tag types.
- Definition and filtering on the basis of specific cycles of events, which are defined based on: (a) Specific time-periodicity or intervals, (b) External triggers or events.
- Control of the filtering of tags depending on whether they are read for the first time (additions) or whether they disappear (deletions).

The filtering rules are defined on the basis of declarative specifications of patterns that have to be reported (Include Patterns) or excluded (not reported) (Exclude Patterns) as shown in the table below (Table 2).

Table 2: EPC Filtering using includePatterns and excludePatterns.

```
F ( R )  = { epc  |  epc in R & epc in I1 & ... & epc in In & epc not in E1 & ... & epc not in En }
```

where `Ii` denotes the set of epc tags matched by the `ith` pattern in the `includePatterns` list, and Ei denotes the set of epc tags matched by the `ith` pattern in the `excludePatterns` list.

Some concrete filtering examples based on corresponding (include) patterns are illustrated in

Table 3: Virtual Sensor Level Filtering Examples based on specific include patterns.

```
1. Single EPC Pattern: urn:epc:pat:gid-96:18.340.7750 (Reports the specified GID-96 tag with ObjectClass=340 and Serial Number = 7750 for the corporation with Domain Number = 18, filters out any other readings)
2. Wildcard on the serial number: urn:epc:pat:gid-96:18.340.* (Reports all tags with ObjectClass=340 for the corporation with Domain Number = 18, regardless of their Serial Number, filters out any other readings)
3. Reference Range for specific items: urn:epc:pat:gid-96:18.[331-340].*, (Reports all tags with ObjectClass between 331 and 340 for the corporation with Domain Number = 18, regardless of their Serial Number, filters out any other readings)
```

Note that an exhaustive presentation of all the filtering capabilities of the EPC Global architecture is beyond the scope of this document. The implementation of these functionalities will be used in the scope of the manufacturing use case of the project (on materials flow traceability), which uses RFID tags.

3.2.2 GSN – Sensor Middleware Level

GSN provides two complementary mechanisms for filtering acquired data. The first one is based on a SQL syntax enhanced with specialized semantics for timed sliding windows and event counting. The second one allows manipulating data with specialised processing classes.

Filtering through SQL syntax is done at the level of the wrappers or when combining them to form a new data stream. It shows an example that defines a virtual sensor reading two temperature sensors; and in case both of them have the same reading (above a certain threshold) in the last minute, the virtual sensor returns the latest picture from a webcam located in the same room together with the measured temperature. All the filtering is described using SQL syntax.

```
<virtual sensor name="room monitor" priority="10" protected="false" >
   <processing class>
      <class name>gsn.vsensor.BridgeVirtualSensor</class name>
      <init params/>
      <output structure>
         <field name="image" type="binary:jpeg" />
         <field name="temp" type="int" />
      </output structure>
   </processing class>
   <life cycle pool size="10" />
   <addressing>
      <predicate key="geographical">BC143</predicate>
      <predicate key="usage">room monitoring</predicate>
      <predicate key="latitude">46.5214</predicate>
      <predicate key="longitude">6.5676</predicate>
   </addressing>
   <storage history size="10h" />
   <streams>
   <stream name="cam">
      <source name="cam" storage size="1" >
          <address wrapper="remote">
            <predicate key="geographical">BC143</predicate>
            <predicate key="type">Camera</predicate>
         </address>
         <query>select * from WRAPPER</query>
      </source>
      <source name="temperature1" storage size="1m" >
         <address wrapper="remote">
            <predicate key="type">temperature</predicate>
            <predicate key="geographical">BC143 N</predicate>
         </address>
         <query>select AVG(temp1) as T1 from WRAPPER</query>
      </source>
       <source name="temperature2" storage size="1m" >
         <address wrapper="remote">
            <predicate key="type">temperature</predicate>
            <predicate key="geographical">BC143 S</predicate>
         </address>
         <query>select AVG(temp2) as T2 from WRAPPER</query>
      </source>
      <query>select cam.picture as image, temperature.T1 as temp
             from cam, temperature1
             where temperature1.T1 > 30 AND
             temperature1.T1 = temperature2.T2
      </query>
   </stream>
   </streams>
</virtual sensor>
```

Figure 9: Virtual sensor description file using SQL filtering.

GSN comes with a library of processing classes that you can use without programming. Processing classes' role is not limited to filtering data; they can also be used to trigger actions like generating alerts. Advanced filtering can be performed at the level of processing classes, which allow the implementation of arbitrary processing in Java classes. The table 4 below shows standard processing classes in GSN.

Table 4: Standard processing classes in GSN

Name	**Processing Class**	**Remarks**
Bridge	gsn.vsensor.BridgeVirtualSensor	GSN's default processing class.
SMA Data Cleaner	gsn.vsensor.SMACleaner	A Sample Simple Moving Average Data Cleaner.
GPS Parser	gsn.vsensor.GPSNMEAVS	Parses input from GPS device
Email	gsn.vsensor.EmailVirtualSensor	Creates an e-mail from streaming data. Can be used for alerts.
SMS	gsn.vsensor.SMSVirtualSensor	Creates an SMS from streaming data. Can be used for alerts.

Stream Exporter	gsn.vsensor.StreamExporterVirtualSensor	This virtual sensor saves its input stream to any JDBC accessible source
R	gsn.vsensor.RVirtualSensor	This virtual sensor uses an R server to process data or generate charts.
WebInteractive	gsn.vsensor.WebInteractiveVirtualSensor	This virtual sensor can interact with a web form.
ScheduledBridge	gsn.vsensor.ScheduledBridgeVirtualSensor	Bridge VS with scheduled output
ScheduledStrea m Exporter	gsn.vsensor.ScheduledStreamExporter VirtualSensor	Stream Exporter with scheduled output
Scriptlet	gsn.processor.ScriptletProcessor	Define the Processing Class logic with a scriptlet in the configuration file
DataClean	gsn.vsensor.DataCleanVirtualSensor	Datacleaning processing class
GridRenderer	gsn.vsensor.GridRenderer	Grid renderer processing class
Modelling	gsn.vsensor.ModellingVirtualSensor	Attaching models to data-streams
GridModel	gsn.vsensor.GridModelVS	Build a grid from a model

In addition to Java classes, which require recompiling and restarting GSN, one can define arbitrary processing using the *scriptlet* processing class. This class executes a script upon reception of a new Stream Element and can implement arbitrary complex processing by specifying its logic directly in the virtual sensor description file. This way of implementing processing classes offer a higher level of flexibility. The current implementation supports the Groovy (http://groovy.codehaus.org) scripting language which offers access to all the GSN java objects and libraries available in the Java class path.

This processor automatically binds the data between the Stream Element and the variables of the scriptlet. It executes the script against each Stream Element available from the Virtual Sensor sources. Upon reception of a new Stream Element, a Context containing all its variables (and their values) is created and passed to the script for execution. When the script completes, it returns the updated Context containing the updated variables as well as those created in the script. Based on the returned Context, a new Stream Element matching the output structure defined in the Virtual Sensor description file is created and may be stored. Figure 10 shows a sample virtual sensor with the scriptlet processing class for arbitrary filtering. In this example, fields (HEAP_CAL and NON_HEAP_CAL) are filtered through the groovy scriptlet. Each time new stream elements arrive to the processing class, they are filtered that way.

```
<virtual-sensor name="MemoryMonitorCalibratedVS" priority="10">
    <processing-class>
        <class-name>gsn.processor.ScriptletProcessor</class-name>
        <init-params>
            <param name="persistant">true</param>
            <param name="scriptlet">
                <![CDATA[
                // fields HEAP_CAL NON_HEAP_CAL
                // and are filtered through the following code
                def p1 = -3.171e-23;
                def p2 = 1.868e-19;
                def p3 = -4.779e-16;
                def p4 = 6.957e-13;
                def p5 = -6.337e-10;
                def p6 = 3.735e-7;
                def p7 = -0.0001421;
                def p8 = 0.03357;
                def p9 = -4.463;
                def p10 = 258.4;

                def rawValue = HEAP;


                HEAP_CAL = p1 * Math.pow(rawValue, 9) + p2 * Math.pow(rawValue, 8)
+ p3 * Math.pow(rawValue, 7) + p4 * Math.pow(rawValue, 6) + p5 * Math.pow(rawValue,
5)  +  p6  *  Math.pow(rawValue,  4)  +  p7  *  Math.pow(rawValue,  3)  +  p8  *
Math.pow(rawValue, 2) + p9 * rawValue + p10;


                NON_HEAP_CAL = HEAP_CAL / 1000.0;


                if (NON_HEAP_CAL <= 1) {
                        NON_HEAP_CAL  =  -20.0  *  (NON_HEAP_CAL  *  (1.0  +  0.018  *
(NON_HEAP_CAL - 24.0)) - 0.55);
                } else if (NON_HEAP_CAL <= 8) {
                        NON_HEAP_CAL  =  (-3.213  *  NON_HEAP_CAL  -  4.093)  /  (1.0  -
0.009733 * NON_HEAP_CAL - 0.01205 * NON_HEAP_CAL);
                } else {
                        NON_HEAP_CAL = -2.246 - 5.239 * NON_HEAP_CAL * (1.0 + 0.018
* (NON_HEAP_CAL - 24.0)) - 0.06756 * Math.pow(1.0 + 0.018 * (NON_HEAP_CAL - 24.0),
2);
                }


                ]]>
            </param>
        </init-params>
        <output-structure>
            <field name="HEAP_CAL" type="double"/>
            <field name="NON_HEAP_CAL" type="double"/>
        </output-structure>
    </processing-class>
    <description/>
    <addressing/>
    <storage history-size="1"/>
    <streams>
        <stream name="stream1">
            <source alias="source1" storage-size="1" sampling-rate="1">
                <address wrapper="local">
                    <predicate key="name">MemoryMonitorVS</predicate>
                </address>
                <query>select * from wrapper</query>
            </source>
            <query>select HEAP_CAL, NON_HEAP_CAL , timed from source1</query>
        </stream>
    </streams>
</virtual-sensor>
```

Figure 10: Virtual sensor with scriptlet processing for arbitrary filtering.

3.2.3 LSM – Semantic Level

LSM transforms the data from virtual sensors into Linked Data stored in RDF. The de-facto language to query RDF is SPARQL [Prud'hommeaux 2008]. A SPARQL query is a so-called one-shot query, i.e., the query is issued to and executed by the system, immediately returning a result (that might be empty). The query is executed over the currently, i.e. at the execution time, available data. In the context of LSM, or IoT application in general, such queries typically refer to queries about sensor metadata and historical sensor readings. The SPARQL endpoint of LSM provides the interface to issue these types of queries. The currently deployed RDF triple store by LSM, OpenLink Virtuosos, provides a Linked Data query processor that supports the SPARQL 1.1 standard. SPARQL provides various features to filter the queried data as outlined in Section 3.2.3.1.

While SPARQL queries are executed once over the entire collection and discarded after the results are produced, queries over Linked Stream Data are continuous. Continuous queries are first registered in the system, and continuously executed as new data arrives, with new results being output as soon as they are produced. For processing continuous queries over Linked Stream Data, the LSM provides the CQELS engine [Le-Phuoc et al., 2011]. The query processing in CQELS is done in a push-based fashion, i.e., data entering the query engine triggers the processing. The continuous queries are expressed in the CQELS language, which is an extension of SPARQL 1.1 standard. In a nutshell, CQELS extends SPARQL by new constructs that define filters over the referenced data streams within a query (discussed in more detail in Section 3.2.3.2).

Throughout the rest of this deliverable, to simplify the presentation of examples, we use the following namespaces prefix bindings, With that, for example, we can write `ssn:observedBy` instead of `http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#observedby`.

Table 5: Namespace Prefix Bindings

```
@prefix ssn:<http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#> .
@prefix dul:<http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#> .
@prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:<http://www.w3.org/2002/07/owl#> .
@prefix xsd:<http://www.w3.org/2001/XMLSchema#> .
@prefix:<http://www.example.org/ns#> .
```

3.2.3.1 SPARQL Queries Over Archived Data

With filtering as basic operation of information retrieval, and hence part of most query languages, SPARQL also features a variety of ways to filter the queried data. In the following, the most common ways to filter data in SPARQL are presented.

Basic graph patterns: Given that RDF forms a directed, labelled graph for representing information, the most basic construct of a SPARQL query is a so-called *basic graph pattern*. Such a pattern is very similar to a RDF triple with the exception that the subject, predicate or object may be a variable. A basic graph pattern matches a sub-graph of the RDF data when RDF terms from that sub-graph may be substituted for the variables and the result is RDF graph equivalent to the sub-graph. Using the same identifier for variables also allows combining multiple graph patterns. To give an example, the SPARQL query in **Figure 11** return all observations including the corresponding observation time of a resource `r1`.

```
select ?obs, ?obsTime
where {
  ?obs ssn:observedBy <http://lsm.deri.ie/resource/r1> .
  ?obs ssn:observationResultTime ?obsTime
}
```

Figure 11: SPARQL query with filtering according to a graph pattern

Each non-variable subject, predicate or object of each basic graph pattern effectively results in a filter operation over the data. The previous example features two basic graph patterns. In the first one, the predicate and the object are constants; in the second basic graph pattern only the predicate is a constant. Both basic graph patterns form a combined graph pattern with `?obs` as shared variable.

FILTER statements. While basic graph patterns involve an implicit filtering of the data, the `FILTER` keyword allows the formulation of explicit filter operations. Most basic FILTER statements are Boolean expressions that compare a variable subject, predicate or object with a constant value according to comparison operators (<, ≤, !=, =, ≥,>). The SPARQL query in **Figure 12** extends previous example query (see **Figure 11**) by adding a FILTER statement to limit the observation to ones recorded after a specific timestamp.

```
select ?obs, ?obsTime
where {
  ?obs ssn:observedBy <http://lsm.deri.ie/resource/r1>.
  ?obs ssn:observationResultTime ?obsTime.
  FILTER ( ?obsTime > "2012-01-11"^^xsd:date )
}
```

Figure 12: SPARQL query with simple `FILTER` statement.

Besides simple comparisons, SPARQL comes with built-in functions that can be executed over subjects, predicates, or objects and return a `TRUE` or `FALSE`. For example, SPARQL allows the evaluation of regular expressions over string values using the built-in `REGEX` function. Other Boolean functions include, e.g., `isNumeric` and `isLiteral` to test if an RDF term is a numeric or string value. A full list of such functions is available in the official standard[17]. Within a single `FILTER` statement, several Boolean expressions can be combined via the logical operators `&&` (AND) and `||` (OR).

In general, variables in basic graph patterns are bound to each possible solution. Consider, for example, that an observation can feature several observation times. As a result the basic graph pattern `?obs ssn:observationResultTime ?obsTime` will match several triples for each observation `?obs`. As a result, the query in **Figure 11** might yield multiple results. If the actual values of the observation times are not relevant, but it is only of interest if an observation features no or at least one observation time, a user can revert to the `FILTER NOT EXIST` and `FILTER EXIST` statements. The following query (**Figure 13**) returns all observations observed by a resource `r1` that feature at least one observation time. Note that `?obsTime` can no longer be part of the `SELECT` statement since `?obsTime` is at no time bound to an actual value.

[17] http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

```
select ?obs
where {
  ?obs ssn:observedBy <http://lsm.deri.ie/resource/r1> .
  FILTER EXISTS { ?obs ssn:observationResultTime ?obsTime }
}
```

Figure 13: SPARQL query with `FILTER EXISTS` statement.

Analogously, using `FILTER NOT EXITS` would return all observations observed by resource `r1` that feature no observation time.

Set operations. SPARQL also supports – implicitly or explicitly – the common set operations union, intersection and set difference. Of these three operations, the intersection and set difference can be considered as filter operation since they potentially limit the result set. Regarding an intersection, SPARQL does not feature a dedicated keyword, e.g., `INTERSECT`, like in SQL. Instead, intersection can be formulated by combining multiple basic graph patterns. For example, in **Figure 14**, the SPARQL query returns all observations that have been observed by both resources `r1` and `r2`. Again, the connection between both basic graph pattern derives from the shared variable `?obs`.

```
select ?obs
where {
  ?obs ssn:observedBy <http://lsm.deri.ie/resource/r1> .
  ?obs ssn:observedBy <http://lsm.deri.ie/resource/r2> .
}
```

Figure 14: SPARQL query with implicit intersection.

The set operation, on the other hand, cannot be formulated implicitly. Hence, SPARQL features the `MINUS` keyword for this. The SPARQL query in **Figure 15** represents an alternative formulation of the query in **Figure 13**. Again, the result are all observation observed by resource `r1` that feature at least one observation time. However, this query uses a set operation to remove all observations made by `r1` with no observation time from the final result.

```
select ?obs
where {
  ?obs ssn:observedBy <http://lsm.deri.ie/resource/r1> .
  MINUS {
    ?obs ssn:observedBy <http://lsm.deri.ie/resource/r1> .
    FILTER NOT EXISTS { ?obs ssn:observationResultTime ?obsTime }
  }
}
```

Figure 15: SPARQL query with set difference (`MINUS`).

These simple examples already confirm that there are typically multiple alternatives to formulate the same query. Which alternative is the best in terms of execution performance is beyond of the scope of this deliverable.

DISTINCT – duplicate elimination. From a technical perspective, duplicate elimination can be considered as a filter operation since it potentially removes tuples from the result set. Consider for example the SPARQL query in **Figure 12**. If a user in only interested in the observations but not in the actual observation times – hence, the `SELECT` statement can be simplified to `SELECT ?obs` – the result contains duplicates (again assuming that an observation can have multiple observation times). While this result might be sufficient in some cases it might distort subsequent operations, e.g., if the number of observations is of interest. In these cases, duplicate tuples need to be removed from the result. The corresponding SPARQL query using the DISTINCT keyword looks as follows (**Figure 16**):

```
select COUNT(DISTINCT(?obs))
where {
  ?obs ssn:observedBy <http://lsm.deri.ie/resource/r1>.
  ?obs ssn:observationResultTime ?obsTime.
  FILTER ( ?obsTime > "2012-01-11"^^xsd:date )
}
```

Figure 16: SPARQL query with duplicate elimination (`DISTINCT`).

Note that the previous query represents an aggregate query, counting the number of observations using `COUNTS`. Aggregate queries are discussed in Section 3.3.3.

3.2.3.2 CQELS Queries Over Data Streams

Since CQELS is an extension to SPARQL 1.1, all aforementioned means to filter RDF data are available within CQELS. The difference is that CQELS queries, at least partly, operate on live data streams. In general it is impractical or, particularly in case of high data rates, even technically impossible to consider all incoming data from a stream. Furthermore, most meaningful queries over stream data refer to a limited set of input date like, e.g., the most recent ones within a specific time frame. Stream-based systems use the concept of a *window* to specify that set of relevant data items. As such, the additional filter operations provided by CQELS specify a window for a data stream, i.e., simply speaking, how many and which tuples within a stream are considered to be relevant for the current execution of a query.

CQELS extends the notion of graph patterns to *stream graph patterns*, indicated by the `STREAM` keyword. In principle, CQELS allows the consideration of all tuples of a data stream using the ALL keyword. However, most meaningful queries require the specification of a window. In CQELS this can be based on the current number of tuples with the window as well as based on time. Both means are described in more detail in the following.

TRIPLES – triple-count-based filtering. With triple-count-based filtering, the size of the window depends on a maximum number of triples with a stream of RDF triples. For example, `TRIPLES 10` denotes a window of size 10, containing the last 10 received triples of a stream. Each incoming new triple will replace the oldest triple in the window. To give an example, the CQELS query in **Figure 17** returns the last 10 detected RFID tags of persons in Room 123. In principle, the window can contain duplicates, i.e., the same person has been detected several times.

```
select ?person
where {
  stream <http://deri.org/streams/rfid> [TRIPLES 10]
     ?person :detectedAt ?location .
  ?loc rdf:name :room-123
}
```

Figure 17: CQELS query with triple count-based filtering.

Naturally, triple count-based windows have a fixed maximum size. On the other hand, this window type is agnostic regarding the time the individual triples were added to the window. Thus, in the previous example, the result may contain persons that have been detected far back in the past.

NOW and RANGE – time-based filtering. Time-based filtering considers the time triples have entered the system. This includes that – compared to triple count-based filtering – a time-based window does not feature a fixed maximum size. Apart from the considered time span specified by a user, the size of the window depends on the arrival rate of new triples in the data streams. The higher the rate the more triples are in the window.

Typically (but necessarily) queries over data streams refer to the most recent data. Often this means that only the latest input data are relevant for a query. For this case, CQELS provides the keyword `NOW`. Note that `NOW` is, in general, not the same as `TRIPLES 1` since several triple may arrive at the same time. For example, if 5 triples arrive together, `NOW` would consider all 5 of them, while `TRIPLES 1` would be executed 5 times, each time with just one of the triples. The CQELS query in **Figure 1** uses NOW to continuously return the current person detected in Room 123.

```
select ?person
where {
  stream <http://deri.org/streams/rfid> [NOW]
     ?person :detectedAt ?location .
  ?loc rdf:name :room-123
}
```

Figure 18: CQELS query with time-based filtering using `NOW`.

For the general case, i.e., that all received triples of a certain “age” are considered within a query, CQELS provides the keyword `RANGE` to specify the length of a window. The following query (see **Figure 19**) continuously returns all persons detected in the last minute in Room 123, each time a new person has been detected in that room.

```
select ?person
where {
  stream <http://deri.org/streams/rfid> [RANGE 60s]
     ?person :detectedAt ?location .
  ?loc rdf:name :room-123
}
```

Figure 19: CQELS query with time-based filtering using `RANGE`.

With no further details, like in previous example, RANGE specifies a sliding window – that is, triples are considered relevant for a query as long as they fall the specified time span. In other words, each triple within a window “expires” at its specific given time. CQELS allows altering this mechanism using the keyword `SLIDE`. Two alternatives uses of `SLIDE` are possible. Firstly, `RANGE 60s SLIDE 5s` (with 5 seconds as example duration) specifies a *hopping window* that the sliding window is moved every 5 seconds. As a result, even if two triples have arrived at different times, they both expire at the same time if they fall into the same 5 second interval. Intuitively, the time interval for `SLIDE` (here: 5s) needs to be smaller than the one for `RANGE` (here 60s). And secondly, `RANGE 60s TUMBLING` specifies a tumbling window. A tumbling window ensures that all triples within a window are considered only once for a query, i.e., all triples of a window expire at the same time.

In previous example, all time spans for RANGE and SLIDE have been given in the number of seconds, e.g., 60s or 5s. More generally, the time span is specified by an integer value followed by a unit of time. Regarding the latter, CQELS currently supports days (d), hours (h), minutes (m), seconds (s), milliseconds (ms) and nanoseconds (ns).

3.3 Aggregation Capabilities

Often individual sensor readings are not relevant but instead an aggregated value is required. For example, to increase the accuracy, rooms are equipped with a set of low-cost temperature sensors. From an application perspective, however, only the average room temperature is of interest. It is therefore meaningful to provide application-independent support for the aggregation of sensor data in the edge server. Like for filtering mechanisms, the three levels (physical sensor, virtual sensor (GSN) and LSM) feature different aggregation capabilities.

3.3.1 Virtual Sensor Level

Similar filtering, OpenIoT supports also aggregation functionalities at the Virtual Sensor Level. The rationale of performing aggregation as early as possible is to alleviate the OpenIoT sensor middleware and business logic layers from the burden of performing common aggregation functions, thereby boosting performance and reducing complexity. Thanks to the integration of SENSAP's SBOX product and of the AspireRfid framework to OpenIoT, there is readily available support for EPC related aggregation functions. One of the most prominent functions relates to grouping, which is performed at the EPC-ALE layer and specified on the basis of appropriate grouping specifications. Some examples of grouping patterns and specifications are illustrated in the following table.

Table 6: EPC Grouping Patterns

```
1. Group according to item value: urn:epc:pat:sgtin-64:X.*.*.*:
2. Group by company prefix: urn:epc:pat:sgtin-64:*.X.*.*
3. Group by item and product: urn:epc:pat:sgtin-64:*.X.X.*
4. Creates a different group for each company prefix, including
   in each such group only EPCs having a filter value of 4 and
   serial numbers in the range 100 through 200, inclusive:
   urn:epc:pat:sgtin-64:3.X.*.[100-200]
```

Additional aggregation of sensor value occurs at the level of SENSAP's ITK (Integra Traceability Kiosk) product, which is integrated as a set of virtual sensors to the OpenIoT platform. This aggregation concerns the combination of quality sensors in order to produce quality indicators. This aggregation represents another example of low-level (i.e. virtual sensor level) functionality that is readily available in the OpenIoT platform. This functionality will be illustrated in as part of the corresponding OpenIoT use case implementation, since it is system and application specific, rather than general purpose like the EPC functions.

3.3.2 GSN – Sensor Middleware Level

GSN offers all aggregation functions supported by standard SQL syntax (count, min, max, sum, average). Aggregation is performed on moving windows (expressed by time or number of tuples), and can be described at the level of the wrappers or combination of wrappers as described in section 3.2.2.

3.3.3 LSM – Semantic Level

Aggregate functions: SPARQL, like SQL, features a set of common aggregate functions to be executed over variables: `COUNT`, `SUM`, `MIN`, `MAX` and `AVG`. **Figure 16** shows a simple SPARQL query counting all observations of a resource `r1` after a specific timestamp. Note that that a DISTINCT is often required to guarantee the correctness of the result. SPARQL also features more specific aggregate functions. `GROUP_CONCAT` performs a string concatenation across the values of an expression with a group. The order of the strings is not specified; duplicates are eliminated. The separator character used in the concatenation may be given with the scalar argument delimiter. To give an example, the SPARQL query in **Figure 20** return a string with all observations done by `r1`, separated by a comma (e.g., “obs_12, obs_4, obs_10”).

```
select (GROUP_CONCAT(?obs, ‘, ‘))
where {
  ?obs ssn:observedBy <http://lsm.deri.ie/resource/r1> .
  FILTER EXISTS { ?obs ssn:observationResultTime ?obsTime }
}
```

Figure 20: SPARQL query with `GROUP_CONCAT` aggregation.

`SAMPLE` is aggregate function that returns an arbitrary value from the multiset passed to it. This is useful if a user is interested in what the final result “looks like” before retrieving the whole result. As a technical side note, Virtuoso, the currently deployed RDF triple store by LSM, supports both `GROUP_CONCAT` and `SAMPLE` but requires the specification of the prefix `sql:` when calling both functions, i.e. `sql:GROUP_CONCAT` and `sql:SAMPLE`. Virtuoso also features `GROUP_DIGEST`, an extended version of `sql:GROUP_CONCAT`, with two more arguments: (a) the maximum allowed length of the result, in characters (redundant values will be ignored; if the last value does not fit, then it can be truncated and "..." is placed at the end of the resulting string), and (b) a bitmask of properties (right now only bit 1 is used and others are reserved. If the value of the argument is 1 then duplicate values are ignored; value 0 will allow duplicate values like in case of `sql:GROUP_CONCAT`).

Grouping and group-based filtering: Closely related to aggregation is the notion of grouping. Grouping logically groups all result rows according to a specified set of variables, and as such allows the evaluation of aggregate functions over each group individually. The corresponding keyword is `GROUP BY`. Additionally, using the `HAVING` keyword, filter conditions to be applied over each group can be formulated. In **Figure 21**, the SPARQL query returns the number of all observations for each resource, but only for resources that made more than 5 observations.

```
select ?res (COUNT(DISTINCT(?obs)))
where {
  ?obs ssn:observedBy ?res .
}
GROUP BY ?res
HAVING (COUNT(DISTINCT(?obs)) > 5)
```

Figure 21: SPARQL query with `GROUP BY` and `HAVING`.

ASK queries. An `ASK` query is not a “classic” SPARQL query since it does not return any data as result from binding values to variables in graph patterns. Instead, an ASK query returns `TRUE` or `FALSE` indicating whether a graph pattern matches or not, i.e., if the graph pattern would return at least one result row. The query in **Figure 22** modifies the one in **Figure 20** into an `ASK` query. If resource r1 made at least one observation featuring an observation time the query returns `TRUE`, otherwise `FALSE`.

```
Ask
where {
  ?obs ssn:observedBy <http://lsm.deri.ie/resource/r1> .
  FILTER EXISTS { ?obs ssn:observationResultTime ?obsTime }
}
```

Figure 22: SPAQRL `ASK` query.

The typical use cases are initial questions like “Is there any data that looks like this?” or “Is there any information about that?” Technically speaking, however, `ASK` queries can be considered and used as an aggregate query since they return a single value to describe a characteristic of an underlying graph pattern. For example, if a user is only interested if a resource has made at least one observation, the user can either formulate a `ASK` query (see **Figure 22**) or an aggregate query using `COUNT` and check if the resulting value is larger than 0. In such cases, ASK queries are potentially the better choice since their execution can be stopped as soon as one result has been found. In contrast, the alternative using `COUNT` has to find all matches before the aggregation function can be applied.

3.4 Data / Result Provisioning

LSM as the “highest” layer in the OpenIoT Edge Server is responsible for providing application access to the sensor data. This includes access to both archived data as well as live data streams. To accomplish this, LSM supports various data access mechanisms, presented in more detail in the following.

3.4.1 SPARQL Endpoint

For experienced users LSM features a SPARQL endpoint, allowing them to directly issue SPARQL queries over the archived data. This includes a query page – accessible at http://lsm.deri.ie/sparql – for formulating and executing queries in an ad-hoc manner.

The endpoint is a service provided by the underlying triple store Virtuoso and implements the SPARQL Protocol for RDF (W3C Working Draft 25 January 2006) providing SPARQL query-processing for RDF data available on the open Internet. As such, the SPARQL endpoint can only execute queries over archived data but not over active input streams. Formulating and evaluating queries of streams requires the CQELS components of LSM.

The endpoint allows user to choose from different output formats. Besides HTML to display query results directly in the browser on the query page, results can also be returned in common formats such as XML, RDF/XML, CSV, JSON, and others.

3.4.2 Streaming Channels

In contrast to snapshot queries over archived date, queries over data streams are typically long-running and continuous queries – that is, such queries are registered in the system for specified time and potentially generate and return results depending on the incoming data. To provide and distribute new result, LSM features different streaming channels based on popular protocols.

3.4.2.1 WebSockets

Originally, the Internet was built to be rather static, being a collection of HTML pages linking from on to another, and each HTML page requested once at a time. This kind of communication was and is handled by HTTP which implements a simple request/response paradigm. New application and services over the Internet, however, require a more dynamic or even-driven communication with minimal latency. Common examples, (a) social networking applications, where, e.g. status updates and messages a directly distributed, (b) online games that require the immediate exchange of the status and actions of players, (c) collaborative platforms where users, e.g., work on the same document at the same time, or (d) any real-time applications, such as a stock ticker the shows the latest stock market information.

In general, HTTP is not suitable for such real-time application or services due to its inherent limitations. Firstly, HTTP is half-duplex with traffic flowing in only one direction at a time. And secondly, HTTP adds latency since each request/response interaction requires the establishment of a new connection. Different technologies have been developed to “simulate” real-time capabilities. For example, Ajax [Holdener 2008] enables clients, i.e., web browsers, to poll for updates over HTTP from the server without the need to request the complete HTML page. However, the client is not aware if there are actually new updates available, and therefore polling might lead to many unnecessary requests for updates. Comet [McCarthy 2008] relies in long polling where the client polls for updates and waits for the server until updates are available. It requires two connections and adds unnecessary complexity to the communication. Simply speaking, solutions to realise real-time applications over HTTP use technologies in a way that they were never designed for, often causing a massive resource overhead.

WebSockets [Fette 2011] represent a standard for bi-directional real-time communication between servers and clients. WebSockets allow a long-held single TCP socket connection to be established between the client and server which allows for bi-directional, full duplex, messages to be instantly distributed with little overhead resulting in a very low latency connection. Both the WebSocket API and the WebSocket protocol are standardised which means that the web now has an agreed standard for real-time communication between Internet clients and servers. Compared to HTTP, communication via WebSockets has several significant advantages in the context of dynamic applications. Firstly, each message has only a very small overhead in terms of number of required bytes. Secondly, since WebSockets establish a permanent connection between clients and servers, no overhead for frequently establishing new connections is involved. And lastly, in contrast to polling, messages are only sent when there is new information.

3.4.2.2 PubSubHubbub

PubSubHubbub is an open protocol designed and implemented by Google for distributed publish/subscribe communication on the Internet. It extends the Atom (and RSS) protocol to avoid a periodic polling of subscribers for new updates from publishers. PubSubHubbub is push-based, enabling a near-instant notification of updates

In the terminology of PubSubHubbub, the hub is a server acting as mediator between the publisher and subscriber. In terms of a implementation, the software component of the hub is not required to run a dedicated machine but may, for example, run on the a publisher’s hardware. In essence, a publisher sends an update notification to the hub, which in turn fetches the update and sends it to all subscribers. In more detail, **Figure 23** shows the main communication steps of the PubSubHubbub protocol:

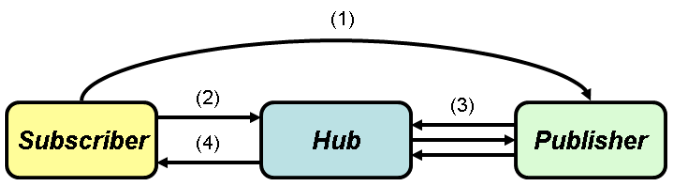


Figure 23: PubSubHubbub basic dataflow.

(1) After a publisher has declares its hub in its Atom or RSS XML file, any subscriber can initially fetch the feed to discover the hub.

(2) Having the required information, the subscriber subscribes to the discovered hub passing the URI of the feed it wants to describe to (note that a hub can manage many feeds in parallel) and the URI at which the subscriber wants to receive updates.

(3) Each time the publisher provides a new update – new or modified content – the publisher notifies the hub about the presence of the update, and the hub fetches the newly published feed.

(4) The hub finally sends the new or changed content to all subscribers subscribed to the corresponding feed using the target URI provided by each subscriber during the subscription process.

PubSubHubbub is designed with large-scale distributed environments in mind featuring big hubs, many small hubs and a very large number of publishers and subscribers. While initially conceived for a publish/subscribe communication of Atom and RSS feeds, the protocol is expected to support any kind of arbitrary content, effectively allowing the subscription to any web resource.

3.4.2.3 XMPP

The e**X**tensible **M**essaging and **P**resence **P**rotocol is an open standard based on XML for the near-instant exchange of messages and presence notifications. The main units of transferred information are called *stanzas*, which, in the context of XMPP, are self-contained XML snippets, for example, including the source and target of a stanza. For an overview, **Figure 24** shows the overall architecture of XMPP:

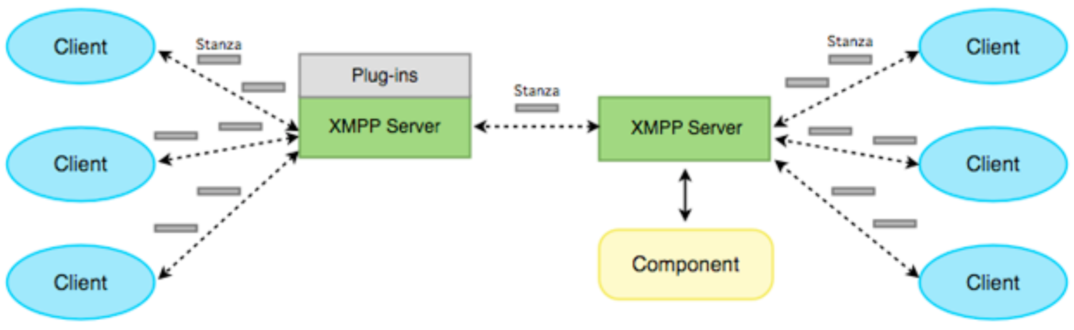


Figure 24: XMPP components and communication architecture.

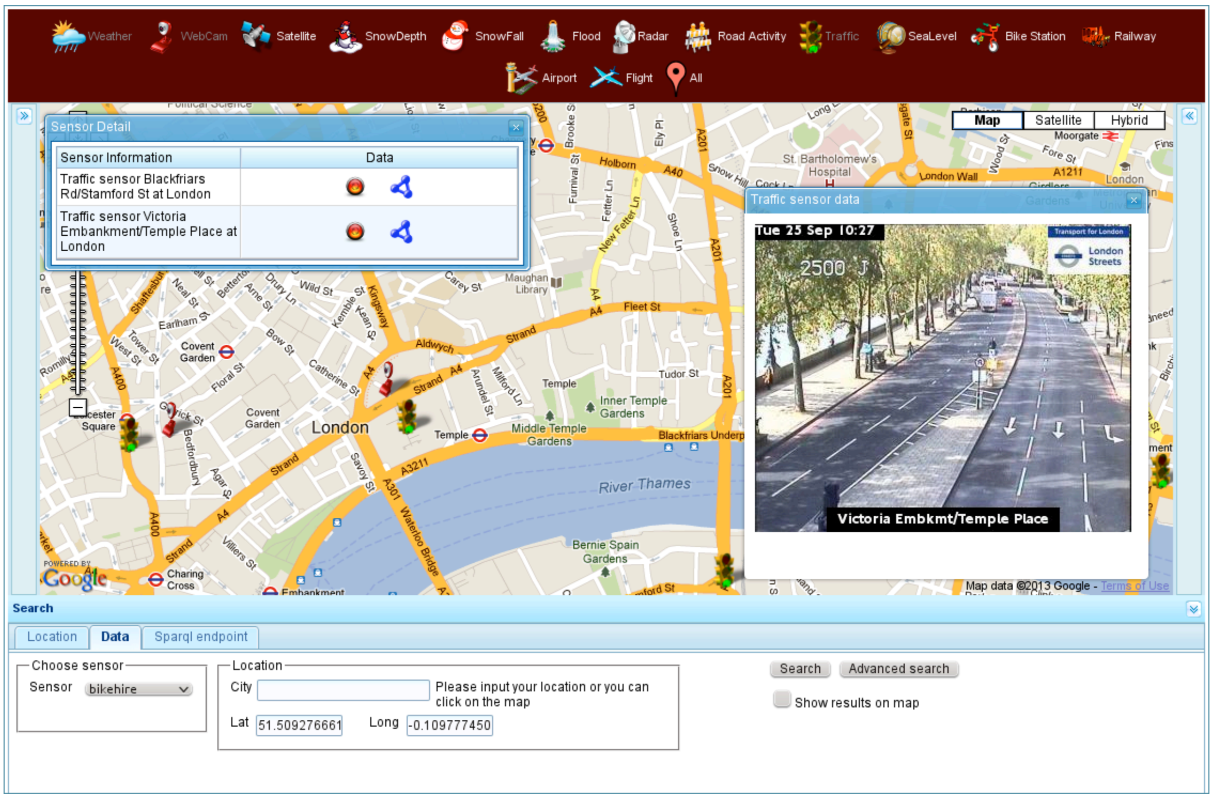
- *XMPP Server:* The essential task of the server is to route stanzas from the source to their destination, whether they are internal from one user to another or from local user to a user on a remote server. Source and destination are specified by a JID (Jabber Identifier[18]), an email address-like identifier essentially containing the name of the user and of the XMPP server.
- *XMPP Client:* A client connects and communicates with the XMPP server according to the protocol. The initial connection typically requires the client's authentication via user name and password, although a server might allow anonymous logins.
- *Component:* Components augment the behaviour of the server by adding some new service. These components have their own identity and address within the server, but run externally and communicate over a component protocol. The component protocol enables developers to create server extensions in a server-agnostic way. An XMPP component effectively becomes part of the server but runs outside (in a separate process) of the XMPP server. So one can shut down the service without affecting the server.
- *Plug-ins:* Many XMPP servers can also be extended via plug-ins. These plug-ins are usually written in the same programming language as the server itself and run inside the server's processes, which makes them not server-agnostic. Their purpose overlaps with external components, but plug-ins may also access internal server data structures and change core server behaviour.

3.4.3 Linked Sensors Explorer

LSM provides a graphical user interface to (a) search for and display registered sensors and (b) add and annotate new sensors. A detailed description of all features can be found in the LSM User Manual[19] **Figure 25** shows a screenshot of the GUI; a demonstration installation with access to over 110,000 sensors is available at http://lsm.deri.ie.

[18] *Jabber* was the original name of the protocol

[19] http://deri-lsm.googlecode.com/files/lsmmanual_v1.1.pdf



Sensor Information	Data
Traffic sensor Blackfriars Rd/Stamford St at London	●
Traffic sensor Victoria Embankment/Temple Place at London	●

Tue 25 Sep 10:27

Victoria Embkmt/Temple Place

Choose sensor
Sensor bikehire

Location
City
Please input your location or you can click on the map
Lat 51.509276661 Long -0.109777450

Figure 25: Linked Sensors Explorer

3.4.3.1 *Sensor Discovery*

The main part of the GUI is a map for displaying the location and type of registered sensors. The example deployment distinguishes between different categories of publicly available sensors (e.g., weather sensors, webcams, flood sensors, trains, flights, etc.) and allows for a selective presentation of sensors of different categories. A sensor is displayed on the map as marker using an appropriate clipart reflecting the sensor’s category. Depending on the current zoom factor of the map, several sensors of the same category are grouped together and represented by a single marker.

Each marker is clickable, revealing different detailed information about the sensor (or sensors in case of grouping). Firstly, user can view the latest and historical data in RDF and can download them as RDF in the N3 format. Secondly, the GUI can show historical sensor data within graph representation. The actual representation of the data depends on the category of the corresponding sensor. For example, for webcams, the provided information is the latest picture. For weather sensors, charts summarising the latest reading are available.

The GUI also features various means to search for locations and sensors. For the basic search, users select a type of sensor and a location – identified by a city name or geo coordinates. The advanced search provides further options such as temporal and spatial filters on the sensor data. As most flexible option for more experienced user, the Linked Sensor Explorer includes a query interface to a SPARQL endpoint. Query results are displayed in an overlying window and can additionally be downloaded as serialized RDF data using the N3 notation.

3.4.3.2 Adding and Annotating Sensors

Registered users are able to add and annotate new sensors. For this, LSM provides an annotation wizard to guide users through the annotation process driven by the supported sensor ontologies.

In the first steps, a user selects the type of the new sensor and specifies the URL from with the sensor data is fed into LSM. In the second step, the user generates the wrapper required to map the incoming feed to input data for LSM. For that, the wizard loads the current data of the feed and enables the users to select the attributes of interest. For each selected attribute, the user needs to map the corresponding XML tags to the sensor properties. If the attribute is a measurement, the user can also define a unit. In the last step, the user sets the metadata for the sensor. This metadata either derives from already supported ontologies or from external ontologies manually added by the user.

Once a sensor is added and annotated it is available in LSM and can be searched for and queried as any other registered sensor.

3.4.3.3 Data feeds

Besides querying archived data using SPARQL, the GUI also enables users to register CQELS queries over data streams into the systems. These queries are then continuously executed as soon new data from relevant data streams, i.e., data streams that are used within the query, arrive. Since such queries may continuously generate results over an, in principle, infinitely long time interval, a presentation of results in form of a result window within the browser is not practical. Instead, LSM offers result for continuous queries in terms of data feeds.

From a user’s perspective, a data feed is a URL identifying the feed. To create a feed, a user first formulates the CQELS to be registered in the system. The user also needs to specify an email address to which the URL of the newly generated feed is sent. With that URL, the user can request the latest results for the formulated query on demand.

4 OPENIOT-SPECIFIC EXTENSIONS

4.1 Ontology-Based Modelling

In terms of data management, OpenIoT is about (sensor) data integration, sharing and discovery, i.e. to deal with the heterogeneity of sensor data from different sources and to support the interoperability between these sources. As such, OpenIoT can leverage existing efforts. This includes, firstly, the Linked Data principles and the involved technologies (RDF, SPARQL, etc.) to provide a common data model, and secondly, the notion of an ontology to add semantics / meaning to data, particularly for the support of data discovery and reasoning. Ontologies are conceptual representations consisting of ontological terms of data and of their relationships, in order to eliminate heterogeneities.

In a nutshell, an ontology typically refers to (a) a controlled vocabulary, i.e. a set of terms with informal natural language definitions that specify meaning, (b) a taxonomy, i.e., a basic hierarchical organisation of the terms of the vocabulary, and (c) additional types of relationships between the terms to specify the meaning of these relationships. In general, ontologies are created for a specific domain to ensure a rather manageable size of the vocabulary. When developing a new ontology it is desirable to reuse existing ontologies as much as possible. This simplifies the development since one can focus on the domain or application-specific knowledge only. It also simplifies the integration/usage of ontologies within applications since well-defined parts of ontologies will be shared. The OpenIoT ontology derives from (a) the requirements resulting from the use case description, i.e. the involved concepts and relationships between concepts, and (b) the set of ontologies relevant to OpenIoT. Here, “relevance” refers to the overlap between the concepts and relationships of the OpenIoT use cases and the ones described by the existing ontologies.

4.1.1 Core Ontologies

Core ontologies refer to all existing ontologies that are reused within OpenIoT – given that the reuse of ontologies is a desirable design goal – and as such are not specific to the requirements of OpenIoT. In the following, this section outlines the core ontologies of the OpenIoT ontology:

Semantic Sensor Network (SSN) ontology[20] is a very comprehensive and robust basis to describe the majority of sensor-related information, i.e., capabilities, physical properties, observations, network characteristics, etc. However, since it has been defined to be cross-domain, it lacks specific information that is required within OpenIoT to cover all defined project use cases. Consequently, as a next step we outline further existing and well-known ontologies that provide additional vocabularies and relations to model the necessary concepts within OpenIoT.

Dolce Ultralite (DUL)[21] is an upper-level ontology, i.e., it is independent from any specific domain or application. Typical examples are ontologies describing the

[20] http://www.w3.org/2005/Incubator/ssn/ssnx/ssn.owl

[21] http://ontologydesignpatterns.org/ont/dul/DUL.owl

concepts of space and time. Such concepts are also important with the OpenIoT uses cases. For example, a user might be interested in sensor values observed in a specific time interval and/or region. Additionally, several ontologies that are reused for the OpenIoT ontology are already aligned to DUL, including the SSN ontology.

Provenance Ontology (PROV)[22]: A major goal of OpenIoT is that providers can make their sensor deployment easily accessible for others over the Cloud. End users use services to discover sensors and sensor deployments that match their requirements. Besides technical requirements (e.g. temperature sensors with a minimum accuracy in a specific area), this might also include requirements towards the provenance of the sensor data, for example, the manufacturer of the hardware or the owner of the sensor deployment.

LinkedGeoData Vocabulary[23] and WGS84 vocabulary[24]: Both vocabularies support the description of geographic locations. The real-world scenarios that OpenIoT is targeting involve points of interest. The location itself is, together with time, one of the fundamental features that characterise sensor readings. Consequently, it is relevant to semantically describe the points of interest and location of a sensor, as it constitutes a part of its surrounding context.

Event Model-F: The Event Model-F ontology[25] provides comprehensive support for both all the structural aspects of events, e.g., metrological, causal, and correlational relationships, and non-structural ones, e.g., time and space, objects and persons involved. With sensors typically streaming data into the Cloud, not only one-shot queries are of relevance but particularly continuous queries. In these cases, typically not the stream of each individual measurement is of interest for end users, but the event when, for example, the value exceeds a predefined threshold. Events in turn can fire other events, leading to complex relations between events, following specific design patterns.

SIOC: The SIOC initiative[26] (Semantically-Interlinked Online Communities) aims to enable the integration of online community information. Cloud computing inherently involves the implicit and explicit formation of online communities. The Cloud is a common platform where different parties provide resources, such as data or services, and other parties consume these resources. For participation, users have to register an account, create and establish a user profile, and can exchange messages, and so on. SIOC aims to model interactive products – in the context of OpenIoT: data from Internet-connected devices and related services – around which communities can grow and feedback can be collected.

Association Ontology (AO): The Association Ontology (AO)[27] specification provides basic concepts and properties for describing specific association statements to something, e.g. an occasion, a genre or a mood, and enables furthermore, a mechanism to like/rate and feedback these associations in context to something on

[22] https://dvcs.w3.org/hg/prov/raw-file/default/ontology/ProvenanceOntology.owl

[23] http://linkedgeodata.org/ontology

[24] http://www.w3.org/2003/01/geo/wgs84_pos

[25] http://events.semantic-multimedia.org/ontology/2008/12/15/model.owl

[26] http://rdfs.org/sioc/spec/

[27] http://purl.org/ontology/ao/core#

or for the Semantic Web. Besides SIOC, the Association Ontology provides additional features from the social/community context. It associates any kind of comment or feedback from each community member, with any other kind of artefact. In OpenIoT such artefacts can be individual sensors, complete deployments, services, other users, etc.

SPITFIRE ontology: The SPITFIRE ontology[28] aligns DUL and Event Model-F – whose relevance has been already motivated – with SSN according to a well-defined Linked Sensor Data model. Network component and energy concepts are also aligned to support energy saving issues both outside and inside a network. Since in OpenIoT resource-constrained devices like sensors are the main information source, this is relevant to the project for an energy-wise usage of the ontologies.

LSM vocabulary: One of the main objectives for OpenIoT, is to integrate different sensor information from variety of sources and enable service delivery, utility management and presentation of the data. In order to achieve this objective, the data has to be collected, transformed analysed and presented by using specific and well-defined units and reference metrics. LSM enables the necessary annotation of units for measurement, raw values and other specific levels of granularity that are not considered/present in other introduced and studied vocabularies.

Ontology for IT Services delivered via the Cloud[29]: The Ontology developed for IT service lifecycles on the Cloud integrates all the processes and data flows that are needed to automatically acquire, consume and manage services on the Cloud. Within OpenIoT, end users access/use the platform by requesting services. With the Cloud as central component of the aspired OpenIoT platform, ontologies that allow modelling Cloud-related concepts, particularly the life cycle of services, represent an integral part for the OpenIoT ontology.

4.1.2 OpenIoT-Specific Extensions

While existing ontologies already cover a broad spectrum of notions required within OpenIoT, they are not sufficient to describe all conceivable data, processes, etc. derived from the project uses cases. As a result, the following concepts have been as additional vocabulary to represent the OpenIoT ontology. The full OpenIoT vocabulary with all defined concepts is presented in Deliverable 3.1.1.

Virtual Sensors: The Semantic Sensor Network (SSN) ontology, providing the most important core vocabulary, defines the notion of sensor and physical devices in general. OpenIoT however, further involves the concept of virtual sensors. Virtual sensors – the basic concept in GSN which is one core element of the OpenIoT platform – represent new data sources created from live data. These virtual sensors can filter, aggregate or transform the data. We therefore have to distinguish between virtual and physical sensors. On the other hand, from an end-user perspective, both virtual and physical sensors are very closely related concepts since they both, simply speaking, produce data.

[28] http://spitfire-project.eu/ontology/ns/

[29] http://ebiquity.umbc.edu/ontologies/itso/1.0/itso.owl

Utility metrics: In the scope of OpenIoT, a variety of utility-based algorithms will be designed and deployed, notably addressing tasks such as resource management, utility-driven privacy and utility-driven security mechanisms. In addition to resource management, optimization, privacy and security, utility metrics will serve as a basis for accounting and management of SLAs (Service Level Agreements) between the OpenIoT cloud services providers and end users. These utility metrics will be recorded as part of the implementation of the Utility Manager component of the OpenIoT platform, while they will be also used to drive the utility-based mechanisms of the project. The definitions of the different utility metric closely follow the classification of the metrics as presented in Deliverable D4.2.1., and can be distinguished between (a) utility metrics for physical sensors, (b) utility metrics for virtual sensors and (c) utility metrics for sensor networks and application level.

4.2 Mashup Development

LSM, as highest layer of the OpenIoT Edge Server, provides with both SPARQL and CQELS powerful mechanisms to query archived data as well as incoming data streams. However, to make full use of this power requires comprehensive knowledge in the formulation of SPARQL and CQELS queries. Furthermore, beside the basic knowledge, formulating (complex) queries can be a time-consuming and error-prone task. To make the functionalities of LSM accessible to non-expert users and to support a quick and easy query formulation, the Super Stream Collider (SSC) platform and tools have been developed [Nguyen-Mau 2012]. SSC provides a web-based interface and tools for building sophisticated mashups combining semantically annotated Linked Stream and Linked Data sources into easy to use resources for applications. The system includes drag&drop construction tools along with a visual SPARQL/CQELS editor and visualization tools for novice users while supporting full access and control for expert users at the same time. Tied in with this development platform is a cloud deployment architecture which enables the user to deploy the generated mashups into a cloud, thus supporting design and deployment of stream-based web applications in a very simple and intuitive way.

4.2.1 Overview and Basic Architecture

The SSC platform is designed as a classical dataflow/workflow execution environment connecting processing input/outputs through pipelines for creating data mashups. Conceptually, each operator has *n* input streams and one output stream. The inputs can be in any format while the output is RDF. Only the final operator of a workflow can return a format other than RDF, if necessary. Operators can be of three classes: A data acquisition operator is used to collect or receive data from data sources or gateways and can be pull-based or push-based. In these operators the data transformation and alignment can be done to produce a normalized RDF output format. A stream processing operator defines stream processing functionalities in a declarative language, e.g., CQELS. A streaming operator streams the outputs of the final operator of a workflow to the consuming applications.

The operators of the same class are executed on an execution container. An example of an execution container is a continuous query processing engine that is used for stream processing operators. The execution containers are running on networked machines that can be dynamically allocated based on the processing load registered to SSC. **Figure 26** shows an informal high-level view of this architecture.

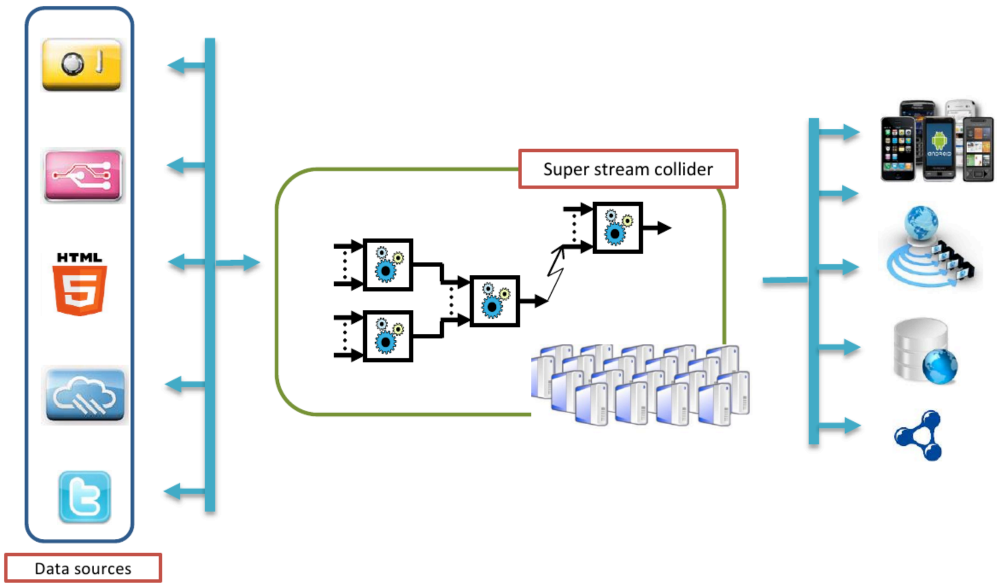


Figure 26: Super Stream Collider - layered architecture.

In a concrete workflow, two connected operators can be executed in different execution containers. For instance, the data acquisition operator for collecting Tweets can stream data via the network to the stream processing engine. The external computing services such as SPARQL endpoints or web services are called external execution containers. To support the easy and intuitive definition of data processing workflows in a “box-and-arrows” fashion, the SSC platform offers a visual programming environment. The interactive process of creating a mashup with SSC features context-aware discovery services for data sources. This process enables the user to incrementally build a workflow in a step-by-step fashion by dragging and dropping the required building blocks and connecting and parameterising them. Also, this supports visually debugging the workflow of the mashup.

A deployment of SSC is online at http://superstreamcollider.org, which provides a user-friendly interface called SSC visual editor. This is a light-weight Web-based workflow editor for composing mashup data through drag&drop. Using the SSC visual editor, we aim at providing a programmable Web environment suitable not only for expert users but also for non-expert programmers. The figure below provides an overview screenshot of SSC with the numbers feature explained in the text below.

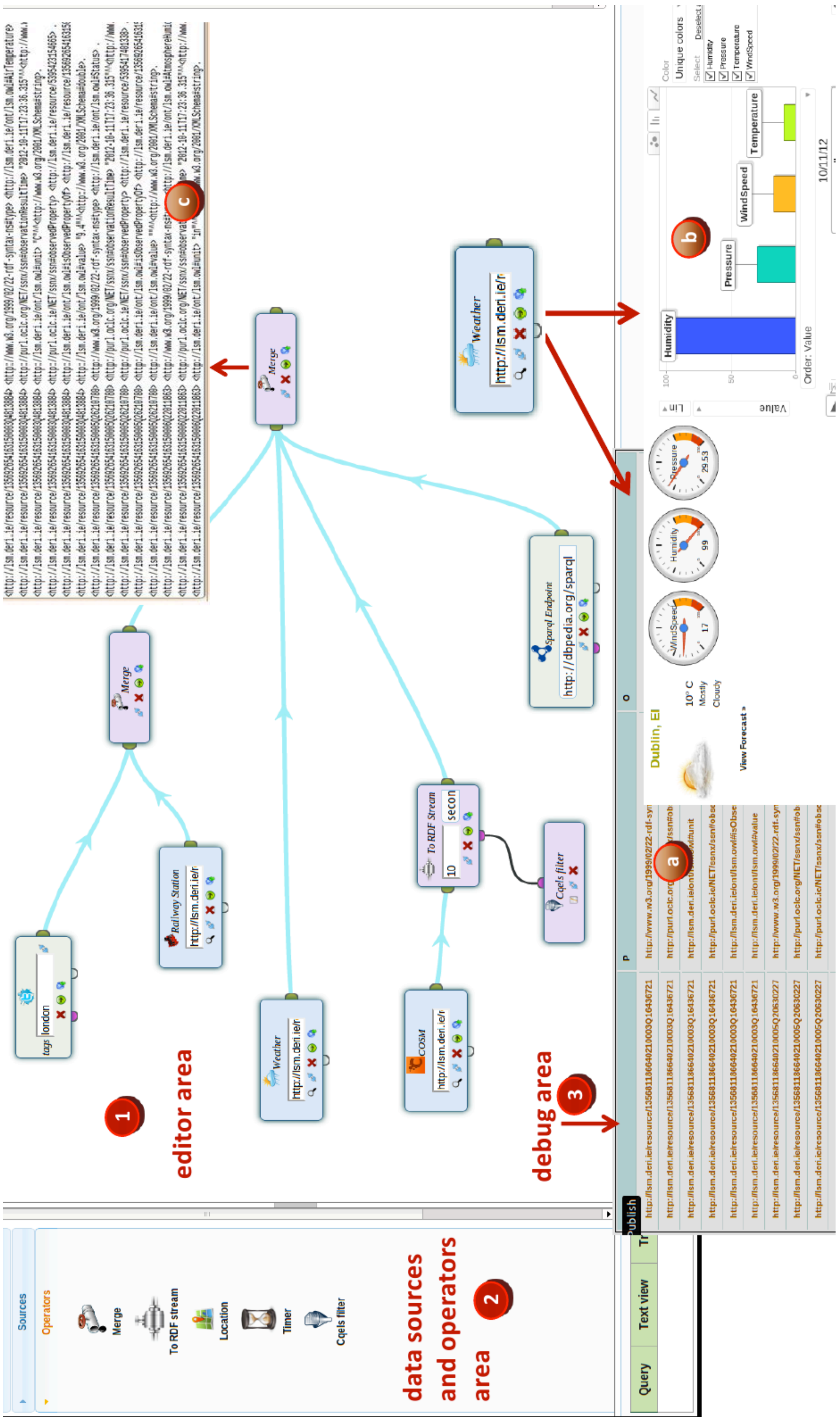


Figure 27: Super Stream Collider - Screenshot.

The main components of the platform are the visual editor (1), data sources and operators (2) and the debugging component (3). Each data source and operator is visualised as a block in the editor area. Visually, a mashup workflow is a combination of connected operators and data sources. It is incrementally designed by dragging the icons representing for corresponding operators in (2) and then dropping to the editor area (1). The flows of data from the sources to the final output are defined by wiring the blocks with configured parameters. The live visualisations of operator outputs are shown in (3). The output of the workflow is a live mashup data stream which can be published, visualised and queried. Currently SSC supports several types of live data sources; see Section 4.2.2. Area (a) in the igure above. shows a temperature sensor as an example. SSC's debugging component supports the user by showing the results of each his/her actions. Area (b) in the figure above shows an example. The result data can be shown raw data, RDF data or can be visualized in different types of charts, so that users can easily monitor their data processing workflows. In the figure above, the output is a merge multiple input streams.

4.2.2 Data Acquisition Operators

The acquisition of data for further processing refers to a set of operators used to collect or receive data from individual sources or gateways, either pull- or push-based. Using a pull-based approach, SSC periodically checks the data source or gateway for new data and, if available, requests them. In case of pushing, the data source or gate actively sends each new data to SSC. The system provides different data acquisition operators depending on the nature of the data source. Currently, SSC provides the following operators:

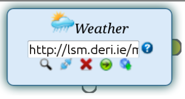
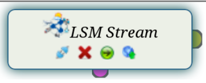
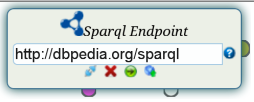
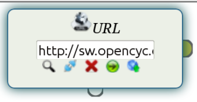
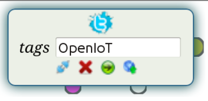
- *LSM sensors:* All sensors registered at LSM can serve as data source for SSC. This includes raw data sources (e.g., from weather stations or traffic cameras) as well as outputs from middleware platforms such as GSN and Xively. LSM sensors are specifies by their unique identifier assigned by LSM during the registration process. To find available sensors, SSC provides a discovery feature searches for sensors of a certain type within a geographic location specified by the user.
- LSM streams: In contrast to one-shot queries that are executed once and return a result, a continuous query over data streams generates new results each time new input data matches the query. To deliver results to any requesting application, SSC supports various standard protocols. Most commonly, SSC uses WebSockets. Further protocols include XMPP and PubSubHubbub (cf. Section 3.4.2).
- SPARQL endpoints: SSC can access public SPARQL endpoints for the execution of SPARQL queries to use the returned result as input for further data processing. The SPARQL Endpoint operator is parameterised with the URL of the endpoint – as list of popular endpoints is already suggested by SSC – and the SPARQL query to be executed.
- Raw data via URL: Using the URL data acquisition operator, data from arbitrary sources that are published via a publicly accessible URL can be

integrated. Again, SSC provides a discovery feature to search for URLs for a given set of keywords. SSC also distinguishes between data formats, including RDF, RDFa[30], hReview[31], adr[32], hCard[33], geo[34].

- Twitter: SCC allows to access live feeds from Twitter and to use them as data source. The set of tweets within a feed depend on the set of specified tags by the users.

Table 7 shows examples of the visual elements for the different data acquisition operators. Note the connectors of each operator to create the connections between operators for creating the overall processing chain.

Table 7: Example of data acquisition operators.

LSM sensor	**LSM stream**	**SPARQL endpoint**	**Public URL**	**Twitter feed**
				

4.2.3 Data Processing Operators

SSC also provides developers with various data manipulation operators. For RDF-based data mashups and data consolidation, we extended and support the operators of DERI Pipes [Le-Phouc 2009]. In more detail, the SSC platform provides the following operators:

- *To RDF stream:* This operator generates an RDF stream from an arbitrary input. For example, if the input is LSM weather sensor, the RDF Stream operator generates a stream by periodically retrieving the current reading of the measurements and to use it as output. The operator is parameterised with the time interval – specified by an integer value and a unit of time (e.g., seconds, minutes) – between generating a new output value. The output of the RDF stream can directly be published via a WebSocket (see Section 4.2.4).
- *CQELS filter:* The RDF stream operator can be extended by additional filters. Working on RDF streams, filters are expressed in the form of CQELS queries. For example, an additional filter of a weather sensor may specify that only temperatures over 20 degrees Celsius are reported, i.e. are forwarded as output of RDF stream operator

[30] http://www.w3.org/TR/xhtml-rdfa-primer/

[31] http://microformats.org/wiki/hreview

[32] http://microformats.org/wiki/adr

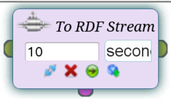
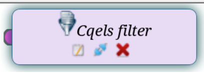
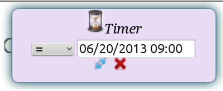
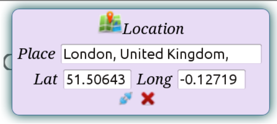
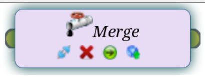
[33] http://microformats.org/wiki/hcard

[34] http://microformats.org/wiki/geo

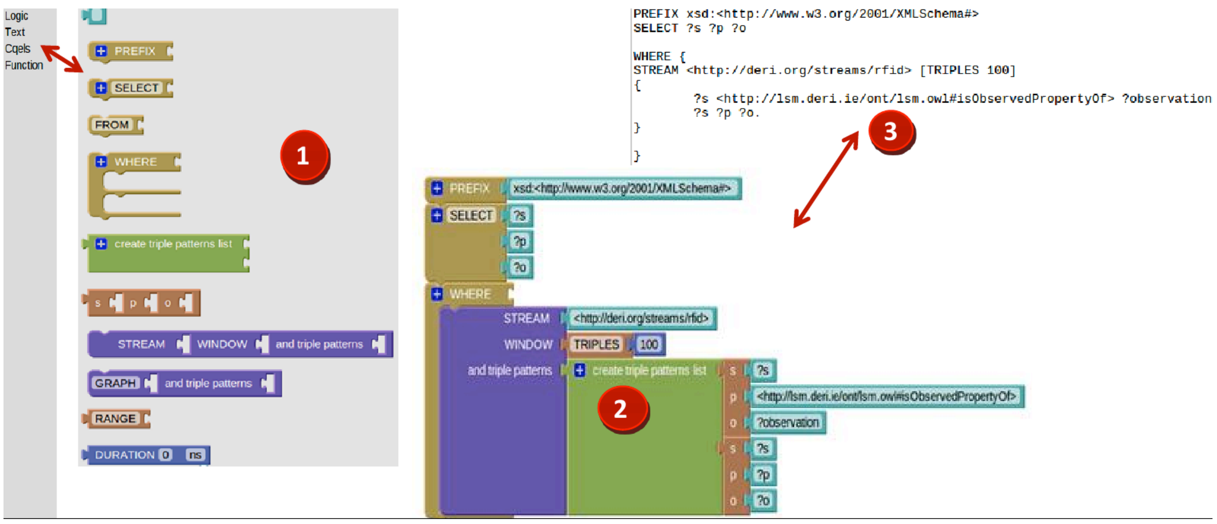
- *Timer:* A timer can be used to activate/enable a data source operator at a specific time for specific period of time. For example, only the traffic information on a particular time of the day may be relevant
- *Location:* The Location operator represents a filter operator that filters according to geographic locations. It can be used to select data from a SPARQL endpoint that refers to a specific location. With this, for example, one can easily retrieve all available data from DBpedia which feature location information and are located in the area of London.
- Merge: The Merge operator takes an arbitrary number of inputs in form of RDF data from other operators and merges all RDF triples into one output. Like for the RDF Stream operator, the output of the Merge operator can be published via a WebSocket (see Section 4.2.4).

Table 8 shows examples of the visual elements for each data processing operator. The parameter for the CQELS Filter, i.e., the CQELS query, is entered into a separated text field.

Table 8: Example of data processing operators.

To RDF Stream	CQELS Filter	Timer	Location	Merge
 To RDF Stream 10 secon	 Cqels filter	 Timer = 06/20/2013 09:00	 Location Place London, United Kingdom, Lat 51.50643 Long -0.12719	 Merge

Given the list of operators and the way to parameterise them, the potentially most complex step is the formulation of CQELS queries for additional filters for RDF Stream operators. To reduce the effort of learning SPARQL and CQELS, SSC also offers visual SPARQL and CQELS editors which enable the user to build SPARQL/CQELS queries interactively and in a step-by-step way; see **Figure 28**. Furthermore, this interactive workflow editing process is leveraged by the context-based discovery services which recommend potentially useful data sources and data items in every step of building a mashup in SSC. These services are powered by Sindice APIs, LSM's sensor database, and other online SPARQL end-points such as Dbpedia, LinkedGeoData, etc. The user can add more knowledge by pointing SSC to further SPARQL endpoints.

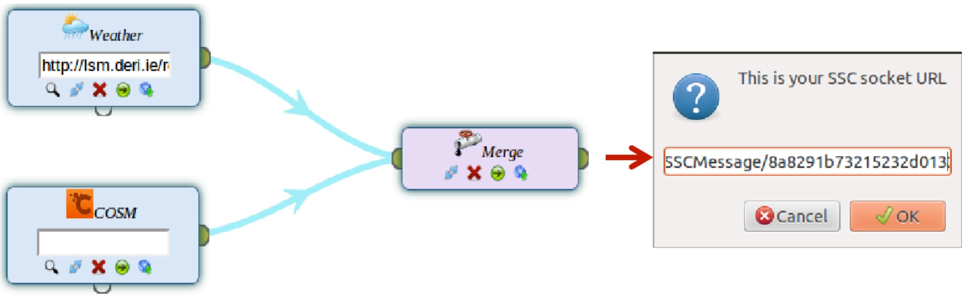

Logic
Text
Cqels
Function
PREFIX
SELECT
FROM
WHERE
create triple patterns list
s p o
STREAM WINDOW and triple patterns
GRAPH and triple patterns
RANGE
DURATION 0 ns
1
2
3
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
SELECT ?s ?p ?o
WHERE
STREAM <http://deri.org/streams/rfid>
WINDOW TRIPLES 100
and triple patterns
create triple patterns list
s ?s p <http://lsm.deri.ie/ont/lsm.owl#isObservedPropertyOf> o ?observation
s ?s p ?p o ?o

```
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
SELECT ?s ?p ?o

WHERE {
STREAM <http://deri.org/streams/rfid> [TRIPLES 100]
{
        ?s <http://lsm.deri.ie/ont/lsm.owl#isObservedPropertyOf> ?observation
        ?s ?p ?o.
}

}
```

Figure 28: Super Stream Collider - visual CQELS editor.

4.2.4 Result Provisioning

The final output of a mashup is a data stream. An important aspect of the SSC platform is that the data produced by a stream mashup can again be published through a web socket URL and thus be re-used as an input by other web applications or mashups. When a mashup is published, it will be assigned to a unique WebSocket URL, e.g., ws://superstreamcollider.org/websocket/8a8291b73215232 as shown in **Figure 29**. Apart from WebSockets to provide output streams of SSC mashups the platform also support streaming protocols such as PubSubHubbub, and XMPP (cf. Section 3.4.2).



This is your SSC socket URL

SSCMessage/8a8291b73215232d013

Figure 29: Super Stream Collider result provisioning via WebSocket.

During the step-by-step creation of a SSC mashup, the platform allows visually debugging of each step. With this debugger, users can immediately see the effects if their actions, e.g., the addition of a new data source or filter. This significantly increases the usability of SSC particularly in case of more complex mashups comprising multiple data acquisition and data processing operators. The result data can be shown as raw data or RDF data or can be visualized in different types of charts, so that users can easily monitor their data processing workflows.

4.3 Cloud Support

In most cases, the “classic” management of Linked Data, e.g., geo-data or DBpedia can be well supported by the existing infrastructure since the data typically change rather slow and infrequently. However, in the context of OpenIoT with its focus on data sources producing streams of data (e.g., sensors, embedded systems, mobile devices, etc.) with a steep, exponential growth predicted in the number of sources and the amount of data, processing has to be performed as soon as new data items become available. Integrating these information streams with other sources will enable a vast range of new “near-real-time” applications. However, due to the heterogeneous nature of the streams and static sources, integrating and processing this data is a difficult and labour-intensive task. Therefore, distributing the processing load of a Linked Stream Data processing engine over networked computers is a promising strategy to deal with the above scalability problems.

Additionally, the trend to Cloud infrastructure, i.e., not owning and operating networked commodity servers but rather renting it on “pay-per-use” basis, provides a further argument for pursuing this strategy. Amazon EC2, Google Cloud, and Microsoft Azure are the most prominent examples of this development. Building a Linked Stream Data processing engine running on such an elastic cluster enables the engine to adapt to changing processing loads by dynamically adjusting the number of processing nodes in the cluster at runtime. This “elasticity” is vital for processing stream data due to its fluctuating stream rates and the unpredictable number of parallel queries which results in hard-to-predict computing complexity and resource requirements. To enable elasticity in a Cloud environment for on-demand load profiles, the used algorithms and data access must lend themselves to parallelisation or must be enhanced with this property. Sequential algorithms and data access will inherently benefit only to a very low degree from deployment in a Cloud environment.

4.3.1 Elastic Execution Model for CQELS Queries

For CQELS queries to be executed within a Cloud environment, extensions to the underlying execution model are required. In a nutshell, the execution model accepts a set of CQELS queries over a set of RDF input streams to continually stream out a set of output streams. These queries will be compiled to a logical query network. This network defines which query algebras the input stream data should go through to produce results for the output streams. For instance, the query network illustrated in **Figure 30** shows that triples are extracted from RDF streams by a set of pattern matching operators (basic graph patterns), and are then sent to a number of sliding window joins (cf. Section 3.2.3.2), and the result triples from this operations are again sent to a set of aggregation operators (cf. Section 3.3.3). The outputs of the aggregation operators become the result streams of the network.

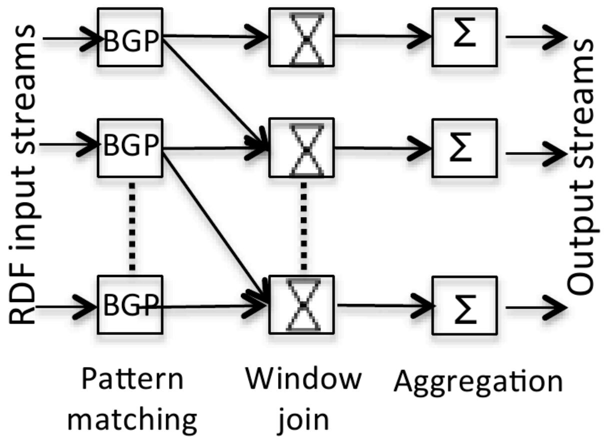


Figure 30: CQELS logical query network.

The execution model is run in a distributed architecture as shown in **Figure 31**. The logical query network is mapped to a processing network distributed among processing nodes, called Operator Containers. The Global Scheduler of the Execution Coordinator uses the Coordination Service to distribute the continuous processing tasks to Operator Containers to trigger the corresponding executions concurrently. The continuous processing tasks are input stream elements associated with signatures that indicate which by which physical operators the stream elements need to be processed to satisfy their processing pipeline (mandated by the original queries). Each Operator Container hosts a set of physical query operators that process input streams and forward the output to the consuming operators in the network. The Local Scheduler of an Operator Container is responsible for scheduling the execution of processing tasks assigned by the Global Scheduler to make the best use of computing resources allocated for that Operator Container.

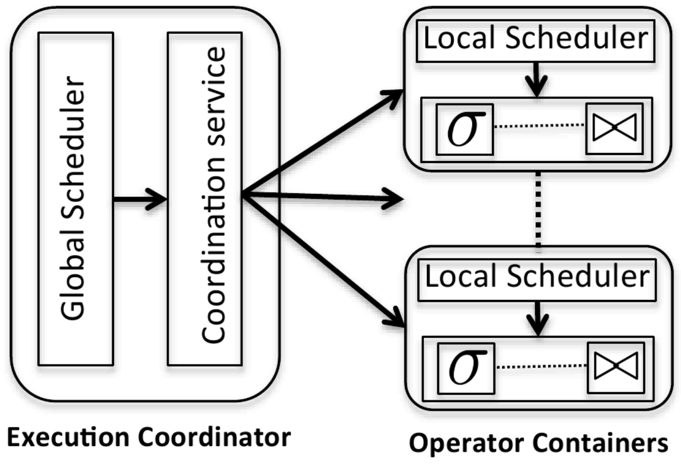


Figure 31 CQELS Cloud execution model.

This execution architecture supports elasticity by allowing the machines running Operator Containers to join and leave the network dynamically at runtime. The Coordination Service monitors the Operator Container instances for failure or disconnection. In the event that an Operator Container instance leaves, the Coordination Service will notify the Global Scheduler to re-balance / re-assign the “missing” processing to the rest of the network. The Coordination Service maintains all processing state of the whole network whereas each Operator Container only has a mirrored copy of the processing state necessary for its processing tasks. When an Operator Container instance leaves, the Coordination Service will recover its processing state (progress) from the last successful processing state and reassigns the tasks to other nodes in the network. When a new Operator Container instance joins, it will notify the Coordination Service its existence to receive tasks. To start processing assigned tasks, each Operator Container has to synchronise its processing state with the processing state of the Coordination Service. To avoid a single point of failure problem through a failure of the Coordination Service, its processing state is replicated among a set of machines.

4.3.2 Implementation Architecture

The architecture of CQELS Cloud comprise of several software components. All components are open source:

- *ZooKeeper* [Hunt 2010]: Apache ZooKeeper provides an open source distributed configuration service, synchronization service and naming registry for large distributed systems. ZooKeeper supports high availability through redundant services. The clients can thus ask another ZooKeeper master if the first fails to answer. ZooKeeper nodes store their data in a hierarchical name space. Clients can read and write from/to the nodes and in this way have a shared configuration service. Updates are totally ordered.
- *Storm*[35]: Storm is a free and open source distributed real-time computation system. Storm makes it easy to reliably process unbounded streams of data, doing for real-time processing what Hadoop did for batch processing. Typical use cases include real-time analytics, online machine learning, continuous computation, distributed RPC and more. Storm is fast, scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.
- *HBase*[36]: Apache HBase is an open source, non-relational, distributed database modelled after Google's BigTable. It is developed as part of Apache Software Foundation's Apache Hadoop project and runs on top of HDFS (Hadoop Distributed Filesystem). It provides a fault-tolerant way of storing large quantities of sparse data. HBase features compression, in-memory operation, and Bloom filters on a per-column basis. Tables in HBase can serve as the input and output for MapReduce jobs run in Hadoop, and may be accessed through the Java API but also through REST and other APIs.

[35] http://storm-project.net/

[36] http://hbase.apache.org/

- *ZeroMQ*[37]: ZeroMQ is a high-performance asynchronous messaging library aimed at use in scalable distributed or concurrent applications. It provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker. The library is designed to have a familiar socket-style API. There are third-party bindings for most popular programming languages, from Java and Python to Erlang and Haskell.

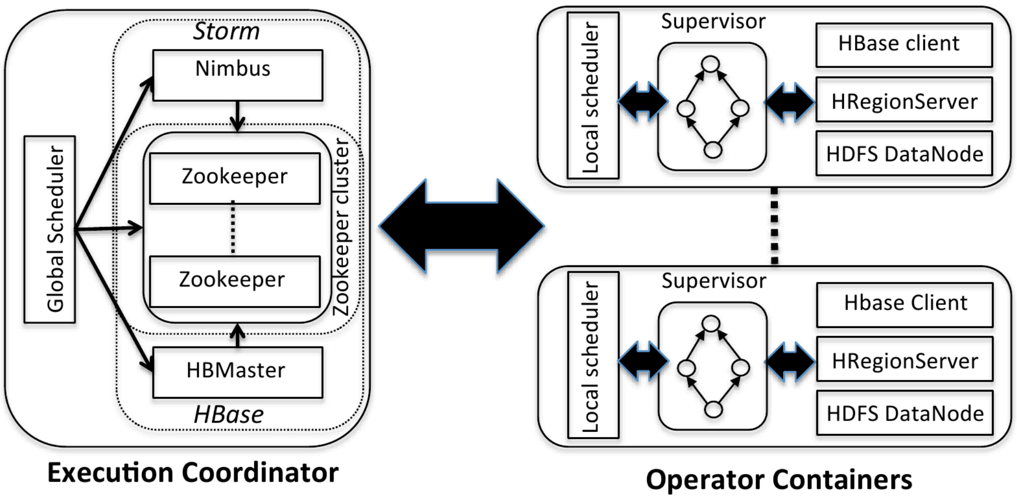


Figure 32: CQELS Cloud infrastructure.

The architecture of CQELS Cloud is shown in **Figure 32**: The Execution Coordinator coordinates the cluster of operator containers using coordination services provided by Storm and HBase which share the same Zookeeper cluster. The Global Scheduler uses Nimbus, an open source EC2/S3-compatible Infrastructure-as-a-Service implementation, to deploy the operators' code to Operator Containers and monitor for failures. Each operator container node runs a Storm supervisor which listens for continuous processing tasks assigned to its machine via Nimbus. The processing tasks that need to process the persistent data use the HBase Client component to access data stored in HBase. The machines running an operator container also hosts the HDFS DataNodes of the HBase cluster. The DataNodes are accessed via the operator container's HRegionServer component of HBase.

Machines running Operator Containers communicate directly without using intermediate queues via ZeroMQ used inside Supervisors. Their available communication bandwidths are optimized by ZeroMQ's congestion detection mechanism. Based on ZeroMQ, Operator Containers use inter-process communication interfaces as defined by Storm's “spouts” (stream source) and “bolts” (processing) infrastructure by sending tuples. Links among spouts and bolt in a Storm topology indicate how tuples should be passed among them. In CQELS Cloud spouts are used to stream data from sources. Bolts receive any number of input streams from upstream processes that trigger processing pipelines and continually output results as new streams. Data is routed to a bolt using a routing policy, called

[37] http://www.zeromq.org/

stream grouping. In CQELS Cloud we use various stream grouping policies provided by Storm.

In CQELS Cloud, input mappings are ordered and put into batches carried by Storm tuples that are routed among bolts and spouts. For optimization purposes, we encode the data, thus, a mappings contains only fixed-size integers. Consequently, batching a series of mappings in an array of integers will reduce the delay of consuming data from the network as well as the serialization and de-serialization. On top of that, this enables us to employs fast compression algorithms such as [Elias 2006], [Lemire 2012] for further speed-up. As the input streams coming to the buffers of an operator instance running in one machine can be unordered, we use the well-established heart-beat approach [Srivastava 2004] for guaranteeing the strict order of stream inputs to ensure the correct semantics of the continuous operators.

We use HBase to store the dictionary for the encoding and decoding operations. The dictionary is partitioned by the keys of RDF nodes to store on Operator Container nodes. The encoding and decoding tasks for RDF nodes for the input and output streams are evenly distributed among the nodes using grouping policies on hash values of the RDF nodes. HBase stores the written data in memory using its MemStore before hashing partitions of the dictionary into sequential disk blocks on the disks of the destined OC nodes. This allows the writing operations of the dictionary encoding to be carried out in parallel and has high throughput on Operator Container nodes. Along with the dictionary, the HBase is also used to store and index huge sets of intermediate results from subqueries on static RDF datasets.

All state is kept in the Zookeeper cluster which enables high access availability from through its built-in replication service. Its reliability, performance and availability can be tuned by increasing the number of machines for the cluster. However, Storm does not directly support state recovery, i.e., resuming a computation state of a node when it crashes. Therefore, we implemented the state recovery for Operator Containers ourselves via timestamped checkpoints and simplifying it by using Storm's concept of guaranteed message processing. Storm guarantees that every input sent will be processed by its acknowledgment mechanism. This mechanism allows a downstream processing node to notify its upstream processing node “up to which time point” it has successful processed downstream inputs. The checkpoint timestamps are encoded in the acknowledgement messages of Storm. This provides an implicit recovery checkpoint, so that when a new node takes over it can restart the computation by reconstructing the processing state up to the last “successful" state.

5 CONCLUSIONS

Two of the major challenges that the envisioned OpenIoT middleware platform faces are (a) the expected high degree of heterogeneity in terms of the variety of Internet-connected devices, and (b) the expected large numbers of devices, users and services requiring scalable solutions. Both challenges affect the design of the OpenIoT Edge Server.

Regarding the heterogeneity of Internet-connected devices, the edge server is, simply speaking, responsible for transforming all data coming from sensors into a unified format. This also provides mechanisms and tools for data providers to inject their data into the platform. To accomplish these goals, the OpenIoT Edge Server features a two-layer solution. In the lower layer is the Global Sensor Network (GSN) component. GSN introduces the notion of virtual sensors to abstract from physical devices, thus providing a unified way for accessing sensor data. On top of GSN runs the Linked Sensor Middleware (LSM) component. LSM takes the output of virtual sensors as input and (a) transforms the data into a unified data format according to the Linked Data format and (b) enriches those data with semantic annotations to enable sophisticated sensor discovery and service orchestration.

Besides access to raw sensor data, the OpenIoT Edge Server also supports filtering and aggregating the data. Filtering and aggregation mechanisms are provided by both GSN and LSM. GSN allows specifying which of the data coming from a physical sensor are considered as output of the derived virtual sensor. Furthermore, a virtual sensor can be a composite of other virtual sensors to aggregate the data. With the data in the Linked Data format RDF, LSM allows the formulation of arbitrary CQELS/SPARQL queries over both archived sensor data and over live data streams. Such queries may include filter or aggregation operations.

With the aim to serve as middleware platform for large numbers of devices, users and services, the edge server also has to address scalability issues. OpenIoT addresses these issues by running resource-consuming tasks in a Cloud setting, either a private or public Cloud. In the context of the OpenIoT edge server, these tasks are primarily archiving large volumes of sensor data and executing queries over that data, as an essential part of the definition and execution of services. Executing queries on archived data is challenging since queries can be arbitrary complex and refer to an arbitrary large subset of the available data. Query execution resides within LSM. As such, LSM implements an elastic execution model for CQELS/SPARQL queries over archived data and data streams in a Cloud environment. The resulting infrastructures leverages from existing, open-source efforts to enable effective an efficient distributed data processing.

As the project advances and the implementation of the OpenIoT Edge server as well as the other platform components mature, the edge server will be subjected to a comprehensive evaluation to investigate whether the aspired goals in terms of performance and scalability are met.

6 REFERENCES

[Elias 2006] P. Elias. Universal codeword sets and representations of the integers. IEEE Trans. Inf. Theor., 21(2):194{203, Sept. 2006.

[EPCglobal-ALE 2009] EPCglobal: The Application Level Events (ALE) Specification, Version 1.1.1 Part I: Core Specification, EPCglobal Ratified Standard, 13 March 2009

[EPCglobal-ARCH 2007] EPCglobal: The EPCglobal Architecture Framework, EPCglobal Final Version 1.2 Approved 10 September 2007

[EPCglobal-EPCIS 2007] EPCglobal: EPC Information Services (EPCIS) Version 1.0.1 Specification Approved September 21, 2007

[Fette 2011] Fette, I., and Melnikov, A. “The WebSocket Protocol.” RFC 6455, Internet Engineering Task Force, http://www.ietf.org/rfc/rfc6455.txt, 2011.

[Holdener 2008] Anthony T. Holdener III. “Ajax: The Definitive Guide”, first ed. O’Reilly, 2008.

[Le-Phouc 2009] D. Le-Phuoc, A. Polleres, M. Hauswirth, Giovanni Tummarello, and Christian Morbidoni, Rapid prototyping of semantic mash-ups through semantic web pipes, in: WWW, 2009, pp. 581–590

[Le-Phouc 2011] D. Le-Phuoc, M. Dao-Tran, J.X. Parreira, M. Hauswirth, A native and adaptive approach for unified processing of linked streams and linked data, in: ISWC, 2011, pp. 370–388

[Lemire 2012] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. CoRR, abs/1209.2137, 2012.

[Hunt 2010] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In USENIX 2010, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association

[McCarthy 2008] Phil McCarthy, Dave Crane. “Comet and Revers Ajax: The Next-Generation Ajax 2.0”, Apress Berkely, CA, USA, 2008

[Nguyen-Mau 2012] H .Q. Nguyen-Mau, M. Serrano, D. Le-Phuoc, M. Hauswirth, Super Stream Collider – Linked Stream Mashups for Everyone, in: ISWC Semantic Web Challenge, 2012

[Prud'hommeaux 2008] Prud'hommeaux Eric, Seaborne Andy, "SPARQL Query Language for RDF". W3C. World Wide Web Consortium, 15 January 2008

[Srivastava 2004] U. Srivastava and J. Widom. Flexible time management in data stream systems. In ACM SIGMOD-SIGACT-SIGART, New York, NY, USA, 2004. ACM.