UNIVERSITY OF TWENTE.

MASTER THESIS

Provenance Aware Sensor Networks for Real-time Data Analysis

Reinier-Jan de Lange

Department of Computer Science

Supervisors:
Dr. Andreas Wombacher
Dr. Philipp Schneider

UNIVERSITY OF TWENTE.

eawag
aquatic research

Master Thesis in Computer Science

Provenance Aware Sensor Networks for Real-time Data Analysis

Enschede, March 14, 2010

Reinier-Jan de Lange, *University of Twente*

Abstract

In environmental science, sensors are most commonly used for forecasting or monitoring environmental processes. The observations are usually collected on a per-project basis, therefore these measurements are often duplicated between projects running at multiple organizations. A step in the right way to avoid this duplication is to introduce sensor networks, as they not only allow researchers to perform real-time data analysis, but enable sensor data sharing as well. However, in order to draw accurate conclusions or validate new models using this automatically collected data, metadata needs to be stored that gives meaning to the recorded observations. The sensor data generated by a sensor network depends on several influences, like the configuration and location of the sensors or the aggregations performed on the raw measurements. This kind of metadata is called *provenance data*, as the origins of the data are recorded. In this thesis, the requirements of a provenance aware sensor network are collected and a workflow is proposed for recording and querying sensor data and their provenance. A prototype system implementing the workflow shows that the proposed approach can effectively process sensor data from several sources, of which the use is justified in scientific research as the data provenance is known as well.

Preface

This publication is the result of a collaboration between the Computer Science (CS) department of the University of Twente in the Netherlands and the aquatic research institute EAWAG, in particular the department of Water Resources and Drinking Water (WUT, Wasser und Trinkwasser), in Switzerland. This collaboration has advantages for both parties:

- At the WUT department the use of sensors for measuring environmental changes is part of most running projects. Due to the day to day use of these sensors, requirements originate for retrieving, processing and querying the measurement results. By optimizing this data workflow, a lot of time and money can be saved. However, the institute is not specialized in this scientific field of work, therefore the collaboration with the CS department is very beneficial.
- The CS department at the university recently started investigating technical solutions for managing sensor data. There is still a lot of uncertainty on how streaming data should be handled. It is not a simple case of just persisting sensor data: sensor data in its raw form doesn't really reveal anything interesting. In most cases, some processing (joining, aggregating, etc.) needs to be done to make the data useful. Although these concepts are not new, it is hard for a CS engineer to tell which processing steps yield interesting results. By collecting requirements regarding sensor data from the researchers at the WUT department, they can get a much better understanding of the challenges involved.

The collaboration will become most apparent in this thesis by means of a case study in which the proposed approach was applied. At a couple of points, it was necessary to travel down to Zürich to gather requirements from the environmental researchers there. This has resulted in some interesting findings that will be presented throughout the chapters of this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

At EAWAG, sensor data is normally processed by first deploying a sensor, then coming back periodically to download the data and finally importing that data in a statistical analysis application like Matlab or R. By setting up a sensor network, data can be archived, searched for and processed online, allowing sensor data to be reused over multiple projects and providing real-time data analysis. When creating a historical archive, metadata becomes very important for the sensor data to make sense: if configuration changes are not recorded, analysis done on the sensor data can hardly be justified. To this end, this thesis will focus on the creation of a workflow that is able to record *provenance*, a type of metadata describing the origin of data.

1.1 Case study: the Distributed Temperature Sensor

A Distributed Temperature Sensor (DTS) is a next generation sensor for sensing temperature over long distances [5], which is described in further detail in this thesis. It is used by EAWAG to measure the temperature on several places in the side channels of the river Thur to find groundwater influxes: a colder measurement usually depicts a location of such an influx. The DTS will be used as a case study. It will show the whole workflow from the beginning to the end: Reading out the data, processing and persisting the results, recording the process and making it available to the end users. Moreover, the case should clarify which requirements exist and of how much importance they are in order to come up with a good system architecture.

1.2 Problem Description

Sensor data can come from several sources, which often only yield interesting results when combined with each other. This data doesn't necessarily have to come from sensors directly; it can also be recorded manually. For example, EAWAG has a great collection of manually sampled data at its proposal that is updated frequently. This is mostly chemical or ecological data that has been gathered by analysis of a sample (e.g. water, earth) taken from a certain location. They provide valuable additional information about the environment that is being monitored. The proposed workflow should be able to process data coming from these sources, while respecting the following conditions:

- The workflow should be able to process streaming data. Getting new sensor data as soon as possible can be very important, since new measurements may predict (natural) disasters. To be able to understand and possibly prevent these disasters, new data should be processed as soon as it comes in.
- The different data sources can be hosted by different organizations, therefore the infrastructure must be able to cross organizational boundaries.
- The workflow should support annotating sensor data on the fly. For example, in order to quickly detect 'interesting' measurements, it should be possible to directly classify and annotate the data. An added advantage is that users may even be alerted (by mail, SMS, etc.) upon receiving interesting values, which can be very useful if actions need to be undertaken fast.
- There should be a way to keep track of the sources and processes that were involved in producing a given set of results, which is needed to justify scientific findings based on those results.
- There should be a straightforward way for the users of the system to query the recorded data.

Distributed processing of streaming data is not a new topic, it has already been applied in some systems. To avoid reinventing the wheel, research should be done on finding already existing systems that aim at solving this problem and see to what extent those solutions can be reused.

1.3 Research Questions

This thesis will try to answer the following main research question:

> *How to apply real-time data analysis on streaming sensor data and manually sampled data considering data provenance?*

In order to answer this question, the following subquestions have been defined:

1. What are the requirements of such an infrastructure derived from existing systems and literature?
2. What is the conceptual model of the proposed infrastructure?
3. What is the supported query language as a user interface for the proposed infrastructure?
4. How does the architecture and implementation of the proposed infrastructure look like?
5. How is the proposed approach applicable to the Distributed Temperature Sensor use case from EAWAG?

The answers to these questions combined provide the answer to the main research question.

1.4 Project Outline

To answer the aforementioned research questions, this report will start in chapter 2 with a description of requirements that should be fulfilled by sensor middleware, followed by a domain analysis of publicly available existing sensor middleware solutions, and to which extent they fulfill these requirements. Next, a separate chapter, namely chapter 3, has been dedicated to provenance. It will describe what provenance metadata is, why it is essential in sensor data processing and how this can be recorded and retrieved. These first two chapters are meant to answer the first research question.

In order to answer the second research question, an architecture of a query processing system will be described in chapter 4. It will show an approach for query processing of sensor network data from multiple sources. The third research question will be treated in chapter 5, which will describe the requirements and the design of a query language for querying sensor data and metadata. The architecture and implementation of the proposed workflow will be described by means of the development of a prototype in chapter 4. This prototype will not implement all concepts, but is mainly meant to clarify and validate the workflow. Finally, the thesis will end with the final conclusions in chapter 7.

The DTS case study is the ‘red line’ throughout the thesis. Every chapter consists of background information or theory, which is applied in the case study. Its main purpose is to give the reader an example of an application of the theory, which may help to apply the theory in different situations. Moreover, the study will also be used to find unforeseen requirements of the system. As already stated in section 1.1, involvement of researchers that use sensors on a daily basis is essential to understand which functionality is currently missing.

Chapter 2

Sensor Middleware Survey

Sensor middleware deals with reading data coming from sensors or sensor networks, optionally aggregating or processing that data and storing the result in a database system. Several solutions have already been implemented and assessed [6], but each solution has its own set of features. This chapter will therefore start off with a comprehensive list of functionalities that sensor middleware should and could have, which can be used as reference for assessing existing solutions and implementing new ones. Next, a survey will be conducted on three relatively recent solutions, which will finally be used in the case study.

2.1 Functionality Description

A lot of aspects should be taken into account by sensor middleware. This section covers which functionalities sensor middleware should have or functionalities that might be nice to have. Requirements may differ greatly depending on the environment, the sensors used and how the sensor data is analyzed. The requirements listed here have partly been derived from literature on this topic [7, 8, 9] and partly by deriving requirements from documented features incorporated in existing middleware systems [6, 10, 11, 12, 3, 13, 14, 15].

2.1.1 General Functionality

Distribution In a sensor network, distributing services can be very important. A common reason is to distribute the load. One computer may be responsible for communicating with a sensor, while processing and presentation is done on other machines. Within sensor networks, often resource constrained devices are involved, such as small Netbooks or embedded systems, which can only handle small tasks and are unable to store large amounts of data. Finally, distributing services enables complex data processing by using sensor data from multiple data sources. These data sources can be sensors, but also manually recorded data or pre-processed sensor data stored on different servers [10].

Sensor specification To successfully record sensor data, a service should know the characteristics of a sensor. It should know the location of the sensor, the output structure and it should be able to uniquely identify the sensor to which incoming sensor data belongs. In some cases, a sensor specification can also be a composition of multiple sensors. Consider for example a weather station: it consists of multiple sensors to detect temperature (air, surface), wind speed, humidity and more.

Sensordata metadata recording To be able to understand what some data actually represents, you need some kind of context [9]. Regarding sensor data, this could for example be the configuration of the sensor (e.g. the orientation or angle) or aggregations/classifications performed on the data. Some of this metadata could be part of the sensor specification (for example the location of the sensor), but this is only the case for static (non-changing) variables.

Sensor access All sensor middleware needs some way to communicate with a sensor. Usually this will involve writing some code, since most sensors have their own communication protocol. There are also a lot of different ways to connect to sensors, for example through a serial connection, LAN, WAN or USB.

Sensor discovery Within a good sensor network, it should be possible to add or remove sensors without stopping the (whole) system, since stopping the system may imply loss of important sensor data.

Processing chains Multiple services can benefit from other services making some transformation on sensor data. This may be an aggregation, combination or classification of the data. As an example, an actuator service and a notification service could use a common classification service that finds interesting measurements.

Querying An important aspect of sensor middleware is how data can be queried. For some services, getting new sensor data as soon as possible is a must, while other services may be more interested in historical data. So, the system should provide a means of querying realtime as well as offline in an efficient way.

Presentation A good presentation layer is needed to efficiently show sensor data to the user. Since queries may take a long time to complete, calls to the system should be made asynchronously. Other important features of the presentation include keeping the user up to date with what the system is doing and presenting the data in a clear way by using charts or multiple page reports.

Service discovery Once a service is running, it should somehow notify people or other systems of that fact. When using Webservices, this can be achieved by using UDDI. Another approach that is becoming popular within the sensor network community is by publishing sensor data to globally available web services, for example SensorMap [16] or SensorBase [17].

Access control Most sensor data is not highly classified material, but there are exceptions, like images or video captions from (security) cameras. Therefore, there should be a way to secure that kind of data.

Communication protocol The communication protocol can be an important aspect with regard to sensor data. Since the amount of data itself can be quite large, the protocol should be kept simple to avoid getting a lot of overhead.

Optimization / Self organization To efficiently answer queries, caching can make a crucial difference. Complex, yet frequently requested queries should be cached to speed up the process. Of course, this only applies to historical data.

Fault tolerance Within a sensor network, a lot of things can go wrong. Sensors can be broken, down or just unreachable and connections are usually not very stable, resulting in failed communication or corrupt data. The system should take this knowledge into account and react appropriately when communicating with a sensor fails.

Standards Using standards is often a good idea, since other users of the system will know what to expect from it. Usually standards are very well documented and will thus enable users to interact with the system by just following the rules defined by the standard. It also simplifies integration with other systems that comply to the standard. The most important set of standards with regard to sensor networks come from the Sensor Web Enablement (SWE) initiative of the Open Geospatial Consortium (OGC) [8]. Standards are provided for the different parts making up a sensor network as well as the communication between these parts. The following specifications have been developed:

Observations & Measurements (O&M) Standard models and XML Schema for encoding observations and measurements from a sensor, both archived and real-time.

Sensor Model Language (SensorML) Standard models and XML Schema for describing sensors systems and processes associated with sensor observations; provides information needed for discovery of sensors, location of sensor observations, processing of low-level sensor observations, and listing of taskable properties.

Transducer Model Language (TransducerML or TML) The conceptual model and XML Schema for describing transducers (devices that that convert variations in a physical quantity, such as pressure or brightness, into an electrical signal) and supporting real-time streaming of data to and from sensor systems.

Sensor Observations Service (SOS) Standard web service interface for requesting, filtering, and retrieving observations and sensor system information. This is the intermediary between a client and an observation repository or near real-time sensor channel.

Sensor Planning Service (SPS) Standard web service interface for requesting user-driven acquisitions and observations. This is the intermediary between a client and a sensor collection management environment.

Sensor Alert Service (SAS) Standard web service interface for publishing and subscribing to alerts from sensors.

Web Notification Services (WNS) Standard web service interface for asynchronous delivery of messages or alerts from SAS and SPS web services and other elements of service workflows.

The choice whether or not to use these standards will often depend on the desired level of interoperability. Following the standards will simplify integration with unknown third party systems, but may make the system overly complex for the taks it is supposed to do.

Data digestion Raw sensor data often consists of a lot of redundant, useless information. A common strategy is to directly summarize (aggregate) the data before it is archived

[7]. This can be done by the collecting system itself or by a separate service. The drawback of the latter is that all data will need to be encoded and decoded before it is processed.

Alerters / Notifiers A frequent use case is that users monitoring something would like to be notified as soon as possible when something out of the ordinary occurs, for example a sudden drop in temperature or a sensor breaking down. This introduces the need for a notification service that is able to alert people wherever they are, for example by sending an SMS or e-mail.

Actuators Sometimes, it would be nice if the system would automatically react upon detecting anomalies. When a sensor is returning remarkable values, an actuator can react by changing the configuration of the sensor or by increasing the sampling rate in the system itself.

Handling of changing variables For archived sensor data to make sense, variables (like the configuration of a sensor) should either never change or changes should be recorded. When changes to the configuration occurs and that information is lost, sensor data with different configurations will get mixed together, causing the data and all analysis on that data to become incorrect and unprovable. Clearly, when using actuators this becomes a very important feature of the middleware.

Shared execution To allow multiple users to access the system at once, it should be multithreaded. It will seldom be the case that only one user is involved in analyzing the recorded data.

2.1.2 Windowing Functionality

When working with realtime sensor data, only an excerpt of a stream is of interest at any given time. This is the motivation for creating *window models*. A window always consists of two endpoints (moving or fixed) and a window size. Windows are either time based or count based [7]. This yields the following functionality:

Time based windows Time based windows define the window size in terms of time. It will only consist of data that falls within a certain time span, for example *one hour* or *ten minutes*. When newly added sensor data is extending the time span beyond the window size, the endpoints are moved, resulting in a *sliding window*.

Count based windows Count based windows define the window size as the number of tuples it contains. A window can for example have a size of 2000 tuples, meaning that there will never be more than 2000 tuples in it.

Fixed windows A fixed window is a window for which both endpoints are fixed: a window of a fixed point in time (e.g. from the year 2000 to 2001). Fixed windows are based on historical data.

Sliding windows A sliding window is a window in which both endpoints move, usually keeping the window size the same (e.g. a window of the last five hours).

Landmark windows A landmark window is a window in which only one endpoint changes. Often the left endpoint is fixed and the right one moves, causing the window to grow over time (e.g. a window of all data since midnight).

Update interval There are a couple of possibilities to update the window as new data comes in. One is to update after every tuple. However, sometimes it's better to batch process, meaning the window is only updated after receiving a fixed number of tuples or after a fixed amount of time. This results in *jumping windows*. When the interval is larger than the window size, the whole window is changed after every update. These kind of windows are called *tumbling windows*.

2.1.3 Storage

A stream management system usually consists of three types of data storage: temporary storage, summary storage and static storage [7]. The storage model chosen for each of these three types is important for a middleware system to work efficiently. Options include relational databases, flat files and storage in main memory (as objects or using an in-memory database).

Temporary working storage The temporary working storage is for storing window queries or caching. This data will usually be stored in-memory.

Summary storage Summary storage is for recording historical data, presumably aggregated in some way. Since this can be a large amount of data, this is usually stored on disk.

Static storage Fixed metadata about sensors, like its geographical location, manufacturer and output specification is all part of static storage. This data can usually be found in flat files or is stored in a relational database.

2.2 Software

Since the concept of sensor networks was invented, several solutions have been implemented. Some of these are domain specific prototypes, others are closed source and for in-house usage only and most of them have been discontinued after the project was finished. Just a few aim at creating a generic, publicly available solution. A full assessment will be made of three projects that are at the time of writing the most active projects that aim at providing a sensor network solution that can be applied in several environment developments, namely Global Sensor Networks (GSN) [1], the OGC SWE implementation by 52°North [2] and the Open SensorWeb Architecture (OSWA) [18]. Finally, at the end of this section a small summary of other solutions will be discussed; these solutions are only described shortly by summarizing available literature.

2.2.1 Global Sensor Networks (GSN)

2.2.1.1 About GSN

GSN is a sensor middleware which 'supports the flexible integration and discovery of sensor networks and sensor data, enables fast deployment and addition of new platforms, provides

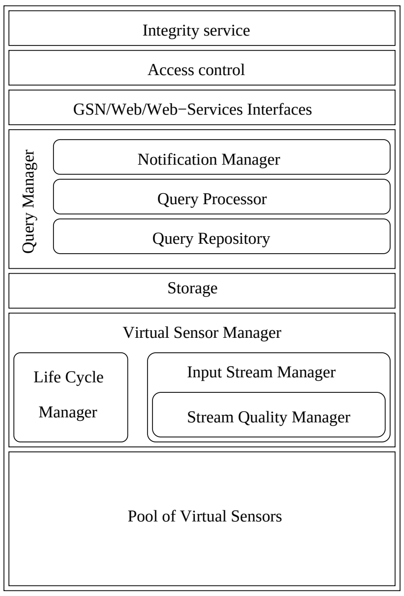

Figure 1: A GSN container [1]

distributed querying, filtering, and combination of sensor data, and supports the dynamic adaption of the system configuration during operation' [1]. GSN was started in November 2004 by Ali Salehi under supervision of Prof. Karl Aberer at LSIR Laboratory, Ecole Polytechnique Fédérale de Lausanne, Switzerland [19].

An instance of GSN is called a GSN container, of which an example can be seen in figure 1. The most important concept within a GSN container is the *virtual sensor* (VS). A VS publishes output from actual sensors or output generated by other virtual sensors, even if those are running in a different GSN container.

2.2.1.2 GSN Functionality

Table 1 assesses to which extent the functionalities described in section 2.1 are available in GSN.

Distribution	A GSN container (a running GSN instance, see figure 1) consists of *virtual sensors*. A virtual sensor can read and combine data from custom built sensor wrappers (java classes), but it can also read from other virtual sensors using XML-RPC. All data sources (so, wrappers or virtual sensors) have an output specification to accomplish this. Since XML-RPC is used, virtual sensors from one GSN container can communicate with virtual sensors running in other GSN containers.
Sensor specification	Virtual sensors have a virtualsensor XML definition, which contains its output specification, data sources used and static metadata, like the sensor's geographical location.

Sensordata metadata recording	GSN does not support annotation of sensordata, but static properties can be specified in the sensor specification.
Sensor access	Data from sensors is periodically collected by using *wrappers*. A wrapper needs to be able to connect to a sensor, read out the sensor data and encapsulate the retrieved data in a *StreamElement* object, which is a general object that GSN uses for communication internally. One stream element is basically one row in a relational database.
Sensor discovery	Virtual sensor XML definitions are stored in a designated directory, which is monitored for changes. Sensors can be added or removed by adding or removing definitions from this directory.
Processing chains	A virtual sensor can be used for data acquisition as well as data processing. Processing chains can therefore be created by coupling virtual sensors together.
Querying	GSN supports realtime querying and offline data retrieval. The system is built to directly process new stream elements, so batch querying can only be achieved by performing some caching in a custom virtual sensor.
Presentation	An XML-RPC servlet has been created to communicate with a GSN container. An example, AJAX enabled web interface is available that frequently checks for new data or sensors, allows a user to query and export offline data and point the location of all found sensors on Google Maps.
Service discovery	By adding a specific attribute to a virtual sensor definition, it can be announced to SensorMap.
Access control	Another optional attribute of a virtual sensor definition is a password. This enables simple authentication.
Communication protocol	Whenever data is being sent over the internet, XML-RPC is used for communication. This is the case for communication between virtual sensors and communication with the web interface.
Optimization / Self organization	Queries over data are not optimized and results are not cached; this is left to the database handling the queries.
Fault tolerance	To gracefully handle errors during data retrieval, GSN includes a Stream Quality Manager (SQM) [20]. If (virtual) sensors can't be reached, the system will keep retrying or disable the virtual sensor if necessary.

Standards	GSN does not use the approved OGC SWE standards like O&M or SensorML. Instead, it uses XML-RPC, in which sensor data is encoded to a self-chosen markup. There have also been efforts in creating a REST interface.
Data digestion	New data is directly stored in the database. The size (amount of records) in the database can be managed. Directly aggregating the data before it is inserted into a database is however not supported and should be done by creating a custom StreamExporter virtual sensor.
Alerters / Notifiers	A Notification Manager is included, which can notify users via e-mail or SMS.
Actuators	There is no support for actuators, but the Notification manager can easily be extended to support this functionality.
Handling of changing variables	Changes in sensor configurations are not recorded; it is expected that the configuration never changes.
Shared execution	The provided servlet is able to handle multiple requests at the same time.
Count / time based windows	If a time window is defined in the virtual sensor definition, GSN instantiates 'time based sliding handlers'. Count based windows are also supported, these are handled by 'tuple based sliding handlers'.
Window specification	Users can request realtime or historical data and therefore it supports fixed, sliding and landmark windows.
Update interval	Windows have a 'sliding value' (default 1), meaning that the window will update after receiving that number of new elements. For example, when the sliding value is three, the window will be updated after three new tuples have been recorded.
Storage	The database that is specified in the main configuration is used for temporary storage and historical data. By default, GSN only supports MySQL, MSSQL and H2 through JDBC. However, if a StreamExporter virtual sensor is used, aggregations on the sensor data can be directly exported to any JDBC compatible database. Static information about the sensor itself is stored within the virtual sensor definition.

Table 1: GSN functionality

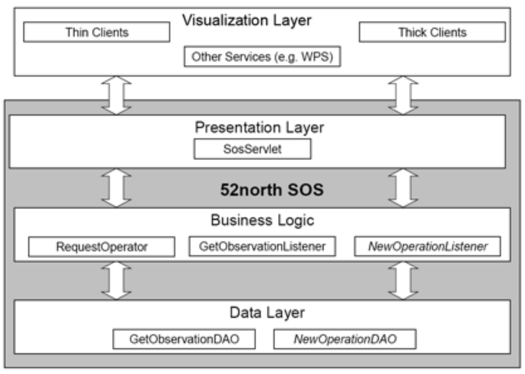

Figure 2: 52°North SOS architecture [2]

2.2.2 52°North

2.2.2.1 About 52°North

52°North [2] is an international research and development company whose mission is to promote the conception, development and application of free open source geo-software for research, education, training and practical use. The 52°North Sensor Web Community focuses on the development of a broad range of services and encoding implementations related to OGC's Sensor Web Enablement, as well as multi functional clients to access all of these services.

Five different services are offered by 52°North, namely the Sensor Observation Service (SOS), Sensor Alerts Service (SAS), Sensor Planning Service (SPS), Web Notification Framework (WNS) and the OX Framework (OXF). The most important service is the SOS, which provides observations for the other services. Figure 2 shows the architecture of the SOS. The SOS is very modular to allow developers to adapt the system to their own needs: the data access layer can be extended by a new DAO implementation which connects to an already existing sensor database, new SOS operations can be added by adding new request listeners and communication with clients can be altered by changing/replacing the SOS servlet.

2.2.2.2 52°North Functionality

Table 2 assesses to which extent the functionalities described in section 2.1 are available in 52°North.

Distribution	Every part of the 52°North implementation of OGC SWE standards is a separate webservice. These webservices can communicate with each other, enabling distribution.
Sensor specification	Sensors need to be specified in OGC SensorML [8]. Using SensorML, compositions of sensors can be modelled, for example an entire weather station.

Sensordata metadata recording	The OGC SWE standards do not mention sensordata metadata. 52°North's implementation does therefore not record sensor metadata. There has been made a prototype though, which adds RDF data to O&M encoded sensor data (The Semantic Sensor Web, see [9]).
Sensor access	The SOS itself does not access sensors. Data should be acquired by using a different service. 52°North provides the SOSFeeder service for this, which inserts data directly into the SOS database. Data acquisition services that cannot access the database can also add observations using the InsertObservation operation of the SOS.
Sensor discovery	Descriptions of sensors are initially read from a local directory of the SOS. When an SOS is running, new sensors can be registered using the RegisterSensor operation, a feature that is part of the OGC SOS Transactional Profile (see [21]).
Processing chains	The OGC SWE services are not processing services. The SOS only provides filtering sensor data; it does not provide aggregations or other processing. Processing should be done by custom made services that communicate with the SOS.
Querying	As the SOS does not directly access sensors, there needs to be an intermediary database. This provides 'near real-time' observation of sensor data [8]. Realtime querying is not supported though, this is defined in the OGC TransducerML (TML) standard which is not (yet) implemented by 52°North.
Presentation	52°North provides the OX Framework, which is a framework that can communicate with the SOS. A thin (web) and a thick (Java Swing) client have been created to demonstrate the functionality of the framework.
Service discovery	Running instances are not automatically published. Within the OX Framework, new services need to be added manually.
Access control	The SOS service itself does not provide access control. Securing the service should be done by running it within a closed network. Of course, access to the web client can be restricted by basic HTTP authentication.
Communication protocol	The 52°North services do not communicate using SOAP. Instead, observation requests are made by sending HTTP GET/POST requests. Responses are encoded in OGC's standards (SensorML, O&M).

Optimization / Self organization	The SOS provides functionality for requesting descriptions and capabilities of sensors. To efficiently respond to that kind of queries, the sensor information described in SensorML is cached. Observation query results are however just passed to the database.
Fault tolerance	Errors during communication do not cause the services to quit; they are properly caught and logged. Most communication will happen between the SOS and the database, so this strategy is usually fine as such a connection is quite stable. The SOS Feeder framework is a difference case however, but no action is taken there either when communication with the sensor fails.
Standards	52°North tries to implement the standards defined by the OGC. So far, they support the SOS Core and Transactional Profile, SAS, SPS, WNS, SensorML and Observations and Measurements (O&M) standards.
Data digestion	All sensor data should first be stored in the SOS database. Further processing must be done by creating services that request the sensor data from the SOS.
Alerters / Notifiers	There have been built a couple of services to alert or notify users, namely the Sensor Alert Service and the Web Notification System. The WNS is able to send notifications asynchronously to several devices. It can use e-mail and SMS and can communicate with phones and faxes. The SAS can be used for online notifications. Users should join a MultiUserChat (MUC) and the SAS will send alerts to all registered users.
Actuators	A client that is registered to the SAS does not necessarily have to be a user. An actuator service that can join a MUC can be created, which can then quickly act upon receiving certain alerts.
Handling of changing variables	The OGC SOS specification describes a dimension of sensor data called its *feature of interest* (FOI). The FOI provides information about the environment of the sensor, for example a geometry specifying the field that the sensor is sensing. Changing system variables should be handled by adding/updating FOIs.
Shared execution	Multiple clients can use the 52°North services at the same time.
Count / time based windows	Since TML is not supported yet, realtime observation is not possible and therefore sliding windows are not supported. Only fixed selections can be done on the SOS sensor data. Count based windows canot be specified.

Window specification	When requesting for data, an event time (period) can be specified for which it should return results.
Update interval	Update intervals don't apply in this case, since sliding windows are not supported.
Storage	The system requires a PostgreSQL database with PostGIS extension. It uses this for storing FOIs. Static sensor information is stored in SensorML files located in a designated directory of the SOS.

Table 2: 52°North functionality

2.2.3 Open SensorWeb Architecture (OSWA)

2.2.3.1 About OSWA

The OSWA [18] is an initiative of the Grid Computing and Distributed Systems (GRIDS) Laboratory of the University of Melbourne. OSWA consists of several implementations of the OGC SWE standards, like 52°North. OSWA is not as far as 52°North, since only the SOS Core Profile, SPS and WNS are implemented. Figure 3 shows the global architecture. The SOS has been named the Sensor Collection Service (SCS), which is the old name the OGC had given to the service. The main differences between the OSWA and 52°North is that the OSWA services are pure SOAP web services and the SCS is able to provide realtime observations, since the SCS is also responsible for data acquisition.

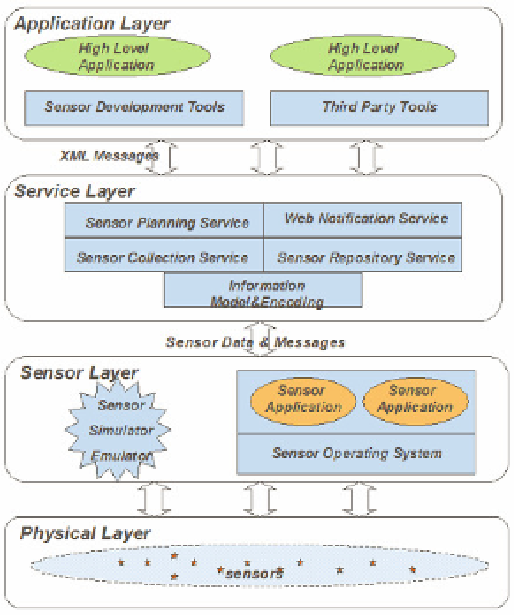

Figure 3: OSWA SCS architecture [3]

2.2.3.2 OSWA Functionality

Table 3 assesses to which extent the functionalities described in section 2.1 are available in OSWA.

Distribution	Just like 52°North's software, every part of the OSWA implementation of OGC SWE standards is a separate webservice. The main difference between the two is that the OSWA services are pure SOAP services.
Sensor specification	A separate database is used for storing SensorML information of sensors. It needs to be filled in manually. This seems a little cumbersome, but the system can create appropriate SensorML XML from this data, so it is probably work in progress.
Sensordata metadata recording	There is no option to record sensordata metadata, but the prototype described in [9] can also be applied in this system.
Sensor access	The SCS uses Proxy instances to retrieve data from several sources. In turn, every sensor Proxy uses Connectors, which are used for communication with the sensor.
Sensor discovery	Sensors and services need to be configured in *properties* files. Registering or unregistering sensors is not possible.
Processing chains	Processing services are not provided. Sensor data must be processed by custom services communicating with the SCS.
Querying	OSWA provides realtime and offline querying, depending on the proxy used. It is possible to do batch querying, because the system supports the O&M complex type *ObservationCollection*.
Presentation	The system exposes its functionality using Axis WS. A frontend is not provided, but by using the WSDL generated by Axis, an application can easily be built on top of it.
Service discovery	Since OSWA uses Axis webservices, the services can be registered to a UDDI service.
Access control	Axis provides access control for web services, so this can be used to secure the exposed services.
Communication protocol	The system sends SOAP messages in which OGC SWE standards, SensorML and O&M, are encoded. Requests should also be made using SOAP encoded messages.
Optimization / Self organization	A student project has resulted in a caching mechanism for OSWA, which is able to cache query results. This caching mechanism has not been properly tested yet though, therefore it is turned off by default.

Fault tolerance	There is not a lot of error correction; when something fails it is only reported in the error log.
Standards	The SOS Core Profile, SPS, WNS, SensorML and Observations and Measurements (O&M) standards are supported by OSWA.
Data digestion	In contrast to 52°North, the SCS does not rely on a database. To save sensor data, a Sensor Repository Service (SRS) has been created. The SRS does not aggregate data, but just saves the data as is. To get observations from the SRS, a special Proxy has been created in the SCS which can communicate with the SRS database.
Alerters / Notifiers	The system contains a WNS to send notifications (e-mail, SMS) to registered users.
Actuators	Actuators are not yet supported, but support for actuators is one of the objectives of the project.
Handling of changing variables	The system does not expect variables to be dynamic, therefore these changes cannot be recorded.
Shared execution	Requests are handled by Axis, which ensures multiple users can use the service at the same time.
Count / time based windows	OSWA enables the user to specify time based windows.
Window specification	The SCS can return observations for a certain period, thus enabling fixed and landmark windows. It is however not possible to define sliding windows, since a window cannot move.
Update interval	When an observation period has been specified, the window is updated after every observation. It should be noted that an observation can possibly be an observation collection.
Storage	The SCS uses the WebService Resource Framework (WSRF) to temporarily store window data. For historical data, which is recorded by the SRS, any JDBC enabled database can be used. Sensor information is stored in a database as well.

Table 3: OSWA functionality

2.2.4 Other solutions

2.2.4.1 System S

System S is a large-scale, distributed data stream processing middleware. It can read data from several data sources and process that data to answer queries inserted by users. The core of the system consists of chained Processing Elements (PE's) that can perform several basic operations. PE's can be created by programming, by using a declarative language (SPADE) or by using a graphical tool called INQ [22]. The query processing approach that will be described in chapter 4 has many resemblances with the processing approach used in System S, but the difference is that System S can consist of multiple clusters. Like GSN, System S applications can communicate with other running System S applications. The system fulfills many of the defined requirements, but a big drawback is that the communication protocol is unknown. As it is a proprietary system, this protocol is probably proprietary as well, in which case it is unlikely that the system can cross organizational boundaries as easy as web services can.

2.2.4.2 SONGS

Service Oriented Network proGramming of Sensors (SONGS) is an implementation of semantic service frameworks for sensor infrastructures [23, 24, 6]. Semantic services contain *inference units*, which enable semantic queries. The idea of an inference unit is that it 'wraps' one or more sensors into a meaningful object. By specifying the region that is covered by these sensors, the inference unit to which a sensor belongs can be inferred. Moreover, when defining an inference unit, the input sensors and the output can be semantically annotated as well. As an example, consider an inference unit called *VehicleDetectionUnit*. In SONGS, this can be specified as follows [24]:

```
inference(VehicleDetectionUnit, needs(
sensor(magnetometer, R) ), creates(
stream(X),
isa(X, vehicle),
property(X,T,time),
property(X,R,region) ) ).
```

In this snippet, *R* is a given region. The VehicleDetection unit consists of one sensor that covers the region *R*, which is semantically annotated as being a 'magnetometer'. The output is an event stream with the time and location in which vehicles are detected. The advantage if this approach is that a user can specify queries that are semantically meaningful. Consider for example the following query, which tells the system to only return car events in a certain region [24]:

```
stream (X), isa (X, car), property (X, [[10,0,0],[30,20,20]], region).
```

While the publications don't give much details about the implementation of the system, the approach for enabling semantic query definitions is very interesting. It is one of the first systems to recognize the importance of understanding the meaning of the data that has been processed.

2.2.4.3 IrisNet

The idea behind the Internet-scale Resource-Intensive Sensor Network Service (IrisNet) is to create a worldwide sensor web to which any sensor can be connected. Interested and authorized systems can collect data from specific sensors, which can then be used by people in a variety if ways. This can be small things like notifying a user when to head to the bus stop, but it can theoretically even be used by organizations for perimeter defense (security). The network consists of Sensing Agents (SA's) and Organizing Agents (OA's), of which the first are the services for acquiring data from sensors and the latter are for persisting the collected data in a database. The SA's contain blocks of code called *senselets*, which send the collected data to nearby OA's that are authorized to record the data. Querying can be done by defining XPath queries, as the data is saved as XML data. [25, 6]

To demonstrate the architecture described, a couple of prototypes have been built that were able to monitor parking lots and collect images of a coastal line. The prototypes showed that the architecture works, but that it was mainly meant to focus on the technical challenges. A lot of issues still needed to be addressed (mostly policy and privacy issues) [25], but not much progress has gone into the project ever since.

2.3 Middleware Functionality Summary Table

Table 4 summarizes the three projects assessed in section 2.2. Clearly, GSN is the most complete software package of the three. However, this does not mean that it is always the best choice; often this will depend on the current situation and the user requirements. A reason to choose for 52°North is its good compliance to the OGC SWE standards, which becomes very important if the network needs to be linked to other networks or third party tools. And why would you choose to support realtime data analysis if it is not required by the users of the system? Finally, there is often already a system in place. When this is the case, replacing it with a completely different system can be very time consuming and therefore costly.

General Functionality				
Functionality	**Question**	**GSN**	**52°North**	**OSWA**
Distribution	Can services be distributed?	Yes: GSN containers	Each main functionality (inserting data, reading data, notifications) is a service on its own	Each main functionality (inserting data, reading data, notifications) is a service on its own
Sensor specification	How are sensors defined?	*Virtualsensor* definition. A unique name, data sources and the sensor's position can be specified	A sensor can be completely modelled in XML using OGC's SensorML standard	A database must be filled with SensorML information
Sensordata metadata recording	Can metadata over the sensor data be recorded?	Only static metadata	Only static metadata	Only static metadata

Continued on next page...

Table 4 – Continued

Functionality	Question	GSN	52°North	OSWA
Sensor access	How are sensors accessed?	Wrapper classes	A SOSFeeder service must be created	Creating an implementation of the SensorConnector class, which is part of the Sensor Collection Service (SCS)
Sensor discovery	How does the system find new sensors?	The virtual-sensors directory is periodically scanned for new definitions	The RegisterSensor operation should be called	No sensor discovery
Processing chains	Does the middleware provide chaining processing services?	Virtual sensors can be chained. A virtual sensor can also be a processor.	Services can use other services	Services can use other services
Querying	What are the querying possibilities? Is it possible to query data in realtime / batch or offline?	Realtime and offline. Realtime querying can be event or poll based. When polling, the sampling interval can be set per virtual sensor	All data needs to be inserted in a database first	Depending on the implementation of Proxy, this can be done in realtime or offline.
Presentation	How does the system present the data (web interface, desktop application, WS)?	Web interface or WS (XML-RPC)	Web interface that communicates with a servlet	Axis WS
Service discovery	Is there a possibility to discover running instances?	SensorMap	No	UDDI
Access control	Can user access be controlled, for example by basic HTTP authentication or ACLs?	Simple password check	No	WS authentication (using Axis)
Communication protocol	How does communication happen between the system and clients?	XML-RPC	Http GET/POST requests result in OGC SOS XML responses [26]	SOAP

Continued on next page...

Table 4 – Continued

Functionality	Question	GSN	52°North	OSWA
Optimization / self organization	Does the system use query optimization or Caching?	Actions are not taken when queries get slow	Information about sensor capabilities is cached, all other data is just queried from the database	Queried observations are cached
Fault tolerance	How does the system respond to errors, for example during communication?	A Stream Quality Manager (SQM) ensures faults are gracefully handled	Exceptions are properly caught but not acted upon	Connections are properly closed when communication fails, but exceptions are usually thrown
Standards	Are there any standards used?	Only XML-RPC for communication between containers and the frontend	OGC SensorML, O&M, SOS and SPS [26]	OGC SensorML, O&M and SOAP
Data digestion	Can the system export data to summary structures?	Should be implemented by using a StreamExporter VirtualSensor	All data is inserted in one database. Further processing should be done by other services	The Sensor Repository Service (SRS) is responsible for storing data retrieved by the SCS.
Alerters / Notifiers	Can the middleware notify users quickly to unusual data values?	Notification Manager. It can be used to notify via mail or SMS	The Sensor Alert Service (SAS) and the Web Notification System (WNS). Can communicate with an SOS and alert in several ways (by default XMPP is used)	Web Notification System (WNS).
Actuators	Can the middleware react quickly to unusual data values?	No, but the notification service could be extended for this	An actuator can use the Sensor Alert Service (SAS)	No

Continued on next page...

Table 4 – Continued

Functionality	Question	GSN	52°North	OSWA
Handling of changing system variables during execution	Long-running queries may encounter changes in system conditions throughout their execution lifetimes. Can the middleware record those changes?	None. Changes in sensor configurations are not recorded or acted upon	Observations are linked to *features-of-interests*, providing information about the sensor environment. Changing variables should be reflected by inserting new features of interest.	None
Shared execution of many continuous queries	Requests should be non-blocking (multithreaded) to provide a multi-user environment. Does the system support this?	Supported.	Supported	Supported
Windowing Functionality				
Time based windows	Windows defined in terms of a time interval	Yes	Yes	Yes
Count-based windows	Windows defined in terms of the number of tuples	Yes	No	No
Fixed windows	Two fixed end-points	Yes	Yes	Yes
Sliding windows	Two sliding end-points	Yes	N/A	No
Landmark Windows	One moving end-point	Yes	N/A	Yes
Update interval	Does the system update after every incoming tuple, does it batch process, or both?	By default after every tuple, but depends on the 'sliding value'	N/A	After every tuple
Storage				
Temporary working storage	Where does the system store data for window queries?	Any database that is accessible through JDBC.	N/A	WSRF

Continued on next page...

Table 4 – Continued

Functionality	Question	GSN	52°North	OSWA
Summary storage	Where can the system store data for stream synopses (digests)	MySQL, MSSQL or H2 through JDBC, or custom made by using the StreamExporter VirtualSensor	PostgreSQL with GIS extension (for specifying features-of-interest)	Any database accessible through JDBC
Static storage	How is information about the sensor itself stored?	XML file (VirtualSensor definitions)	SensorML file	SensorML data in a database

Table 4: Middleware survey summary table

2.4 Case study: Middleware

To choose the middleware to use for the case study, it is needed to understand more about the domain and the people involved (the *stakeholders*). They will play a big role in finding user requirements.

2.4.1 Domain

One of the projects in which the WUT department is involved is the RECORD (REstored CORridor Dynamics) project. The objective of the RECORD project is "to increase mechanistic understanding of coupled hydrological and ecological processes in near-river corridors.". [27]. For this purpose, a (channelized) section of the river Thur is one of the sites where experiments are conducted. One measurement tool is the DTS, the sensor that was described in the introduction. The RECORD project itself is also part of the Swiss Experiment (SwissEx). The primary objective of the Swiss Experiment is to "enable effective real-time environmental monitoring through wireless sensor networks and a modern, generic cyber-infrastructure (...) to work efficiently and collaboratively in finding the key mechanisms in the triggering of natural hazards and efficiently distribute the information to increase public awareness" [28]. This research is a contribution to accomplishing that goal, therefore it is good to understand more about this project. Figure 4 shows the project and the stakeholders of the Swiss Experiment. It consists of about twelve projects, but only SensorScope and PermaSense are explicitly added, since these projects already have an automatized way of collecting sensor data.

Most user requirements will come from the researchers that are actually using sensors for their experiments; the funding partners and the government are mainly interested in the research outcomes. For the RECORD project, these are researchers from ETHZ and EAWAG.

2.4.2 Current situation

One objective of SwissEx is to enable real-time monitoring [28], but most sensor data is still collected in the old-fashioned way: deploy a sensor, let it record data to a datalogger and come back once in a while to download the data. Knowing that there are hundreds of

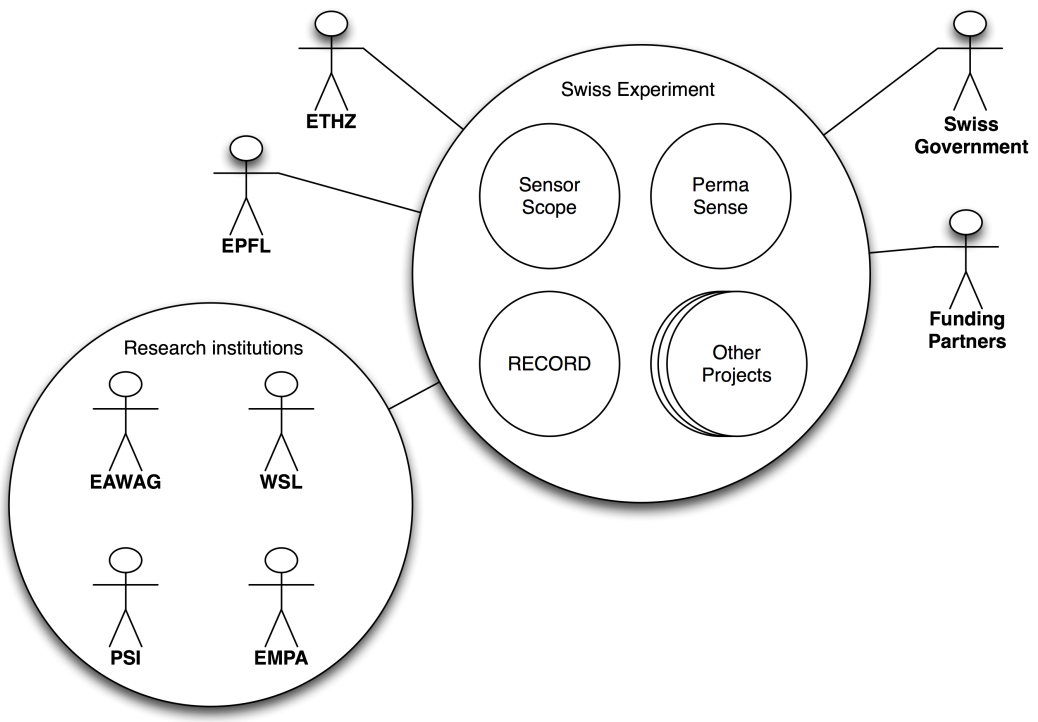

Figure 4: RECORD project domain

deployed sensors, such an undertaking can take some days. Moreover, some sensors also need to be checked frequently in order for them to work correctly. And finally, this way of working also causes a lot of observation duplication, since measurements are done over and over again for every project. SwissEx itself is already a big step in the right direction regarding this last point, since the research institutions depicted in figure 4 used to work independently of each other.

The previous paragraph clearly shows the usefulness of direct availability of sensor data:

- Researchers don't have to visit every single sensor for getting sensor data, which is a very time-consuming job
- Researchers can be notified by services that can detect when sensors need maintenance
- Data can be queried or searched for from a central place, thus avoiding observation duplication

It must be noted that some projects do already provide real-time monitoring, namely SensorScope and PermaSense. SensorScope is a real-time monitoring system itself. It is a Wireless Sensor Network (WSN) that supports *multi-hop routing* to be able to reach distant sensors [29]. PermaSense is a project that investigates permafrost and has therefore deployed several sensors at a height of 3500m. GSN is used for recording and analyzing the sensor data, which initially is collected using a WSN as well [30] [31].

The DTS described in the introduction (section 1.1) is however one of the many sensors that is still used in the traditional way and is used for lateral profiling (measurement over

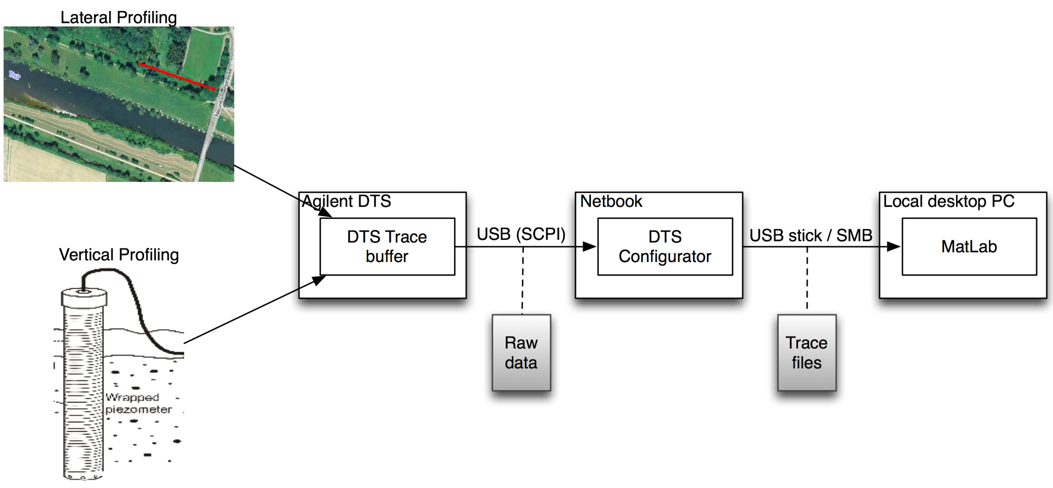

Figure 5: Current DTS data flow

a long distance) as well as vertical profiling (distributed measurement over several depth levels in the soil). An abstract representation of the current data flow is given in figure 5. A small computer (Netbook) communicates with the DTS over USB. The DTS Configurator is a DTS administration application supplied by the sensor manufacturer Agilent and is used to configure the DTS and export new measurements to 'trace files', which have a format that resembles CSV. Every now and then, new trace files are downloaded by connecting a USB stick to the Netbook or by joining the local network to which the computer is attached. The downloaded data is taken back to the office, after which a MATLAB script is used to parse the trace files and generate plots for analysis and use in publications.

2.4.3 User requirements

Clearly, a lot of time and money can be saved by switching to a real-time processing solution. However, this change does yield new requirements. They were gathered by speaking with the people using the DTS and looking into the publications that have followed from their research and are listed in table 5. Real-time processing is clearly a new feature, but the others mainly follow from the different way of working; in the current situation, the researcher himself would manually extract the relevant data from the set of measurement results and write down the sensor configuration for use in a publication.

2.4.4 Conclusion

For reading out DTS data, GSN seems to be a good start. It supports most functionalities described at the start of this chapter and is already used in some research projects. It supports data selection and real-time processing, which are two of the requirements. Interoperability may be an issue because GSN does not follow the SWE specifications, but since the objective is to share data with members of the Swiss Experiment this is not a problem, as they do not have any SWE enabled systems anyway. Adaptations still need to be made however: querying data from GSN is not a trivial task and there is no support yet for recording and querying metadata. This will have the main focus during the rest of this thesis.

Table 5 DTS retrieval user requirements

RE1: Selection	The system should be able to make selections on the data, such as time intervals, position intervals and temperature intervals. This is a high priority requirement for lateral profiling, where analysis of measurements in a given area at a certain time is common.
RE2: Real-time processing	The system must enable the user to process data in real-time. Another application of the DTS is to monitor the temperature in sewers for detecting sewer leakage. A sudden change in temperature may imply that a pipe is leaking, in which case it should be reported as soon as possible.
RE3: Easy querying	There must be a straightforward way to query the network. Most researchers have only little knowledge of computer science, therefore a natural way of requesting the data is required.
RE4: Metadata	The system should be able to store static metadata (fixed sensor information) as well as variable metadata like the sensor configuration and other annotations. In order to understand the values that the system returns, it should be possible to record and query metadata like the sensor configuration/calibration and the processing that has been performed.

Chapter 3

Provenance

The provenance of a painting is a record of the derivation and passage of an item through its various owners. It is of great importance to determine the value of the painting, which is not only defined by its painter, but also by the owners of the painting, the price that has been paid by each owner and what caused the painting to be in its current state, for example by acts of renovation. In this chapter, we will see that this concept can also be used in the field of computer science to trace back data to its source.

3.1 Basic Types of Provenance

Within computer science, there are two basic types of provenance: process provenance and data provenance. Process provenance, which is also known as *transformation provenance* [32], describes the provenance of a data item as the processes that lead to its creation. Recording this provenance can be automatic, manual or semi-automatic and can involve user interaction. Data provenance or *source provenance* [32] is information about the data that was involved in the creation of a certain piece of data. This information can contain references to the original source of the data, input data that has contributed to the result or even data that did not change the result, but were just involved in the creation of the data item.

This chapter will mostly focus on process provenance, which is the type used by the Open Provenance Model (OPM) described in section 3.3. The main argument for focussing on process provenance is that recording data provenance for streaming data should intuitively generate a huge, unmaintainable amount of provenance data, without providing remarkable benefits over just recording the process provenance. As will be explained in section 3.4, *reproducibility* of a result is a main objective when recording provenance for scientific research, which can be achieved best by recording process provenance.

3.2 The Provenance Challenge

During a discussion on provenance standardization at the International Provenance and Annotation Workshop, the participants concluded that they needed to get an understanding somehow of the different representations used for provenance: its common aspects, and the reasons for its differences [33]. As a result, the community agreed that a “Provenance

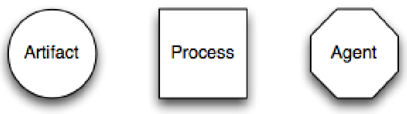

Figure 6: Graphical representation of OPM entities

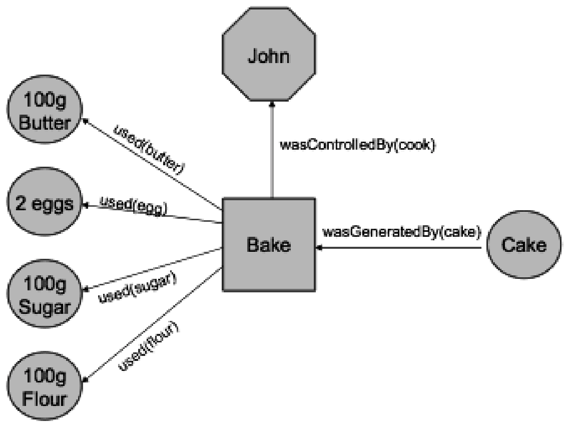

Figure 7: The provenance of baking a cake [4]

Challenge" should be set to compare and understand existing approaches. At the time of writing, two of these provenance challenges were already held and a third was in progress. The first two challenges resulted in the Open Provenance Model v1.01, which was evaluated in the third challenge [34]. The rest of this chapter will describe the OPM, how it can be used for streaming data and finally an architecture will be presented for web service based provenance recording and querying.

3.3 The Open Provenance Model (OPM)

The OPM is a technology-agnostic provenance model that is aimed at enabling systems to exchange provenance information. It has been designed to represent any kind of provenance, even if it has not been produced by computer systems. The model defines a causal graph that consists of Artifacts, Processes, Agents and the causal relationships between these entities [4]. Figure 6 shows how the different types are represented. An example of representing the provenance of baking a cake by using the OPM can be found in figure 7. It consists of the following parts:

- Five Artifacts, namely the cake and the ingredients it is made of. Artifacts are represented as circles.
- A Process, the 'Bake' process. Processes are always represented as rectangles.
- An Agent named John, represented by a hexagon.

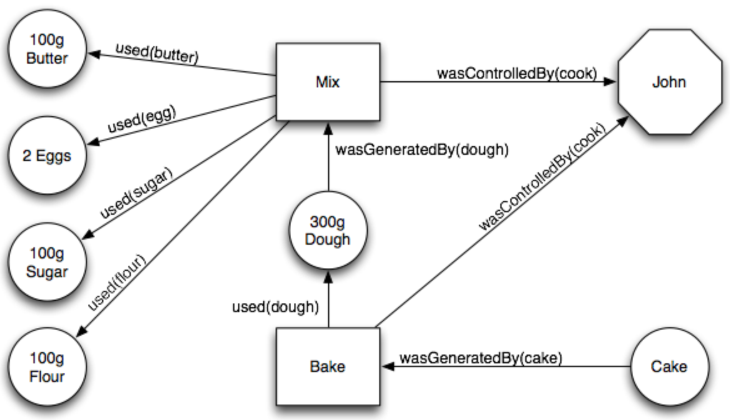

Figure 8: Overlapping account of the provenance model of baking a cake

- Six causal relationships between the before mentioned entities, represented by arrows.

The meaning of a causal relation depends on the entities that it connects. This is the case for every combination of two entities connected by a relation. Causal relations can also contain an optional *role*, which designates an artifact's or agent's function in a process.

The OPM has explicitly been designed to be able to specify provenance for *past* events. It does not provide standards for recording or querying provenance, nor does it specify how provenance models should be serialized by computer systems. Version 1.01 of the OPM does not yet incorporate details about the question how provenance of streaming (sensor) data should be recorded, although there are some footnotes suggesting this will be added in future versions. These limitations make the integration of OPM provenance models into a sensor network a real challenge and the final system presented in this thesis will hopefully clarify to what extent the OPM is suitable for recording provenance of streaming sensor data.

3.4 Provenance Recording Objectives

The big question in recording provenance for sensor data is what level of detail should actually be recorded. The simplest form of provenance in this case is to only record the information that is externally observable, which is basically a list of processes that have performed some function on the data. However, we can also 'zoom in' on every process and record the provenance of what is actually happening inside of each process. This extra provenance refinements are called *overlapping accounts* in the OPM.

Figure 8 is an overlapping account of the model depicted in figure 7, because they share some common nodes, namely the ingredient artifacts, the Cake artifact and the cook John. It is a refinement of the baking process.

Overlapping accounts can also be made when recording provenance for sensor data, which would be a refinement of what a node in the sensor network is actually doing with the sensor data. For example, when there is a node in the network that calculates average values per hour, this node can be described by just one process called 'CalculateAvgPerHour' or by creating an overlapping account refining that process. This refinement could be interesting if the node behaves differently under different circumstances.

Within scientific research, the most important reaosn for recording provenance is reproducibility [35]. Sensor data used in journals and other publications should be reproducible to justify the conclusions that have been made. Therefore, the level of detail of the provenance of sensor data should be as high as is needed to reproduce the result. Of course, it should be taken into account that some information will be lost to limit the amount of provenance data that is generated. Getting back to the example of a node generating hourly averages, one could require that the provenance graph should contain a direct reference to the sensor data used for generating a particular tuple. This would mean that the provenance graph of every tuple is different and would therefore result in a massive amount of provenance data. However, this could be limited easily if instead the choice is made to see all input of a node as being one input view and thus just one artifact. The actual data that is aggregated will be unknown, but one is probably still able to reproduce a result by using the timestamp of the generated tuple.

3.5 Provenance over Time

Usually, a workflow is not continuous, but will occur every now and then at arbitrary times. It is therefore a logical choice to record the provenance of every time that workflow yields a result. Taking the OPM example of the provenance of a cake, we observe that this workflow is finite and occurs in a clearly distinguishable time period. This is not feasible for continuous workflows; new results can come in every second and storing the whole provenance graph for every tuple would yield a massive amount of provenance data in short time.

An important clue in finding a solution to this problem is that, regardless of the continuity of a workflow, a lot of duplication is introduced when the execution of the workflow is fixed. The process of making a cake is defined by a recipe, therefore the provenance graph of one cake will often be the same as another cake. Essentially, the recipe of making a cake can be seen as a stateless workflow: it does not change over time. This is the reason for the fact that it can be represented by a timeless provenance model.

3.5.1 Workflow State

As explained in the previous section, the provenance of a workflow is related to the *state* of the workflow. A workflow state is defined as follows.

Definition 1 (Workflow State) *A workflow state is a collection of artifacts, processes and agents with causal relations that hold for a given time interval. A workflow changes state when any of the entities or relations in the workflow change state.*

Definition 2 (Stateless Workflow) *A stateless workflow is a workflow that never changes state over time. It can also be seen as a fixed state workflow: a workflow that only has one state and never transitions. Its time interval is unbounded. Conclusively, the entities and causal relations in a stateless workflow never change either.*

Deciding that a workflow is stateless is not as obvious as it may seem and can depend on the level of detail of the provenance model. This can best be illustrated by going back

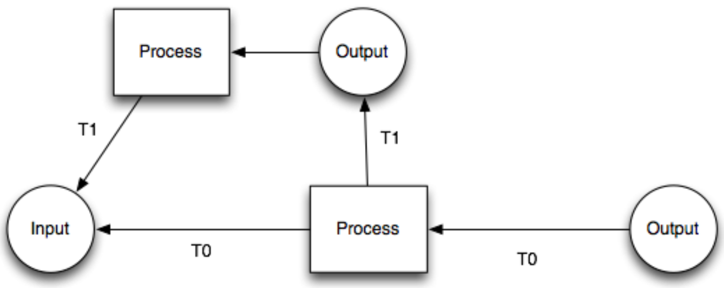

Figure 9: Example provenance graph using timed annotations

to the provenance of baking a cake. If we assume that the recipe of baking a cake is fixed, we could say that the workflow is stateless. However, the model depicted in figure 7 also incorporates the cook John as being the Agent of the baking process. Can it be assumed that John will always be the cook? Could it also be Alice? If that is the case, we can clearly see that the workflow is not stateless after all. However, if we decide to lose some detail, for example by calling the Agent 'Employee' or by removing the whole Agent, the provenance model can become stateless again.

The problem that remains is that there is no standard way yet for relating provenance to a particular state of a workflow. How can the state of a workflow be identified? And how do we know that a particular result was generated while the workflow was in a certain state? The two following subsections will discuss two possible solutions to solve this problem.

3.5.2 Partial Provenance Graph Updates Using Timed Annotations

If the state of a workflow changes, a new provenance graph could be created. However, if it is known which process in the workflow has changed, it is also possible to update the old graph using timed annotations on the causal links coming in and going out of that process. This is based on the fact that a workflow state transition occurrence means that one or more of its entities or causal relations has transitioned, as described by definition 1. Optionally, the provenance of the process itself can then be specified by creating an overlapping account.

Figure 9 is a small example explaining this concept. The original provenance graph, time annotated with $T0$, shows that a process used the $Input$ artifact directly to produce the $Output$. On time $T1$ however, it uses the $Output$ that was generated by a newly introduced other process, which pre-processes the $Input$.

This solution is suitable when just a few processes within the workflow change over time; creating a whole new provenance graph to record the change of one process is unnecessary. However, when the whole workflow undergoes big changes, only using timed annotations would result in one huge provenance graph containing a mish mash of timed relations.

3.5.3 Workflow State Checkpoints

The second possible solution to recall to what state a workflow was in at a particular time is to see the state of a workflow as a *checkpoint* in time. When we associate the state of a process with a certain point in time, this state will hold until a new checkpoint is created.

Results can be associated with a certain state of the process by finding out to which interval of two checkpoints the result belongs, meaning that the creation timestamp of the result lies within that interval.

It should be noted that this slightly violates the OPM standard: the OPM is strictly limited to provenance of *past* events, while the relation of provenance with the state of a process says something about the provenance of future events as well, namely that its provenance graph will be the same as the graph of the previous event if the state of the process has not changed.

In contrast to using timed annotations, this approach could be useful when a workflow changes substantially over time. When this happens, adding timed relations for every change will make the provenance graph unnecessary complicated. The difficulty here is when a workflow change can be called 'substantial'. At what point should be decided to stop using timed relations and create a new graph instead? This will mostly be a decision that will have to be made by the developer of the provenance system, but two options in particular come to mind:

- Don't use timed relations if it is known that every workflow change will greatly influence the provenance graph. In this case it makes sense to not use timed relations at all.
- Create workflow state checkpoints based on the amount of timed relations in the current graph. One can decide to create a new checkpoint regularly upon hitting a certain threshold of timed relations in the current graph in order to keep the graph simple. A new checkpoint can always be created by creating a new graph that only contains the latest timed relations from the old graph, even if the workflow state did not change.

3.6 Related Work

3.6.1 PReServ: Provenance Recording for Services

A lot of work on recording provenance has also been carried out by Groth et al. [36]. They record different kinds of provenance using *p-assertions*. A p-assertion is a *representation* of provenance. Officially, it is defined as 'an assertion that is made by an actor and pertains to a process'. The complete process that a piece of data goes through can be documented by letting involved actors record a set of p-assertions. There are three kinds of p-assertions:

Interaction p-assertion *An assertion of the contents of a message by an actor that has sent or received that message; the message must include information that allows it to be identified uniquely.*

→ When a sender S sends a message M to receiver R, sender S must create an interaction p-assertion that it has sent M to R and receiver R must create an interaction p-assertion that it has received M from S. By doing this, a data flow from S to R is documented. Interaction p-assertions document flows *between* actors.

Relationship p-assertion *An assertion by an actor that the sending of a message would not be occurring or a data item it is sending would not be as it is (the effect), if it had* *not received other messages or data items had not been as they are (the causes), and that this relationship is due to its own action, expressible as the function applied to the causes to produce the effect.*

→ A relationship p-assertion documents the relationship between an actor's output and inputs. A *primitive* actor is an actor whose output data is a function (transformation) of its input data. Non-primitive actors are influenced by other inputs; the output depends on the input message and other incoming messages. Relation p-assertions document flows *within* actors.

Actor state p-assertion *An assertion, by an actor, of data received from an (unspecified) internal component of the actor just before, during or just after a message is sent or received. It can, therefore, be viewed as documenting part of the state of the actor at an instant, and may be the cause, but not effect, of other events in a process.*

→ Based on the internal state of an actor, it might perform a different function on its input data, thus resulting in a different transformation and output result. Therefore, the actor state must possibly be recorded as well to completely document a process.

The notion of p-assertions may seem very different from the approach described in the OPM. However, the assertions described above can easily be translated to different parts an OPM model:

- Interaction p-assertions make up the main graph of the model, thus the workflow;
- Relationship p-assertions can be modeled as *overlapping accounts* in an OPM model, they are refinements of the processes in the workflow;
- Actor state p-assertions can be modelled using timed annotations. These concepts are not the same thing, but state information about a process can clearly be seen as being a kind of annotation on a process.

The idea of p-assertions and other ideas of recording provenance have been used to find shared concepts, of which the OPM is the result. Therefore, the choice has been made to use the OPM to record provenance for sensor data.

3.6.2 Provenance Aware Storage System (PASS)

Ledlie et al. discuss the requirements of a Provenance Aware Sensor System [37]. A PASS deals with storing sensor data as well as provenance data. By looking at versioning systems (CVS, SVN) and requirements of research communities, they come up with a list of characteristics and criteria of a PASS. According to the authors, a PASS has four fundamental properties distinguishing it from other storage:

- Provenance is treated as a first class object.
- Provenance can be queried.
- Nonidentical data items do not have identical provenance.

- Provenance is not lost if ancestor objects are removed.

The main type of provenance that is considered is *data provenance*: collections of tuples can be recorded and queried for within the same system, like versioning systems do. The PASS members have contributed in all Provenance Challenges that were held until now (see [33]), and have therefore been strongly involved in the creation of the OPM specification.

3.6.3 Trio

Trio is a DBMS that can store *uncertain* data and provenance (lineage) of data [38]. Data tuples inserted may be approximate, uncertain, or incomplete; all this information can be stored in the system. When updates occur, new data values are inserted in the database while old values are 'expired' but remain accessible in the system. This ensures the lineage of a tuple can still be obtained when the tuple has been derived from tuples that have been updated or deleted.

Lineage is stored in a separate relation and is able to record **When** a tuple was derived, **how** a tuple was derived and **what** data was used to derive the tuple. A lineage tuple consists of three attributes that specify the type of lineage, a description of how the lineage was derived and an optional field with additional 'lineage data'. This approach for storing provenance is very different from other approaches, of which a translation to the OPM can usually be made.

Data lineage can be queried by using a custom made, SQL-like query language called TriQL. It supports queries like '*Retrieve all temperature data imported from climate database C on 4/1/04*' or '*Given a sighting tuple t, find all data derived from it directly or indirectly*'. [38]

3.6.4 Chimera

Chimera is the oldest system described here for storing provenance, which they also call *lineage*. Data lineage is stored in a Virtual Data Catalog (VDC). Lineage data is considered *virtual*, as it is not the actual, *real* data, but data generated on demand to document information about the real data. The system uses a relational database as its backend and the VDC can be populated and queried by using the Virtual Data Language (VDL).

VDL allows a user to specify two kinds of lineage: transformations (TR) and derivations (DV). A transformation is seen as being an executable program accepting input and producing output, which can be specified in a TR statement. Derivations on the other hand are executions of transformations on a given data object. Chimera proves that these two concepts allow users to store the provenance of data objects, which was the main objective of the authors. [39]

3.7 Case study: Provenance

For the case study, the process provenance is the information that should be known to the user. Knowing exactly which data has been used to produce a certain tuple is not feasible, since new data may be produced every second. Instead, if the system is able to show a user which path a certain tuple or collection a tuple has taken, this should be sufficient to reproduce the result. To support this, the following four requirements should be met:

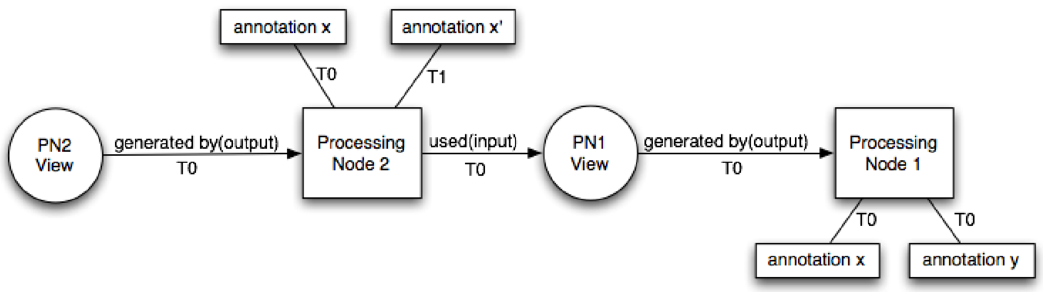

Figure 10: Provenance graph for a sensor network view

RE5 As the sensor network used consists of web services, the provenance service should also be a web service, which supports recording and querying provenance data.

RE6 Nodes in the sensor network should send provenance information to that service upon (re-)initialization. By doing this, an automatic system for recording process provenance is created, thus eliminating the need for researchers to manually add provenance information into the system.

RE7 To enable to record the state of a process, for example the sensor configuration at a certain time, it should be possible to add annotations to processes within the provenance graph of the network. This is clearly necessary to be able to reproduce a certain result.

RE8 Changes in the network should be recorded by using timed annotations or workflow state checkpoints.

3.7.1 Provenance Graphs for Sensor Data

What should a provenance graph for a piece of sensor data look like? To answer this, the following mapping is proposed that defines what a process and an artifact signifies in such a provenance graph:

Definition 3 (Process) *A* process *can be seen as any element that produces or transforms sensor data. This implies that sensors and GSN instances are in fact processes.*

Definition 4 (Artifact) *An* artifact *is something that is generated or used by a process. Simply speaking, processes (except for sensors) in the sensor network have a certain input and produce some output, therefore these inputs/outputs can be seen as artifacts. However, since such an artifact can be output for one process and input for another, these are not two different artifacts: they are the same, but have a different role. To be clear about this, an artifact is defined as being a* view *of a node, having either the role 'input' or 'output'.*

To demonstrate a provenance graph that uses these definitions and supports the mentioned requirements, consider the graph found in figure 10, which shows the provenance graph of sensor data stored in the view of 'Processing Node 2' (PN2). This graph gives the following information:

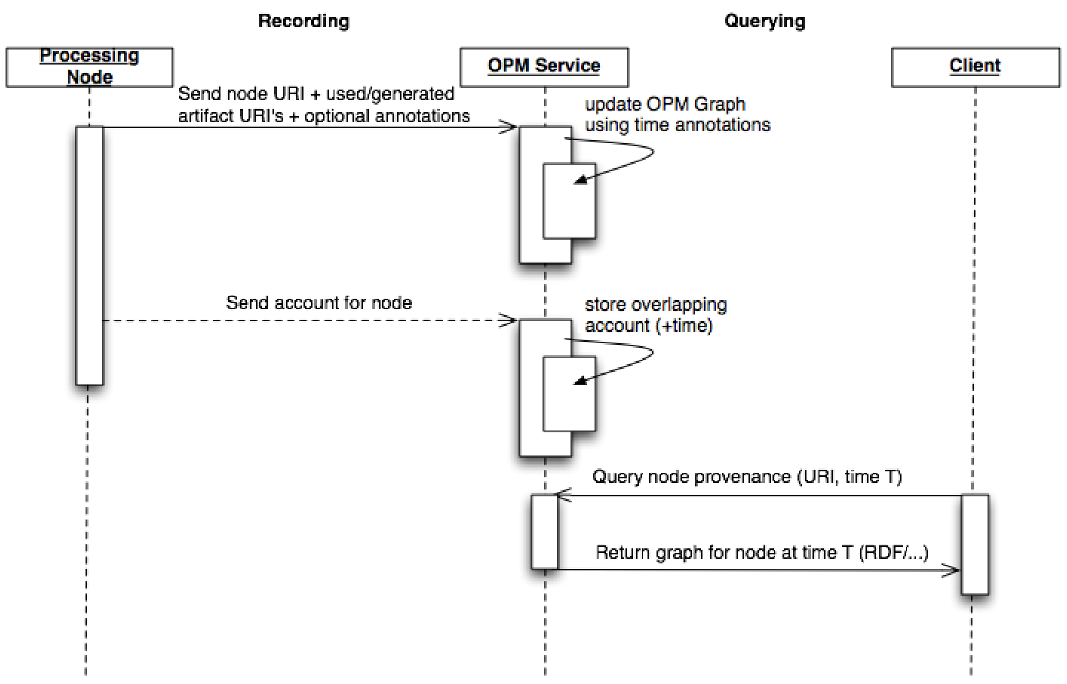
Recording
Querying
Processing Node
OPM Service
Client
Send node URI + used/generated artifact URI's + optional annotations
update OPM Graph using time annotations
Send account for node
store overlapping account (+time)
Query node provenance (URI, time T)
Return graph for node at time T (RDF/...)

Figure 11: Example of recording & querying provenance using an OPM enabled provenance service

- This view has been generated by PN2, therefore the *output* role has been defined in the *generated by* arc.
- PN2 has one annotation, *annotation x*, which has changed **since** time $T1$. A request for this provenance graph for a time between $T0$ and $T1$ should therefore contain *annotation x*, while every request for the graph after $T1$ should return *annotation x'*.
- PN2 used the view of PN1, therefore the role of the view is in this case *input*.
- The view of PN1 has been generated by PN1, so in this case the arc defines the role *output*.
- PN1 has two different annotations, *annotation x* and *annotation y*, that have not changed since the initialization time $T0$.

3.7.2 Recording and Querying the Provenance Service

Figure 11 shows what the workflow should look like. A processing node first needs to register itself. To do this, it sends the unique URI of itself, plus the URI's of the artifacts it has generated or used. The service adds this information to its known provenance graph, which is the graph of the entire network. New information could create a new graph if all passed in URI's are not yet known, but otherwise the existing graph(s) can be extended or updated. Next, a node could optionally send an overlapping account (see section 3.4) defining the internal process within the node. This could be needed if the node description does not sufficiently clarify what is happening inside the node.

Finally, a user can send a query to get the provenance graph of a node in the network at a given time by passing a node URI and a time T. The service will in this case return a subgraph of the complete graph by following outgoing arcs, starting from the node with the given URI. This will yield a graph like the example graph shown in figure 10, which should give the user enough information on how the sensor data recorded by the requested node was produced.

Chapter 4

A Query Processing Approach

Now that the requirements of the infrastructure are clear, this and the following two chapters will focus on the design of the infrastructure. As discussed in chapter 2, GSN offers a suitable way to build a sensor network that enables sensor data collection and processing in a distributed manner. However, to be able to combine data from such a sensor network with manually sampled data, which is the objective of the main research question, a query processing system is needed that can accept data coming from both sources and can process this to fulfill user specified queries. This process is different from the stream processing that a sensor network performs; query processing and stream processing are two different concepts, although they are very dependent on each other. The main difference is that stream processing doesn't have an explicit end goal: the specific data that a particular user is really after can be disregarded. A workflow can be created that provides sufficient information for all users. Query processing on the other hand does work towards an end goal, namely returning a subset of the available data that is specified by a query. Queries may contain data selections, transformations, pagination and order parameters and more, therefore some data processing is still needed to be able to return the requested result [40].

This chapter will show an approach for query processing by describing a system called the Query Manager (QM) that is responsible for accepting and processing queries. It follows the architecture that is used by System S [41] [22] and the Aurora/Borealis processing engine [11] [42]. A formalization of the workflow presented will be given that can be used for validation of implementations.

4.1 The Query Network

The core part of the Query Manager is the Query Network (QN), of which each node is called a Processing Element (PE). Unlike the GSN sensor network, the QN is residing in one application; it is not distributed. To explain the basic structure, take a look at figure 12. A QN is a directed acyclic graph, of which the root elements are called source PE's. A source PE is like a generic PE, but it is also able to directly communicate with a node in the sensor network. It is depicted in the diagram as a square with a double line. Every PE produces output, which is called a *view*. These views, which were already shortly discussed in section 3.7, are represented by the circles with the letter v in it. They are not defined as being 'output', since views can be input for a particular PE as well.

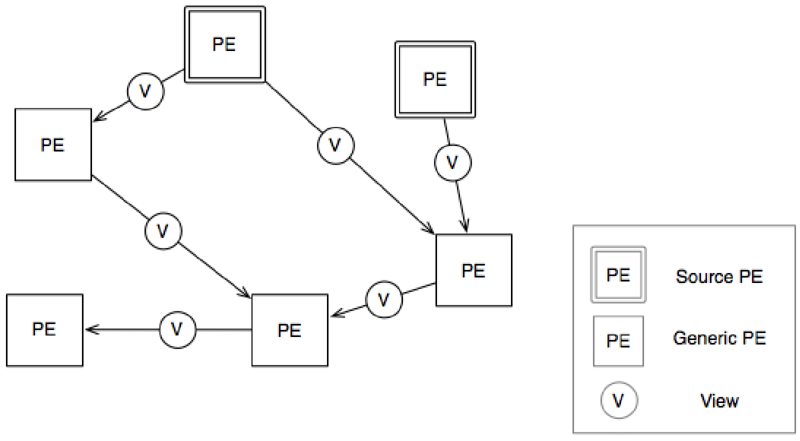

Figure 12: The Query Network

4.1.1 Processing Elements (PE's)

Processing elements are the main blocks of the QN. The big advantage of PE's is that one instance can be used by multiple queries. PE's can have the following generic functions [22] [13]:

Union Merges multiple streams.

Functor Execute a function on every tuple in the stream. Specific types of functors are selectors and projectors.

Join Joins streams based on a join predicate.

Aggregator Performs aggregation on a set of tuples in the stream: sum, average, count, etc.

Sort Sorts a set of tuples.

Noop Just pass the data through.

Of course this list of PE's only presents the most common functions of PE's. They could also perform other (special) tasks that may be required based on the system domain. For example, System S [22] also supports Barrier, Punctor and Split processing elements, but these are not necessary for a simple query processing system.

4.1.2 Formalization

This section will give a mathematical description of the QN in order to be able to verify the correctness of implementations. It will also help to understand how the system actually works in general.

To create a formalization of the QN described in this section, the following variables are defined:

- V is an input view, a schema that defines incoming tuples;
- V' is an output view, a schema that defines outgoing tuples;

- B is a buffer, a window specified by a predicate over the state of incoming views;
- T is a trigger over a buffer B which specifies when a Functor, Aggregate or Sort PE should execute;
- I is the internal state of a PE, which can be deterministic or based upon the state of incoming views.

Generic PE's accept some input view V coming from a parent PE. Source PE's accept data streams coming from the sensor network, therefore these streams are seen as input views as well. Note that a view is usually an input view as well as an output view; it is called an output view if the parent PE that generates the view is considered, while it is an input view when considering a PE that is using the view.

Given the following views:

- V_x: The output view of a given PE x from the perspective of a PE using the view;
- V_y: The output view of a given PE y from the perspective of a PE using the view.

A PE of the QN can perform these operations on these views:

- Union: $V'_u = V_x \cup V_y$. If z is a tuple in the output view of V'_u then:

  $$z \in V'_u \quad \rightarrow \quad z = t \ \wedge \ t \in V_x \vee t \in V_y$$

  $\rightarrow$ A tuple in the output view of a Union is a tuple from exactly one of its incoming views.

  Note that this formalization is the easiest case, as a union could accept more than two views. The extension to the formalization is however trivial and therefore omitted.

- Functor: $V'_f = F_{T,B_x}(V_x, I)$, where T is the trigger for executing the functor, B_x is a buffer and I is the internal state of the PE. The following functors are noted in particular:

  - Selector: $V'_\sigma = \sigma_{T,B_x}(V_x, I)$
  - Projector: $V'_\pi = \pi_{T,B_x}(V_x, I)$

  If z is a tuple in the output view of V'_f then:

  $$z \in V'_f \quad \rightarrow \quad z = F_T(t) \ \wedge \ t \in B_x(V_x)$$

  $\rightarrow$ A tuple in the output view of a Functor is a tuple from the buffer of its incoming view to which the function F has been applied.

- Join: $V'_j = \sigma_{T,B_{xy}}(V_x \times V_y, I)$, where T is the trigger for executing the functor, B_{xy} is a buffer and I is the internal state of the PE. The join is in our case the cartesian product of two views followed by a selection. The reason for this choice is that the join will usually be based on time, which can differ slightly for the incoming views. Matching on a join predicate without having a buffer would rarely yield matching elements. If z is a tuple in the output view of V'_j then:

  $$z \in V'_j \quad \rightarrow \quad z = \sigma_T(t \times u) \ \wedge \ t \in B_x(V_x) \ \wedge \ u \in B_y(V_y)$$

→ If z is an outgoing tuple of a Join, it has been selected from the cartesian product of the buffers of both incoming views (meaning it fulfilled the join predicate).

- Aggregator: $V'_a = A_{T,B_x}(V_x, I)$, where T is the trigger for executing the aggregate, B_x is a buffer and I is the internal state of the PE. If z is a tuple in the output view of V'_a then:

$$z \in V'_a \quad \rightarrow \quad z \in A_T(B'_x) \ \wedge \ B'_x \subseteq B_x(V_x)$$

→ The outgoing tuple z of an Aggregator is the result of performing the aggregate function A on a subset of its incoming view.

- Sort: $V'_s = S_{T,B_x}(V_x, I)$, where T is the trigger for executing the sort, B_x is a buffer and I is the internal state of the PE. If z is a tuple in the output view of V'_s then:

$$z \in V'_s \quad \rightarrow \quad z \in S_T(B_x(V_x))$$

→ An outgoing tuple of a Sort is a tuple from the buffer of its incoming view. The Sort function S has been applied to this buffer in advance.

- Noop: $V'_n = V_x$. If z is a tuple in the output view of V'_n then:

$$z \in V'_n \quad \rightarrow \quad z = t \ \wedge \ t \in V_x$$

→ A tuple in the output view of a Noop comes directly from its incoming view.

The Union and the Noop do not contain a trigger and buffer predicate, which is shorthand for saying that these elements have a buffer of size 1 and are triggered on every incoming tuple. It is omitted from the definition, because the buffer does not have a significant role for these elements.

The formal semantics of the outgoing tuples can be used for validation of the implementation of the processing elements; most of them will validate the correctness of the functionality of each PE. Note however that the correct implementation is not necessarily guaranteed. As a simple example: when a tuple is in the outgoing view of a Sort PE, it should be a tuple from the input view to which the sort function has been applied. However, this will not guarantee that the actual view is indeed sorted by the PE. In other words: the semantics can only be used to verify on tuple level, not collections of tuples.

4.2 The Query Manager

The Query Manager (QM) is responsible for updating the QN, optimizing queries, generating query plans and querying the network. The QM can help a user forming a query by providing details about the available nodes in the QN and the sensor network and what data structures are returned by these nodes.The following subsections will describe the general structure of the QM.

4.2.1 Architecture

Figure 13 shows the architecture of the query manager. Through node discovery, the QM is able to find out which nodes in the sensor network it can use as data sources. Within

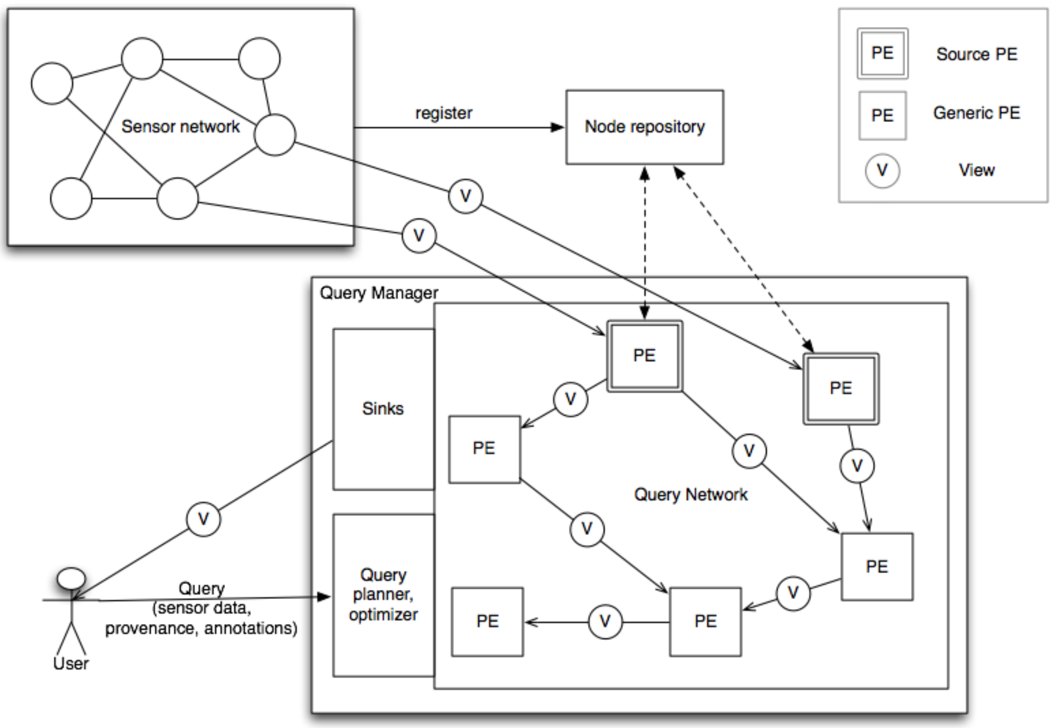

Figure 13: The Query Manager with the Query Network

the QM a query network is generated, which consists of one or more processing elements (represented by a square). A query Q from the user be split up in definitions of several processing steps and each of these steps can be handled by a certain PE. The query result (view) is generated by chaining these PE's together and collecting the data created by the PE at the end of the chain.

4.2.2 Sinks

PE's are not responsible for things like pagination and handling different output formats. Instead, it should be something that sits between the client and the view of a PE. These elements are called *sinks*. They are not part of the processing network since they should not process stream data, but instead batch process the data provided by PE's. A sink is the endpoint of a query. It handles pagination, ensures the queried data is returned in the requested output format and it could possibly also perform ordering or caching.

4.2.3 Query Planning & Optimization

The third and final block in figure 13 that was omitted until now is the Query Planner (QP). It has several tasks to fulfill after receiving a query:

- Validate the query
- Split the query into subqueries that can be handled by the available PE's
- Optimize the subqueries based on the current QN to support PE reuse

- Add new PE's to the network if necessary
- Provide a handle to enable retrieving results from the newly created path

Most points mentioned are straightforward and depend on the system software design, but a hard one is optimization. Optimization can be done in several ways and will often be achieved by chaining PE's. As a small example, consider two queries that make a selection on time, in which the specified interval of the second query is completely contained by the interval of the first query. In this case we can make two Selector PE's that both make a selection on the source data, but a good optimization in this case would be to chain the Selector of the second query to the Selector of the first query, scoping the results of the second query. This can dramatically increase the efficiency of the query. Similar optimizations can be achieved with other generic PE's as well. These optimizations are out of the scope of this thesis, but it is worth mentioning that research on this topic is currently being done as well at the university of Twente, which will be publicized in the near future.

4.2.4 Query Definition

As already mentioned, the PE network in a QM is a directed acyclic graph, from here on called graph G. The vertices V are PE's and the set of edges E are the links between PE's:

$$G_{PE} = (V, E)$$

A query Q updating the query network in the QM is seen as a *representation* R of a subgraph of graph G_{PE} that contains the vertices and edges of a path P through G_{PE}:

$$Q = R(G'), G' = (V', E') \ \wedge \ G' = P(G_{PE})$$

The query language will define what this representation will actually look like. It should be easy for the end users of the QM to define them. Section 5.2 has been dedicated to address this problem.

4.3 Case study: Query Processing

As query processing is the biggest responsibility of the prototype that will be designed for the case study, application of the theory described in this chapter for the case study is described in the prototype design created in chapter 6. The formalization described in section 4.1.2 will be used in the prototype to validate the functionality of the individual processing elements, the use of buffers and triggers and the definition of queries.

Chapter 5

Sensor Network Query Language Design

An important part if the query processing system described in the previous chapter is the query language that will allow users to update the query network and retrieve processed data. This chapter will discuss the requirements and design of this language, answering the third research question stated in section 1.3.

Currently, most stream processing systems have their own query language, which is often SQL based [7]. Another solution commonly used is to provide the user with a graphical user interface in which they can define a data workflow, as is used by Aurora [42]. GSN offers a graphical interface as well, which allows a user to compose an ordinary SQL query by selecting attributes and conditions [10]. Finally, System S provides three solutions for querying data and specifying workflows, namely by specifying the workflow programmatically, by means of a declarative query language called SPADE or by querying 'Inquiry Services' (INQ), which can only use already existing workflows, but is very easy to use for users with only little knowledge of computer science [22]. Most query languages described are custom built for the system that was designed, allowing users to only specify queries that the system is able to answer. While this is a good thing, a common idea of the requirements of a language for querying stream data becomes a bit vague. The next section will therefore first describe a list of requirements of a query language for sensor networks, followed by some extensive work on the design of a query language for the case study.

5.1 Sensor Network Query Language Requirements

Requirements of a query language for querying sensor networks does of course strongly depend on the purpose of the system that is queried. The list below is therefore not a summation of essential requirements of such a query language, but rather a list of what features one would expect the query language to support. Some important research on this topic has already been conducted by [7], so most of these points are a paraphrasis of the points stated in that publication (cited accordingly). Note that the examples given do not necessarily have to be the syntax of the final query language, the SQL-like syntax has been deliberately chosen because SQL has the most awareness within computer science.

5.1.1 QL-RE1: Support for interval selections over multiple attributes

Streaming applications always require support for selections [7]. These selections can be defined as intervals over data attributes [43]. For each attribute of some sensor data (e.g *time*, *location*, *value*) one should be able to specify multiple intervals. For example, a query like this should be possible: ‘time BETWEEN 5 PM,7 PM AND time BETWEEN 10 PM, 11 PM AND value BETWEEN 10,15’. Note that the word BETWEEN in this example makes it unclear of the given interval includes or excludes the values given. A mathematical notation should be considered in which these two cases can be distinguished. For example: ‘time:(5 PM..7 PM], time:[10 PM..11 PM),value:(10..15)’, where the parentheses mean ‘between these values but not including these values’ and the square brackets mean ‘between and including these values’.

5.1.2 QL-RE2: Support for fixed, landmark and sliding windows

The query language should support any kind of data windows and can be either time based or count based [7]. A fixed window is essentially a query over historical data. Sliding windows can be specified by creating relative interval selections. For specifying time based windows, a relative point in time is needed, since one or both endpoints of the window move over time. A logical choice is to allow intervals relative to NOW. For example, to specify a window of the sensor data of the last hour, this would look something like ‘$time : (NOW - 1h..NOW]$’. Count based windows are a little different, since they are not based on an attribute. It handles rows as a whole. So, to support count based windows, intervals need to be specified that are relative to the last row added, let’s call it $LAST$. An interval of the last 2000 values can then be specified as: $(LAST - 2000..LAST]$.

5.1.3 QL-RE3: Support for aggregations

Simple as well as complex aggregations should be supported [7]. Commonly used aggregations are averages and sums, while more complex aggregations can be found in the field of data mining, where systems support slicing, dicing and drilling with OLAP cubes [44].

5.1.4 QL-RE4: Support for stream merging

It should be possible to merge streams into one stream (a union) [7]. A typical example in which a union is useful is the case in which sensor data of the same subject can come from different sources. In the case study, some data may be manually sampled, while other measurements are gathered by GSN instances.

5.1.5 QL-RE5: Support for joining streams

To be able to see dependencies between values that have been collected, it should be possible to join streams [7]. Matching tuples in joins are usually found by comparing an attribute of one stream with an attribute of the stream to be joined on equality. Within sensor networks, it would be particularly nice to be able to join streams based on time, but in that case there will be rarely a match when checking on equality. Therefore, a special kind of join should be supported that is able to join on a *join interval*, allowing the user that two tuples match when their time attributes both lie within the given interval.

5.1.6 QL-RE6: Support for querying sensor data through annotations

For streaming data, the simplest way to store annotations is to add them as attributes to stream tuples. It should then be possible to specify intervals on annotations. Since it is common for annotations to be text rather than a numeric value, the system should have a notion of textual intervals: Examples of textual intervals are [‘cold’, ‘hot’] or [‘bt001’..‘bt004’], of which the latter is a range containing the words ‘bt001’,‘bt002’,‘bt003’ and ‘bt004’ (a *lexicographic* range).

5.1.7 QL-RE7: Support for output structure transformations

The output view of a node in the sensor network is a table in which the dimensions of the sensor data ($time$, $location$, $value$, etc.) are the columns of the table. It should be possible to transform this data into different output formats, for example HTML or CSV. To let the language support any kind of output format, a possibility could be to create several sink elements that have a given name, which can then be called by passing that name in a query. However, the downside of this is that the user needs to know somehow which algorithms are available; the language parser itself cannot check that the query is valid in this case.

5.1.8 QL-RE8: Pagination support

During query processing, pagination is not important. However, when sending data back to the client pagination is an absolute must-have. It provides a way to reduce load times, while the data is presented in a clearer way. Not having pagination increases the running time of queries and results in massive amounts of data to be transferred to the client.

5.1.9 QL-RE9: Limit expressiveness for easy querying

The query language should be simple and easy to use. By restricting the expressiveness of the query language and thus the number of ways a particular piece of data can be queried, the language becomes much easier to understand. If we see the selections in a query as boolean expressions, a selection can be represented like this: $(A \wedge \neg B) \wedge (C \vee D) \vee (\neg E \wedge F)$. One can limit the expressiveness of the queries that can be made by only allowing selections that are in conjunctive normal form (CNF). Also, an option could be to not allow the $\neg$ operator.

5.1.10 QL-RE10: Support for querying provenance

As explained in chapter 3, provenance is essential information to have for sensor data to make sense. A streaming application should be able to record this data and therefore the query language should support querying this provenance data.

5.2 Case study: Query Language Specification

The queries that will be sent to the Query Manager described in chapter 4 should be created by the end users of the system. An understanding is needed of what feels *intuitive* for these users, which is not a trivial task. Query languages come in many forms: some are very verbose and human readable, while others are very concise and mathematical. This section is a report of a small empirical study done at EAWAG, which was targeted at finding out what is preferred regarding query languages in the field of hydrology.

5.2.1 Query Syntax Examples

To find out what is an easy understandable query language for the researchers that will be using this system, a small survey was conducted in which two query examples were presented, one being verbose (snippet 1) and the other very functional (snippet 2).

Snippet 1 Verbose query example

```
GET timed, position, value OF Hourly_Avg_Dts1
INCLUDE class ANNOTATIONS OF Temp_Classifier_Dts1
LIMIT RESULTS TO channel IN [1..1] AND value IN (3..7) AND ANNOTATION class IN ("cold","hot")
LIMIT WINDOW TO (NOW-1h..NOW+1h]
FORMAT RESULTS AS CSV WITH timed AS "Measure date", position AS "point", value AS "Temperature (C)"
ORDER BY timed ASC
```

Snippet 2 Functional query example

```
data = Hourly_Avg_Dts1.get(timed|position|value, window => (NOW-1h..NOW+1h],
                           conditions => {channel:[1..1], value:(3..7)},
                           order => { timed, ASC})
 + Temp_Classifier_Dts1.annotation(class, conditions => {class:("cold","hot")});

data.format('csv', labels => { timed: "Measure date", position:"point", value:"Temperature (C)"});
```

Both queries would yield exactly the same result, namely: *Give me $timed$, $position$,$value$ and the $class$ annotation of $Hourly\_Avg\_Dts1$, $Temp\_Classifier\_Dts1$ since the last hour until an hour from now, where $channel$ is 1, $value$ is between 3 and 7 and the $class$ annotation is 'cold' or 'hot'. Order the results by $timed$ and yield the results as CSV, where the label for $timed$ is 'Measure date', $position$ is 'point' and $temperature$ is 'Temperature (C)'*. By presenting the participants of the survey with these queries, it is expected that the preferences of the users can be derived. This could be one or the other, or maybe a combination of both. Of course, both queries were explained in detail in a presentation held while the survey was filled in.

5.2.2 The Query Language Survey

The aim of the survey was to answer the following questions:

- Is there are correlation between a participant's academic background and his knowledge of programming and programming languages?
- Do the participants prefer the verbose or the mathematical language?
- Why is one preferred over the other? Is it the readability, the familiarity with the syntax or something else?
- Is the preferred language also the easiest to understand for the participants?
- Are the participants familiar with the interval notation?

The survey used can be found in Appendix A.

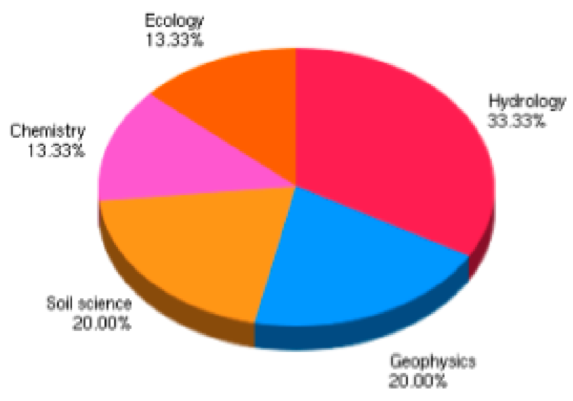

Figure 14: Main research areas of the participants

5.2.3 Survey Results

Fifteen researchers from the RECORD project group were attending the presentation during which the survey was conducted, which is about 25% of all members of the group (including master students). The pie chart in figure 14 displays the main research areas of the participants, which clearly shows that the whole group was a good selection of the members of the RECORD project. Normally, a good survey has a 95% confidence level and a 5% precision level [45]. As the number of participants is already known, the confidence level and precision level can be determined by using the formula of determining the sample size for proportional results, where we have assumed 95% of the population has an attribute of interest and Z is the Z value for a 95% confidence level:

$$n = \frac{P(1-P)}{\frac{A^2}{Z^2} + \frac{P(1-P)}{N}} \Rightarrow 15 = \frac{0.95 \cdot (0.05)}{\frac{A^2}{1.96^2} + \frac{0.95 \cdot (0.05)}{60}} \Rightarrow A \approx 0.096(\pm 10\%)$$

Conclusively, with a 95% confidence level only a 10% precision level can be achieved instead of 5%. This should be kept in mind when deriving conclusions from the survey. The following subsections will further elaborate on these results to answer the questions listed in subsection 5.2.2 as good as possible.

5.2.3.1 Correlation between Academic Background and Programming Skills

The survey showed that about 73% of the participants have had some experience in programming, mostly by creating scripts in R or Matlab. This number comes mainly from the participants in the field of hydrology and geophysics, of which all have used at least one programming language, as can be seen in figure 15 the percentage of participants with programming skills per main research area has been plotted.

In contrast to the high percentage of participants with programming experience, the majority of them do not have any experience when it comes to query languages. The ones that did mention to have experience with query languages often chose Wiki Query, which was greatly influenced by the fact that just before the survey was conducted, a small presentation about this language was given.

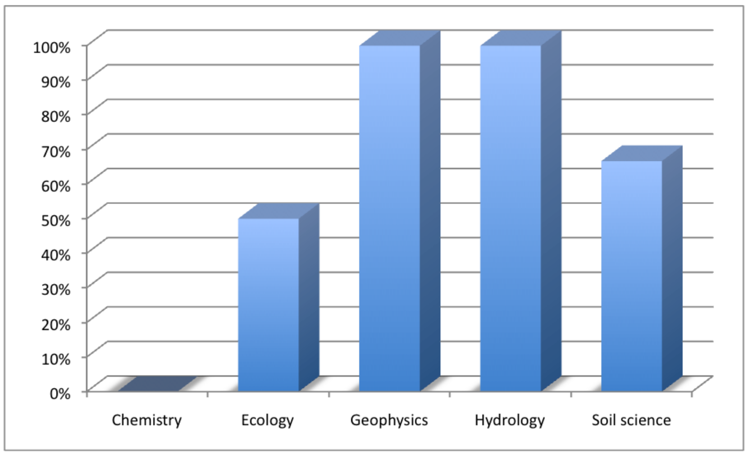

Figure 15: Percentage of participants with programming experience per main research area

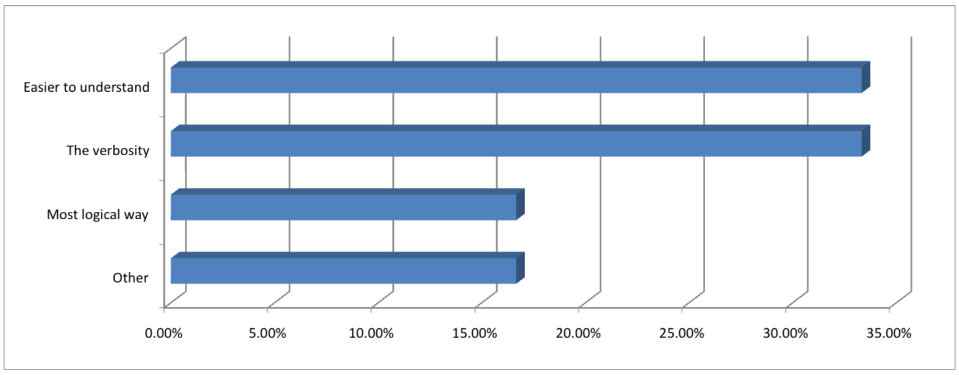

Figure 16: Reasons for preferring the verbose notation (40% of the participants)

Conclusively, the majority of the users of the system have a reasonable knowledge of scripting and functional programming, for which a mathematical syntax may be the most appropriate choice for the final query language.

5.2.3.2 Query Example Preference

By forcing the participants to choose between the verbose and the mathematical notation, the mathematical notation was preferred by 60%. The reasons for choosing one over the other can be seen in figures 16 and 17. Noticeable is that the mathematical way is preferred for it being the most logical way, while the majority agrees that the verbose notation is easier to understand.

The 60-40 result is not a significant difference to conclude anything useful. Based on these results it can only be said that there is a slight preference for the mathematical notation, but it wouldn't be a problem if the other notation is chosen.

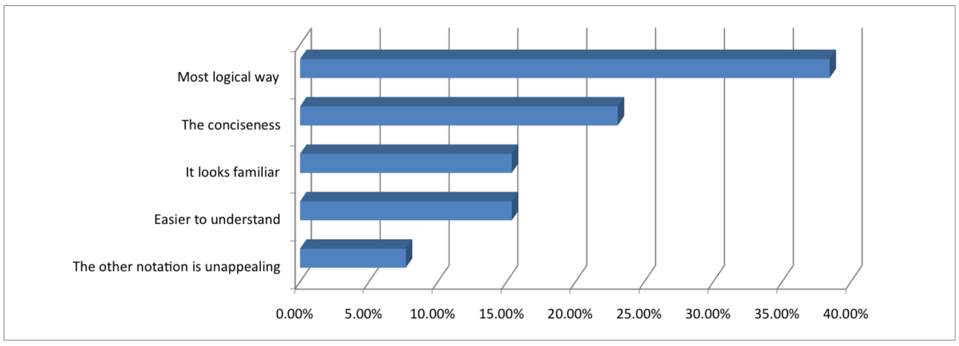

Figure 17: Reasons for preferring the mathematical notation (60% of the participants)

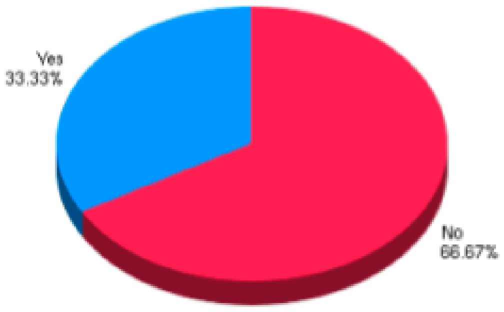

Figure 18: Familiarity with interval notation

5.2.3.3 Interval Notation

Surprisingly, two third of the participants were not familiar with the interval notation that had been used in both examples, while it is quite common to define them like this (see figure 18). Fortunately, more than 90% found that the interval notation to be a logical choice of which a big majority (80%) found it easy to understand. Therefore, it is unnecessary to look for a different notation.

5.2.3.4 Other Remarks

Documentation & Examples A lot of the participants mentioned that providing extensive documentation and examples is actually more important than how the syntax will look like. Currently, the query languages they need to use are Wiki Query and SPARQL and some are struggling for hours to find out the right syntax. This is of course an important point that should indeed be done after integrating the new system. An even better solution would be to design a user interface that allows for visually creating the queries. This is however out of the scope of the project, but could be done in succeeding projects.

Symbols An interesting idea that was mentioned was to introduce symbols for the most commonly used functions. There are a couple of possibilities here that should be thought of when designing the system:

- Inclusion of several custom functors in the system that can be called by passing functor identifiers in the query.
- Separate query formulation and registration from query representation. A query interface in the wiki could be created for managing the queries in the system, while the actual output of the registered queries is done by formulating queries that contain identifiers referencing the several registered queries in the query manager.

5.2.4 Survey Conclusions

First, based on the fact that most participants have reasonable knowledge on scripting in Matlab or R and slightly prefer the mathematical notation, that language syntax should be the first choice. However, this is not a must since the verbose notation is clear and easy to understand for the participants as well. The final choice can therefore be based on what is easier to integrate in the system. Second, the interval notation should not be changed; despite the fact that it is not well-known, it is easy to understand. Finally, sufficient documentation and examples should be supplied to the end users and the possibilities of defining symbols, described in paragraph 5.2.3.4 should be explored.

5.2.5 Final Query Language: PASN-QL

Since the survey showed a slight tendency towards the mathematical query syntax, the final query language will also be functional. The language proposed will be called *PASN-QL*: a Provenance Aware Sensor Network Query Language. This subsection will present the context-free grammar of this language, which will be specified by following the approach described in [46]. As an example of a query, snippet 2 has been used, extended and slightly adjusted for easy parsing in snippet 3.

Snippet 3 PASN-QL Functional query example

```
table1 = data(Hourly_Avg_Dts1, attributes => timed|position|value,
                               conditions => {channel => [1..1], value => (3..7)});

table2 = data(Temp_Classifier_Dts1, attributes => timed|class,
                                    conditions => {class => ('cold','hot')});

jointable1 = join(table1|table2, on => timed, delta => 5s, trigger => LAST < MAX_SID,
                                 window => { table1 => [NOW-1h..NOW], table2 => [NOW-15m..NOW] });

show(jointable1, 'data', format => 'csv',
                         labels => { timed => 'Measure date', value => 'Temperature (C)'},
                         prefixes => { position => 'P'},
                         postfixes => { value => 'C' });

show(jointable1,'provenance_graph');
```

The terminal symbols that are included by PASN-QL are:

$=$	(	)	|	,	$=>$
′	;	.	..	{	}
[	]	*NOW*	*MAX_ID*	*LAST*	*MAX_SID*
$+$	$-$	$<$	$<=$	$=$	*show*
data	*project*	*select*	*union*	*join*	*attributes*
on	*delta*	*conditions*	*window*	*trigger*	

These symbols will show up in bold in the production rules defined later on. Next, the query language contains the following nonterminal symbols:

Query (start symbol)	Command
Show-Method	Assign-Method
Attributes	Conditions
Join-Predicate	Delta-Value
Window-Predicates	Window-KV-Pair
Window-Value	Window-Expression
Window-Interval	Window-Hash
Trigger-KV-Pair	Trigger-Expression
Time-Literal	Time-Letter
Operator	Comparison-Operator
single-KV-Pair	
Hash	Non-Bracket-Hash
Collection	Interval
Value	single-Value
Left-Bracket	Right-Bracket
String	Number
Identifier	

Using these symbols, this yields the following production rules (defined in EBNF):

```
Query                  ::=   Command+
Command                ::=   Identifier = Assign-Method ;
                        |    Show-Method ;
Show-Method            ::=   show ( Identifier , String (, Non-Bracket-Hash)?  )
Assign-Method          ::=   data ( Identifier (, attributes => Attributes)?  (, Conditions)?  )
                        |    project ( Identifier , attributes => Attributes (, Window-Predicates)?  )
                        |    select ( Identifier , Conditions (, Window-Predicates)?  )
                        |    union ( Identifier (| Identifier)+ (, Window-Predicates)?  )
                        |    join ( Identifier | Identifier , Join-Predicate (, Window-Predicates)?  )
Attributes             ::=   Identifier (| Identifier)*
Conditions             ::=   conditions => Hash
Join-Predicate         ::=   on => Identifier (, delta => Delta-Value)?
Delta-Value            ::=   Number (Time-Letter)?
Window-Predicates      ::=   Window-KV-Pair (, Trigger-KV-Pair)?
                        |    Trigger-KV-Pair (, Window-KV-Pair)?
Window-KV-Pair         ::=   window => Window-Value
Window-Value           ::=   Window-Interval
                        |    Window-Hash
Trigger-KV-Pair        ::=   trigger => LAST Comparison-Operator Trigger-Expression
Window-Expression      ::=   NOW (Operator Time-Literal)?
                        |    MAX_ID (Operator Number)?
                        |    String
Trigger-Expression     ::=   NOW (Operator Time-Literal)?
                        |    MAX_SID (Operator Number)?
                        |    String
Time-Literal           ::=   Number Time-Letter
Time-Letter            ::=   h | m | s
Comparison-Operator    ::=   < | <= | =
Operator               ::=   + | -
Hash                   ::=   { Non-Bracket-Hash }
Non-Bracket-Hash       ::=   single-KV-Pair (, single-KV-Pair)*
Window-Hash            ::=   { Identifier => Window-Interval (, Identifier => Window-Interval)* }
single-KV-Pair         ::=   Identifier => Value
Value                  ::=   Collection
                        |    Hash
                        |    single-Value
Collection             ::=   Left-Bracket single-Value Interval Right-Bracket
Interval               ::=   ..  single-Value
                        |    single-Value (, single-Value)*
Window-Interval        ::=   Left-Bracket Window-Expression ..  Window-Expression Right-Bracket
single-Value           ::=   String
                        |    Number
Left-Bracket           ::=   ( | [
Right-Bracket          ::=   ) | ]
Identifier             ::=   Letter | Identifier Letter | Identifier Digit
String                 ::=   ' Graphic* '
Number                 ::=   Digit+ (.  Digit+)?
```

The nonterminal symbols *Letter*, *Digit* and *Graphic* have been omitted from the production rules since they are self-explanatory. For more information, see [46].

As can be seen in snippet 3, queries can be split up in a definition part and a representation part. This will allow users of the system to display the result of a query on several places, while being able to manage the query itself on a central place. How this can be implemented will be shown in the next chapter, where the development and implementation of a prototype will be described.

Chapter 6

Case study: Prototype

This chapter describes the design and implementation of a prototype to exemplify the theory of the previous chapters. The design of the system will use the solutions that have been chosen in the 'case study' sections in the previous chapters and will provide an architecture from the global level to the implementation level.

6.1 Global System Architecture

To get a first impression of the processes, software components and the infrastructure of the system, a diagram showing this architecture can be found in figure 19. It has been created using ArchiMate [47], which is typically meant for specifying enterprise architectures, but turns out to be a suitable solution for describing distributed systems. The diagram consists of three main layers: the process layer (blue), the application layer (orange) and the infrastructure layer (green). The following subsections will further elaborate on the different parts of the diagram, starting from the top. The components described in this diagram are worked out in more detail in section 6.2.

6.1.1 Processes

The first group of objects shows the four main processes that the system will handle, namely Provenance Retrieval, Query Data Retrieval, PE Information Retrieval and Query Registration. Query registration consists of some subprocesses that will need to be done for successfully registering a new query: a new query needs to be composed and submitted, after which the query network should be updated. Note that getting information about the network is probably also required for being able to compose a query, but since this process may also be used on its own, it has been excluded from the query registration process.

6.1.2 External Services

There are four (external) services to enable the aforementioned processes. These services can be web services, but can also represent a software application or interface. In this case, the Data Retrieval Service, PE Information Service and Query Registration Service are in fact web services, while the Query Management Service will probably be a software or web application.

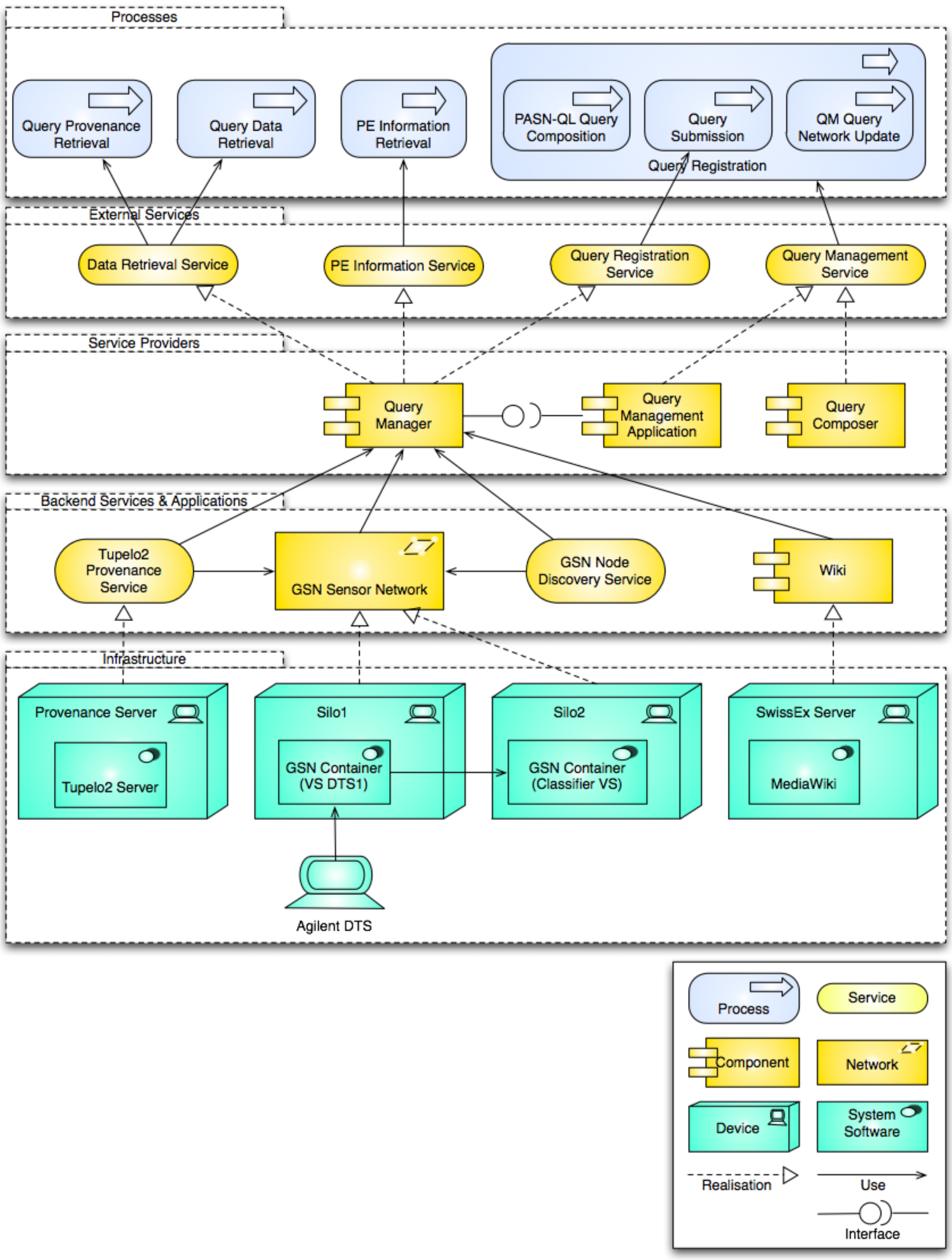

Figure 19: Prototype global system architecture

6.1.3 Service Providers

The systems that provide the external services are modelled as components of the prototype. The dashed arrow in ArchiMate stands for realisation, and as such it can be seen that the Query Manager is responsible for most of the external services provided. The Query Management Application, which can be a software application or webpage, is an interface of the Query Manager for maintaining the query network and thus provides the query registration service. Finally, the Query Composer is a stand-alone tool that can aid the user in specifying queries in a visual way.

6.1.4 Backend Services & Applications

The backend services & applications layer consists of services and components that are not visible by the users of the system. First, there is a service for recording and querying provenance named Tupelo2 Provenance Service. The Tupelo2 provenance library [48] will be used to implement this service, hence its name. Next, there is the GSN Sensor Network, which has been represented as one object in the architecture. This object entails all GSN containers that the Query Manager can use as data sources. The Query Manager will use the Node Discovery Service to find these containers. Finally, the SwissEx Wiki will also be a data source of the Query Manager. It contains a lot of manually recorded and annotated sensor data that can also be useful for the users of the system.

6.1.5 Infrastructure

The last group shows the infrastructure of the system. There will be a separate provenance server and multiple servers will be running the GSN containers that make up the GSN Sensor Network. The Agilent DTS has been modelled as a device connected to one of these GSN containers. The SwissEx Wiki is running on its own server somewhere in Switzerland.

One might notice that the Query Management Application and the Query Composer are not connected to any device in the infrastructure layer. The reason for this is that the device on which these components are running is unknown: it may be running on a client computer, but it can also be provided by a webpage on a separate server or the SwissEx Wiki.

6.2 Implementation

6.2.1 Query Manager (QM)

The Query Manager can roughly be divided into three parts: the query network, the query parser/planner and the web frontend. Figure 20 shows a simple diagram explaining the overall structure.

A client of the system can use three services to interact with the network: an information service for retrieving information about the current status of the network, a data retrieval service for getting the data recorded by nodes and a registration service for adding new nodes. All these services use the query parser to understand what the client wants to do, although registration queries are first passed to the query planner to perform possible optimizations (see section 4.2.3). The following sections will further describe the implementation of these three parts of the system.

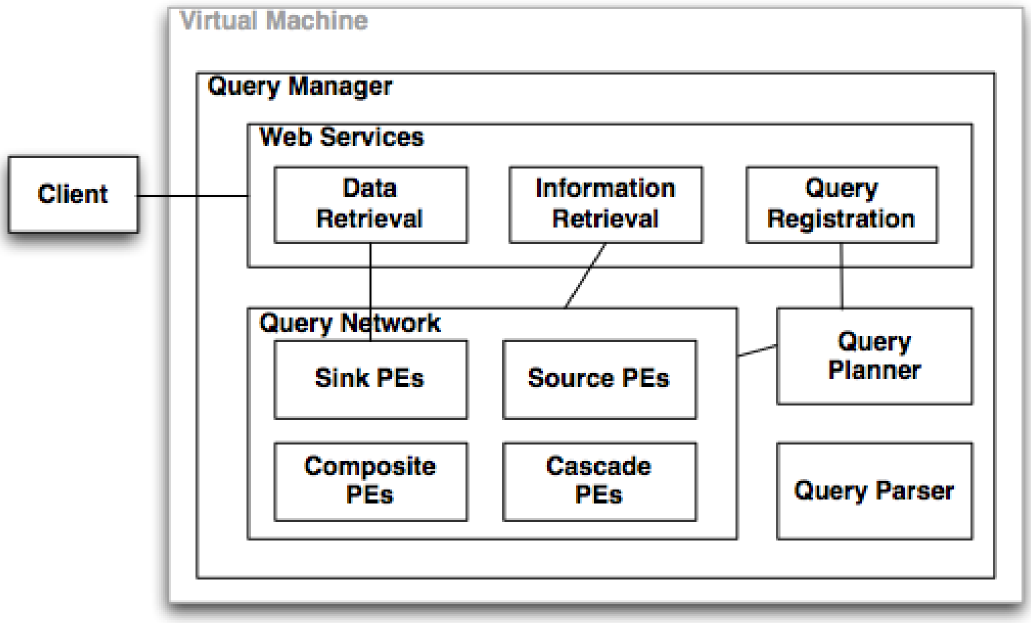

Figure 20: Query Manager global architecture

6.2.2 Query Network (QN)

As described in chapter 4.2, the query network is a hierarchy of nodes that pass data to each other. It closely follows the stream processing algorithms described by [43]; it consists of processing elements, views, buffers and triggers. Several design patterns have been used to implement the network, of which the Composite Filter pattern as proposed by Yacoub et al. [49] is the most important pattern used, since it defines the structure of the network. This structure can clearly be seen in the class diagram in figure 21. It might be a bit confusing why some methods defined in the AbstractPE class are also defined in its subclasses. The reason is to show that the AbstractPE provides default implementations, which are overridden by a subclass. For example, the methods *setParent*() and *unsetParent*() defined in the class *StreamSource* are explicitly defined, since they raise an exception when someone tries to give them a parent node.

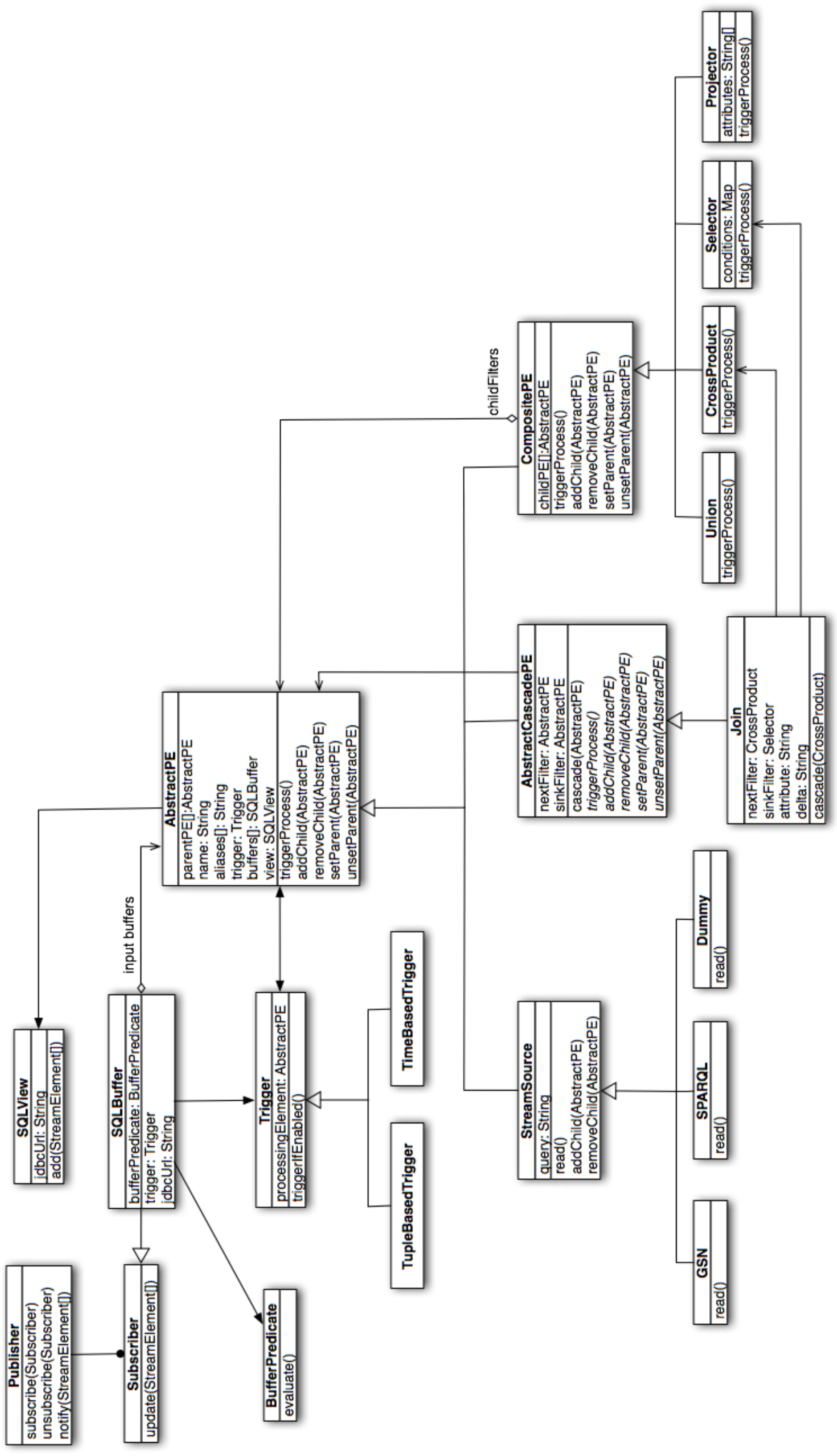

Figure 21: Query Network architecture

6.2.2.1 Workflow

The core element in the network is the Abstract Processing Element or *AbstractPE*. AbstractPE's can have children of the same class, which are added as described by the composite filter pattern [50]. Each node can either be a source node or a processing node. A source node periodically reads data from an arbitrary source, for example a running GSN instance or a sensor, and adds any new data to its *view*. The view of a node persists all data that a node has generated.

After new elements have been added to a view, the view notifies all objects that are interested in this event. Here, the second software pattern is introduced, namely the Publish-Subscribe or Observer pattern [50]. In this case, views are publishers notifying buffers of direct child nodes of the new elements that have been added. It should be noted that this event notification is asynchronous and multi-threaded, as is not the case with the Observer-Observable implementation found in the Java Core API.

For every tuple that is added to a buffer, the *buffer predicate* is evaluated, which enforces the size of the buffer. In other words, a buffer is a window and the buffer predicate enforces that window. As described in section 2.1.2, this window can be a fixed, landmark or sliding window.

A processing node performs its task on all buffers when it is triggered by its trigger. The trigger can be time or tuple based. A time based trigger executes at fixed intervals, while a tuple based trigger is explicitly notified of new tuples by the buffers that it is observing. The Observer pattern has again been applied here to achieve this. After a processing node has been triggered and it has added new elements to its view, the described workflow starts again between that node and the children of that node.

An example of this workflow with two sources and two processing elements, namely Join and Union, can be found in figure 22. Note that the union has a tuple based trigger, while the join has a time based trigger. The dashed arrows pointing to the time based buffer denote that the trigger processes its buffers like the tuple based trigger does, but it is not notified by the buffers to start processing. This is done by an external timer thread instead.

6.2.2.2 Orchestration

To manage the whole network, a facade class has been created called the Query Network Manager. The query network manager is able to start and stop the network, it can serialize and deserialize the network and it provides an easy way to add or remove nodes. It is also an important object for the network itself, since it manages the thread pool in which every new thread runs.

6.2.2.3 Stream Sinks

Stream sinks directly handle data requests of a user. A sink depends on the view of an AbstractPE, from which they get the requested data. They provide different output structures, pagination and (simple) ordering of the data. Since these sinks are not part of the processing chain of the network, they are not modelled as AbstractPE nodes but as a separate class of objects.

Besides sinks that retrieve data from views, there are also a couple of sinks that retrieve and transform provenance graphs. All nodes record provenance data upon initialization,

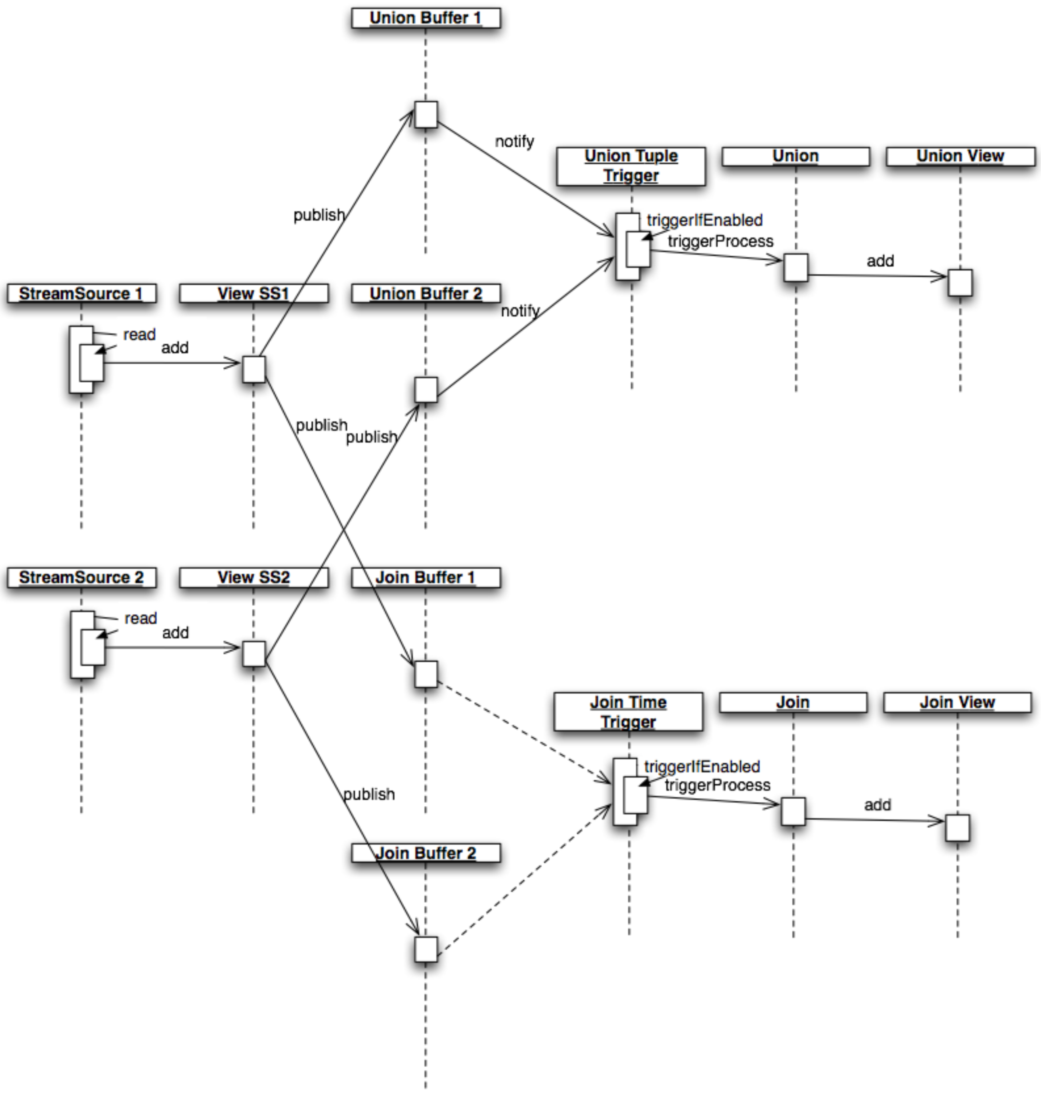

Figure 22: Example workflow sequence diagram

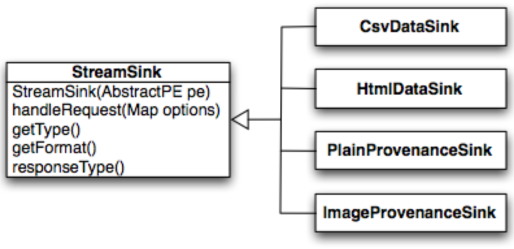

Figure 23: Stream sinks

which a provenance sink can request again by passing the URI of a node.

Figure 23 shows the structure of a StreamSink. A sink has always three attributes: a node in the network to which the returned data applies, a format that can be passed in a query (for example ‘html’ or ‘image’) and a response type denoting the MIME type of the returned data.

6.2.2.4 Implementation Alternatives

At several points during development, choices had to be made in case multiple solutions were available. These alternatives are described here, including an explanation on why a certain alternative was chosen.

- Composites and cascades
  The network supports both composite and cascade nodes. This choice was made to account for processing elements that consist of multiple processing steps, like the join. The join is actually an encapsulation of a selector and a crossproduct node. An option could have been to leave out the cascades and create two different nodes upon creating a join. The main reason for not doing this was because a user will pass a node name upon creation of a node to which he can reference in later queries. Since a node name must be unique, a consequence of splitting up the join would be that only one of the nodes would get the passed name. This didn’t seem intuitive, therefore both composites and cascades are supported.

- Buffer cloning/calculation
  The task that a processing node is executing is run in a separate thread. A problem that comes with this is that somehow the state of the buffers at the time the process was triggered must be known or calculated. This problem has been solved in the current system by copying the buffers in memory and passing those to the process. Although buffer calculation would probably have been a better solution, there were a couple complex problems that had to be accounted for:

  - Buffer size. The size of a buffer should be bigger than the requested buffer size, since a processing node would need elements that may not be part of the defined window. Pointers would need to be introduced that specify the boundaries of the buffer. A size of twice the requested buffer size would not be sufficient, since it is well possible that element addition can occur multiple times while a node is processing new data.
  - Removal of old elements. If the size of the buffer is defined by pointers, somehow the system would need to find out which elements are not needed anymore and can be thrown away. Although this is definitely a feasible option, the solution would be less elegant, since the processing node would need to give feedback to the buffer on which elements it has processed.

- Database processing of buffers
  The processing nodes in the system have their own implementation of the different kinds of transformations that a node could perform. Of course, these transformations are long known within the world of databases, so why not use the power of the database to perform the transformation? The main argument for not having done this

is to keep the system database agnostic. If the speed gain would really be a concern, it would still be possible to introduce new processing elements that delegate the transformation to the database. Besides this, a couple of other reasons are:

  - that the buffer elements would need to be stored in the same database as the view of the processing node;
  - that it introduces the challenges of knowing the buffer state again, see the previous point.

- Sinks as processing nodes
  Sinks were originally modelled as nodes in the network, but this was changed during the implementation of the network. The reason for this is that a sink is not part of the processing, it just provides a representation of the view of a given node. It doesn't need buffers, triggers and views, because it is using the view of another node. However, this does not mean that certain capabilities of a sink could still be performed by a processing node, like for example the possibility to order the data. A good reason for doing this is that a sink does not persist (cache) its results, it transforms data on the fly. Introducing a processing node that performs some ordering beforehand could greatly increase response times.

- Source definition
  Different kinds of sources are defined in an XML file. This has been done because there are several variables that need to be supplied when defining a new source, something that a user wouldn't want to supply within a query. As it turns out, this definition file could also be used as a resource containing defaults values, which a user can override in a query. This is part of the future work that could be done on the system.

6.2.2.5 Features

The query network provides the following features:

- Multithreaded processing of stream data from several sources, including GSN Virtual Sensors and SPARQL services
- Element selection, projection, union, join and crossproduct
- Factories for building buffers, triggers and buffer predicates
- Database connection pooling
- Thread pooling
- Detailed logging, JMX monitoring using *jconsole*
- Network serialization/deserialization
- Decoupled persistence layer (defaults to SQL)
- CSV and HTML output representations
- Process provenance recording using the Tupelo2 service

- Plain text and PNG image representations of provenance graphs
- Pagination and ordering of results

6.2.3 PASN-QL

6.2.3.1 Lexer & Parser

The query language processor has been implemented using ANTLR V3 and ANTLRWorks [51] [52]. Every query received by the system first passes the Lexer, which splits the query into tokens. These tokens are the terminal symbols defined in section 5.2.5. The resulting token stream is then sent to the Parser, which checks if the query is semantically correct. For this, it uses a grammar based on the EBNF grammar that is also described in section 5.2.5. The grammar can be found in appendix B. Note that this grammar also contains the grammar of the Lexer. If the query is correct, it builds an Abstract Syntax Tree (AST) of the query. This AST is not the direct AST resulting from the parser grammar, but rather a simplified one in which unnecessary tokens are dropped from the tree. 'Unnecessary' means in this case that a token is only needed to make the language more readable, that it was only needed for successful lexing, or that the meaning of it is implied by the final AST. The grammar in appendix B reflects this. Every rule in the grammar is of the form: $\textit{non-terminal symbol} : AST \rightarrow SimplifiedAST$.

6.2.3.2 Tree Walker

A tree walker has been created to walk down the (simplified) tree generated by the parser. It is specified as an ANTLR grammar as well and can be found in appendix C. The tree walker converts the tree into a list of Command objects. There are for example show, data and join Commands. The grammar also shows that in this step all specified intervals are created and that time expressions are converted to milliseconds, the unit of time that is always used by time related objects in the query network.

6.2.3.3 Command Interpretation

Every Command object created by the tree walker has a $validate()$ function, which is executed just before the command itself is executed and it checks whether the passed node names and attributes exist. The reason why it is executed just before execution is that a command can be dependent on a command that is higher up in the complete list of commands. This does imply that a complete query can partially succeed. This may be undesired, but commands executed can easily be rolled back if needed. The actual execution of a Command is contained within the object itself. For commands specifying nodes, a new node is created and registered to the Query Network Manager. Factories in the query network are responsible for creating the buffers, the trigger and the view of a new node. After the node has successfully been added, a user can directly query its view. The Show command, which is responsible for returning view data to the user, reuses or instantiates applicable stream sinks on execution, which are discussed in section 6.2.2.3. The query information will finally be passed to the sink, which generates a response that is directly written to the servlet output stream.

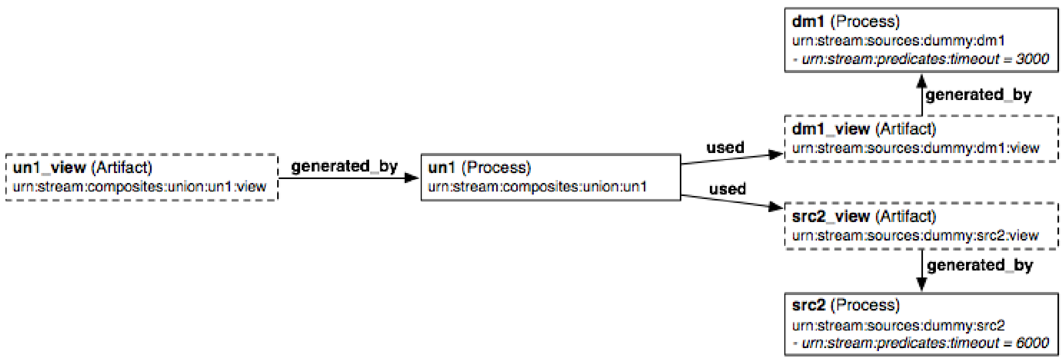
dm1 (Process)
urn:stream:sources:dummy:dm1
- urn:stream:predicates:timeout = 3000
generated_by
dm1_view (Artifact)
urn:stream:sources:dummy:dm1:view
used
un1_view (Artifact)
urn:stream:composites:union:un1:view
generated_by
un1 (Process)
urn:stream:composites:union:un1
used
src2_view (Artifact)
urn:stream:sources:dummy:src2:view
generated_by
src2 (Process)
urn:stream:sources:dummy:src2
- urn:stream:predicates:timeout = 6000

Figure 24: An example provenance graph returned by the Tupelo provenance service

6.2.4 Tupelo2 Provenance Server

To illustrate the provenance recording and retrieval capabilities of this system, a Tupelo2 [48] service has been created. The Tupelo2 project is aimed at creating a metadata management system. It can store annotation triples (subject-predicate-object) in several kinds of databases, including normal relational databases. The creators of the project have also created an API for managing the Open Provenance Model (OPM) entities described in section 3.3. The service returns provenance graphs like the one found in figure 24, which is the provenance graph of the Union processing node named *un1*.

To keep the implementation simple, a servlet has been created that adds new triples to the database based on HTTP POST requests and returns (a part of) the provenance graph when it receives a GET request. Although the Tupelo2 project also comes with its own Tupelo2 server web application, this package was not used for a couple of reasons:

1. No OPM support. The OPM implementation provided by Tupelo2 is basically a wrapper around the normal API and was probably added after the Tupelo2 Server was implemented. An option could have been to extend the server functionality, but this brings me to my second argument:

2. Too complex communication protocol. The server listens to an extended set of HTTP methods, which are based on the URIQA communication protocol [53]. Examples are MGET, MPUT and MQUERY. The goal of this protocol is to be able to manipulate the actual data using the normal HTTP methods, while the extension handles its metadata. This approach would make the implementation overly complex because of the following two reasons:

   (a) The prototype only records process provenance. The actual data is not available and therefore the extensions are unnecessary.

   (b) The purpose of the service is not to return metadata about a single resource. Instead, it should return the whole subgraph of resources that were involved in the creation of a given resource.

The current implementation can use a PostgreSQL or a MySQL backend and is able to return the whole graph in RDF or a subgraph in a textual format that is easy to read and parse. To add new provenance data, a POST request should be sent that contains an *action*

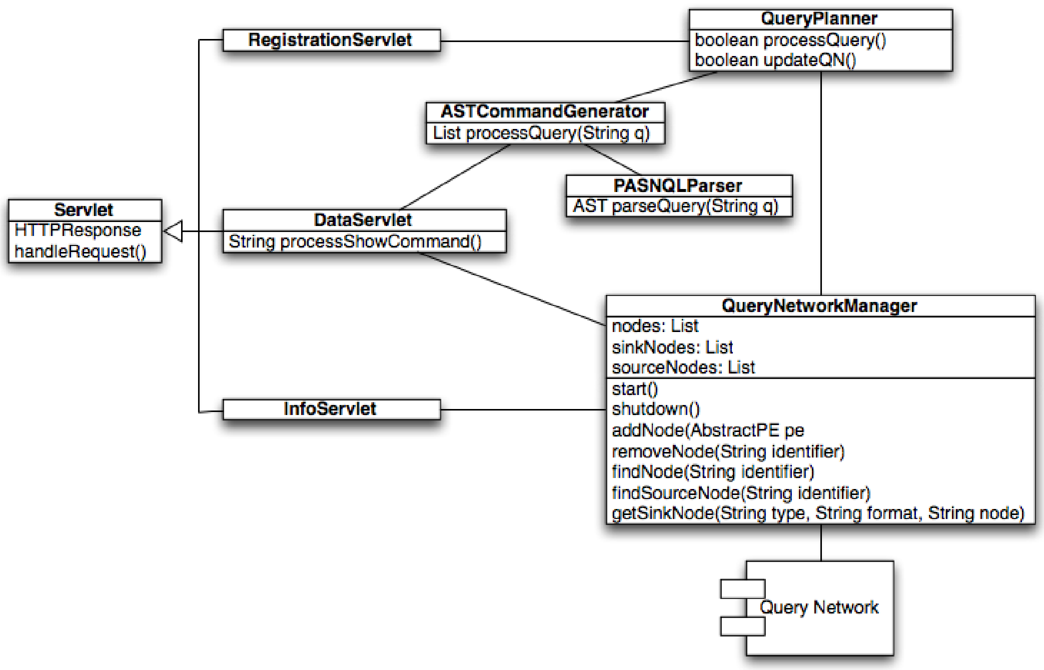

Figure 25: Query Manager services & the Query Network manager

parameter followed by required attributes that are needed to execute that action. See table 6 for a list of available commands.

6.2.5 Web Services

The web services provided by the Query Manager are simple servlets. Figure 25 shows the basic structure. The query planner and the data servlet both use the PASN-QL parser to parse the received query. The resulting tree is then sent to the ASTCommandGenerator, which initiates the PASN-QL tree walker and returns a list of Command objects. Finally, the QueryNetworkManager (the query network facade) is used to update the network or to create a sink when the commands received are executed.

The InfoServlet does not accept queries, but communicates directly with the network manager. It can only perform two tasks: starting/stopping the network and providing information about the state of the query network. The action that is performed is simply based on checking a POST parameter called ‘action’.

To show what happens when a servlet retrieves a query, take a look at the sequence diagrams of the registration servlet (figure 26) and the data servlet (figure 27). Note that the involvement of the query network manager is not shown in the diagrams to keep them simple.

When the registration servlet receives one or more registration queries through a POST request, they are delegated to the query planner. The query planner in turn sends the (possibly optimized) queries to the ASTCommandProcessor, which returns Command objects. The commands are then validated and the query network is updated. If every command completed successfully, positive feedback is sent back to the user. The diagram only shows the successful case. When something goes wrong, the process immediately stops and re-

Table 6 Tupelo servlet commands

Action	Attributes	Description
new_artifact	**name** The artifact name **artifact_uri** An URI identifying this artifact	Builds and asserts a new artifact
Example: *action=new_artifact&name=join_output&artifact_uri=urn:stream:join_output*		
new_process	**name** The process name **process_uri** An URI identifying this process	Builds and asserts a new process
Example: *action=new_process&name=selector&process_uri=urn:stream:select*		
annotate	**process_uri** An URI identifying the process to be annotated (the subject) **pred** An URI specifying the annotation predicate. This can be a custom URI or for example a Dublin Core URI. **object** The object (value) of the annotation	Annotates an existing process
Example: *action=annotate&process_uri=urn:stream:join& pred=urn:stream:predicates:buffer&object=15m*		
new_usedby	**artifact_uri** An URI identifying the artifact being used **process_uri** An URI identifying the process using the artifact **artifact_role** An URI specifying the role of the artifact	Define a 'Used By' relation between an artifact and a process
Example: *action=new_usedby&process_uri=urn:stream:join& artifact_uri=urn:stream:source_output&artifact_role=urn:stream:input*		
new_generatedby	**process_uri** An URI identifying the process generating the artifact **artifact_uri** An URI identifying the artifact being generated **artifact_role** An URI specifying the role of the artifact	Define a 'Generated By' relation between a process and an artifact
Example: *action=new_generatedby&process_uri=urn:stream:join& artifact_uri=urn:stream:join_output&artifact_role=urn:stream:output*		

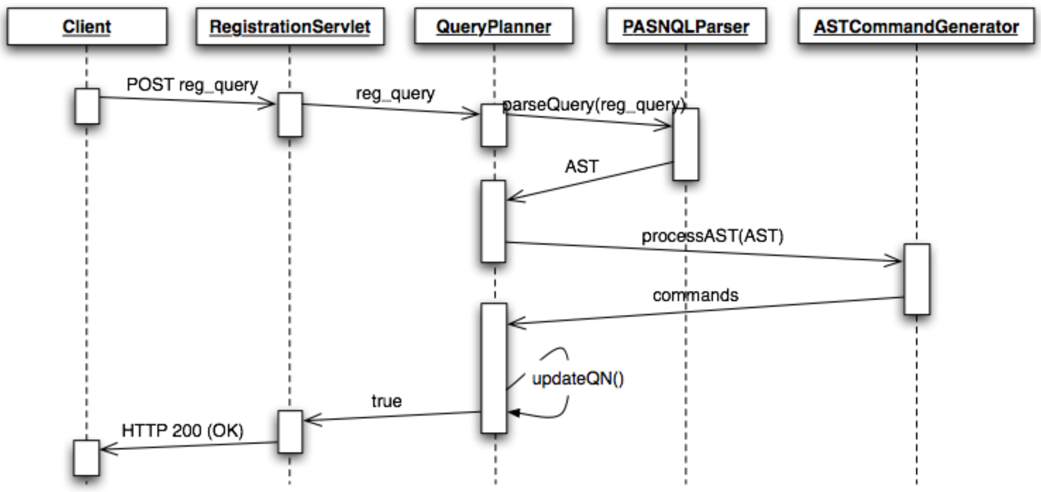

Figure 26: RegistrationServlet interaction sequence diagram

turns an error message to the user, which describes what went wrong.

Requesting node data works somewhat different. The data servlet only accepts show queries. When a show command has been parsed by the ASTCommandProcessor, a StreamSink is instantiated that is able to create a result that conforms to the format that was requested in the show query. If there is such a sink, the data servlet calls the $handleRequest()$ method that every sink should implement. The options specified by the user are sent to that method call as parameters. Finally, the sink returns a byte array of data, which is then directly written to the output stream. The reason for returning a byte array instead of plain text is that this approach will also enable transferral of binary data like images and PDF files. If a sink cannot handle the request made, an error message is returned instead.

6.3 Testing & Validation

The prototype has been tested by translating the properties described in section 4.1.2 to JUnit tests. To validate the functionality of the system, the rest of this section will list the requirements that were found throughout the thesis and to what extent the prototype fulfills these requirements.

6.3.1 Fulfilled User Requirements

The following requirements were described in section 2.4.3:

RE1: The system should be able to make selections on the data, such as time intervals, position intervals and temperature intervals.

Selections are supported on two levels. On the highest level, running GSN instances can communicate data to each other and make selections if needed. Finally, the prototype is also able to make selections by registering Selector nodes. Selections are always intervals. An interval can be a range of numbers, strings (lexicographic range) or timestamps, but it can also be a sequence of values.

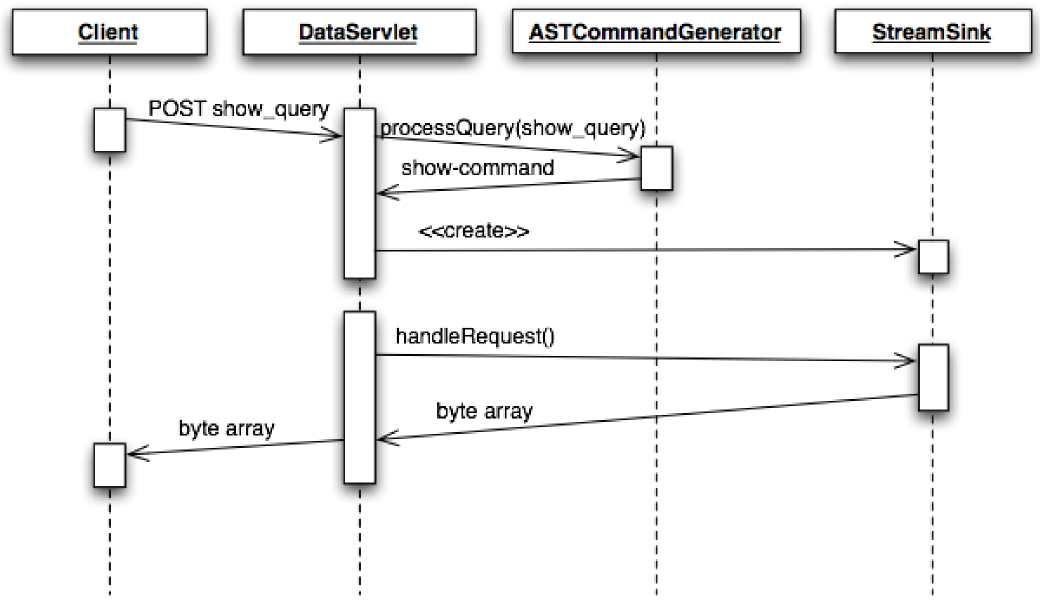

Figure 27: DataServlet interaction sequence diagram

RE2: The system must enable the user to process data in real-time (ex. sewer leakage)

The prototype has been built to efficiently process streaming data. It is multithreaded to be able to process new data as soon as it comes in and it doesn't need to wait for nodes to finish processing while new data is arriving. Another benefit of the multithreaded implementation is that it will be able to use multiple cores to process the data.

RE3: There must be a straightforward way to query the network. Most researchers have only little knowledge of computer science

The PASN-QL is a custom query language that has been built to ease query specification. It's simple syntax will make it easier to specify queries by hand, but it will also simplify the creation of a GUI capable of composing these queries.

RE4: The system should be able to store static metadata (fixed sensor information) as well as variable metadata (sensor configuration, annotations)

Metadata is handled by the Tupelo OPM service. All nodes in the prototype send provenance information to that service upon initialization. The advantage of using an external service for this, is that other applications can also use the service for recording and querying metadata. For example, a node can record that it is using a certain GSN Virtual Sensor as its source, but the GSN instance can also record what it knows about the data that it has generated itself, thus extending the provenance graph created by the node in the prototype.

6.3.2 Fulfilled Provenance Requirements

The provenance requirements listed below were described in section 3.7:

RE5: The provenance service should be a web service, which supports recording and querying provenance data

Provenance can be recorded and queried using the Tupelo OPM service, which can be accessed over HTTP and supports simple GET and POST requests.

RE6: Nodes in the sensor network should send provenance information to that service upon (re-)initialization.

Upon registration or deserialization of nodes in the network, the provenance info is sent to the Tupelo service. The service ensures that this provenance is not recorded twice, but if some information for a given URI differs from the information passed, the information is updated.

RE7: To enable to record the state of a process, for example the sensor configuration at a certain time, it should be possible to add annotations to processes within the provenance graph of the network.

The Tupelo service allows creation of annotations to processes. These annotations are passed when the provenance graph is returned. Annotations consist of a subject, a predicate and an object, of which the first two are URI's. Annotation predicates can have standard URI's (for example Dublin Core), but it also supports custom predicate URI's.

RE8: Changes in the network should be recorded by using timed annotations.

At the moment, the Tupelo service does not support timed annotations. The concept of timed annotations is part of the Open Provenance Model, but this has not been integrated yet in the Tupelo2 library.

6.3.3 Fulfilled Query Language Requirements

A description of the following requirements can be found in section 5.1, where the common requirements of a sensor network query language were described:

QL-RE1: Support for interval selections over multiple attributes

PASN-QL supports several specifying different interval specifications over multiple attributes. As the query network supports different kinds of intervals, the QL supports definitions of the intervals available in the network:

Range intervals

- value => (1..5]
- device => ['bt001'..'bt100')
- timed => ('01.01.2008 12:00:00'..'01.01.2009 20:30:00')

Sequences

- channel => (1,2,3,4)
- device => ['bt001','bt044','ct010']
- mixed => [1,'bt044',3,'ct010']

QL-RE2: Support for fixed, landmark and sliding windows

The query language supports the most common windows, namely sliding windows (time or tuple based), time based landmarks and time based fixed intervals. Examples:

- window => { noop5 => (MAX_ID-5..MAX_ID), dts1 => (NOW-10m..NOW) }
- window => ['01.01.2009 12:00:00'..NOW)
- window => ['01.01.2009 12:00:00'..'01.01.2009 20:30:00']

Some window definitions are however not available yet: tuple based landmarks, tuple based fixed intervals and reverse landmarks (left-relative landmarks). These should be implemented in a future version.

QL-RE4: Support for aggregations

Aggregations should be handled by a separate class of processing elements, but these have not been included in the prototype. Enabling users to specify aggregations in queries is not essential though; if needed, GSN instances can be created that perform this task beforehand.

QL-RE5: Support for stream merging

It is possible to specify unions in PASN-QL. A union will instantiate a Union PE. Example:

union1 = union(table1|table2, trigger => LAST < MAX_SID-10, window => { table1 => [NOW-1h..NOW], table2 => [NOW-15m..NOW] });

QL-RE6: Support for joining streams

Joins can be specified in a similar manner as a union, but it also needs to know the join attribute and the *delta*, which specifies how far apart two values can be until they are considered not to be matching tuples.

join1 = join(table1|table2, on => timed, delta => 5s, trigger => LAST < NOW-5m, window => { table1 => [NOW-1h..NOW], table2 => [NOW-15m..NOW] });

QL-RE7: Support for querying sensor data through annotations

Annotations are considered to be attributes that can be joined with the sensor data. Therefore, annotations can be queried by specifying a Selector node that contains interval selections on annotation attributes.

QL-RE8: Support for output structure transformations

In a show query, the last (third) argument is always a hash that can contain an arbitrary number of options. The output of a sink node in the network is based on these options. Some obvious transformation options are the output format and the order of the data, but more sinks could be created that handle more complex transformations like transpositions or conversions into diagrams.

QL-RE9: Pagination support

Pagination is handled by sink nodes as well. Like the previous point, pages can be specified by passing pagination options:

show(node1,'data', page => 2, perPage => 25);

QL-RE10: Limit expressiveness for easy querying

Regarding selections, PASN-QL limits the expressiveness by stating that it should *always* be specified as an interval. NOT and OR operators are not supported, but it is still simple to get the same result by using sequences or by chaining/combining Selector nodes. On the other hand, the options hash that can be passed to a sink node gives a lot more freedom to the user. The downside of this is that required and optional options should be documented somewhere; the language parser itself cannot decide that the query is incorrect. However, the advantages of allowing this does have a couple of benefits:

- New sink nodes can be created that come with their own set of options, which can be passed without altering the language;
- Options can be left out, in which case the system chooses a default value (applying convention over configuration). A query can become much shorter this way, saving the user some time while still understanding what the user is requesting from the system.

Based on these advantages, the choice was made to put this options hash in the query language.

RE18: QL: Support for querying provenance

Sink nodes in the network can connect to nodes in the network, but there are also sinks available that connect to the provenance service. A user can specify a Show query that returns the provenance graph of a node instead of the data itself. This depends on the second argument, which can (for this version) be either 'data' or 'provenance_graph'. Conclusively, to get the provenance graph for a given node, a user could specify the following query:

show(node1,'provenance_graph', format => 'image');

Chapter 7

Conclusion

7.1 Results

The thesis has shown that there are a lot of aspects that can to be taken into account when creating a workflow for real-time sensor data processing. This ranges from the requirements for online retrieval of sensor data to the requirements that need to be met in order for the end user to analyze the retrieved data. By describing the whole workflow from the beginning to the end, a clear list of requirements has originated that can be used as reference material for the development and assessment of real-time processing systems. It should be noted though that these requirements have mostly followed by combining requirements found in other literature and systems. It may be that there are additional (domain specific) requirements, or that some requirements are not applicable in different situations. The main purpose is to give insight in the common ideas that people have regarding stream processing workflows.

As a special case, the concept of provenance has been taken into account when designing the infrastructure. It has shown that provenance can successfully be combined with stream processing solutions. Being able to record provenance increases the options to share the data with other users and systems, which is a key factor of the Semantic Web [54].

The research shows that there is a common notion of how stream processing workflows should look like, namely a network of processing nodes that use each others output to gradually yield more meaningful results. In chapter 4, an attempt has been made to formalize this behavior for the proposed infrastructure. This formalization not only gives better understanding of the process, but it also enables the formulation of properties of the process, which can be used for validation.

The case study was a very important part of this research and has revealed some interesting findings. First, it has shown that just the availability of raw sensor data is often not enough for researchers to be of good use. Second, an intuitive way of exposing the data is very important. If a user does not understand how the recorded data can be queried, it will seldom be used at all. A key observation here is that what is *intuitive* strongly depends on the people involved. Within the field of computer science people will usually favor a query language similar to SQL, but in other fields a more functional language or even a graphical user interface may be more suitable. It is therefore important to know the people that will be using the system. Although the survey in chapter 5 was rather inconclusive, the designed and implemented query language has the advantage that it has resemblances with the query language of statistical tools like MatLab and R. Also, it provides *one* way for querying sources that have their own communication mechanisms, taking the need away to learn the query language of every system used. The actual effect of such a custom query language has however not been proven in this thesis, future research should confirm that it positively influences the effectiveness of sensor networks.

Finally, the prototype has shown that the proposed infrastructure provides an effective way of processing sensor data in real-time from multiple sources while retaining process provenance for reproducibility and justification of data analysis results. The architecture and implementation explain the concepts explored during this research very well and they clearly show which steps and decisions should be taken to apply those concepts.

7.2 Contribution

This research has lead to the following contributions in the field of sensor networks and stream processing:

- The chapters in this thesis can be used as reference material for the design and validation of sensor networks. It contains a clear listing and explanation of requirements for stream and query processing of sensor data.
- The thesis gives a nice overview of currently available sensor middleware and their features. Until now, most information regarding existing sensor middleware was outdated or scattered over multiple publications.
- The use of semantic information in sensor networks is not new, as this has already been done in the SONGS system (see section 2.2.4.2) [23], be it in a different way. However, this is one of the first publications that applies the concept of provenance in sensor networks by using the fairly new proposed OPM standard.
- The prototype is the first implementation of the stream processing workflow defined by [43]. It shows that the workflow can efficiently handle streaming data in a multithreaded environment. Moreover, it provides a good basis for future work in the field of sensor data management at the university, as the architecture can easily be extended to support new functionality.

7.3 Future Work

The research done in this thesis gives several possibilities for future research.

7.3.1 Middleware Research

- More research can be done by extending the list of requirements and assessing more software packages. It would also be interesting to know how the assessed open source packages compare to commercial solutions.
- It would be nice to know the performance of the several available systems. What effects this performance? Several possibilities come to mind, like the communication protocol used, the language in which it has been developed, the processing approach, the communication with the database backend, and so on. One of the reasons for

the fact that GSN was used in this research was that the use of SOAP and the OGC standards seemed to imply a lot of overhead in exchanging sensor data between processing systems, but this decision was not based on solid evidence.

7.3.2 Provenance

- Two solutions were proposed for recording changes in a provenance graph when the graph of processing nodes changes, namely by using timed annotations on provenance graph arcs or by introducing workflow state checkpoints. The validity of these approaches is not ensured, and there may be even better solutions. Future work could entail to find out how to effectively store provenance graphs that change over time.

7.3.3 Query Processing

- The formalization described is not complete yet. More processing elements could be distinguished, like the additional processing elements that are used by System S: Barrier, Split and Punctor.

- Only simple query processing optimizations were defined, which are based on subsumption of new nodes in existing nodes. A lot more research is still to be done on optimization of queries and the query processing tree. This ranges from defining and detecting possible optimizations to the implications of the workflow implementation to support such graph transformations.

7.3.4 Query Language Design

- An interesting research topic is to find out to what extent the design of a custom query language positively affects the use of it by users with only little knowledge of computer science.

- The query language built for the case study, PASN-QL, currently has some limitations:

  - It does not support the definition of new kinds of processing elements.
  - There is no way for a user to destroy processing nodes.
  - The ‘data’ command does not allow source option parameters to be overridden.
  - Not all kinds of buffers are supported yet: missing are tuple based fixed buffers and landmarks, and ‘reverse’ landmarks (landmarks that are fixed on the right side). These were considered to be less important than the others, therefore these have not been implemented.

  In future work the language can however easily be extended to support the above features, thanks to the use of ANTLR and ANTLWorks, the IDE for ANTLR. The grammars shown in appendix B and C are the only files needed to build all code needed to parse PASN-QL queries.

7.3.5 Prototype

There are several open tasks that could be fulfilled in future research:

- Performance testing
- Prototyping of the system at EAWAG
- Extending the existing system:
  - Enable batch processing of data for querying historical data. Like the real-time processing workflow, the workflow for batch processing has been documented by [43] as well.
  - Implementation of new stream sources, sinks, processing elements and database systems.
  - Add query planning and optimization.
  - Caching mechanisms for data that is frequently queried.
  - Extend the tupelo2 provenance service to support timed annotations on provenance graph arcs.
  - Extend the tupelo2 provenance service to support overlapping accounts that for example explain the inner functionality of processing nodes in more detail.
  - Integration with web applications like for example the SwissEx wiki.
  - Implement the GSN instance discovery service to be able to automatically use newly created instances.

Acknowledgments

I would like to thank my supervisor Andreas Wombacher for guiding me all these months. I am very grateful he has given me the opportunity to do this joint research, of which the trips to EAWAG were a great experience. Moreover, his own interest and expertise in the field of sensor networks was a great help in understanding the concepts involved and coming up with solutions to the problems described.

Big thanks also go to my second supervisor Philipp Schneider, Tobias Vogt and the other researchers at the WUT department for their support and collaboration during my stay in Switzerland. Without their help, this research would not have been possible.

Personally, my stay in Zürich would not have been such a memorable experience without the great cooking efforts of Philipp Schneider, Sushanth Tiruvaipati, Jakob Helbing and Marie Madeleine Paré. Thank you for some good evenings!

Last but not least, I would like to thank my parents for all their support during my college years and these last months. They never stopped believing in me and I hope this achievement will cheer them up during these stressful times.

Appendices

Appendix A

Query Language Survey

```
=======================================
=======================================
EAWAG Query Network Language Survey
=======================================
=======================================

Note: Multiple answers are possible for questions with
answers in square brackets ('[]').


=============================================
 Background information
=============================================

1. What is your main research area?
( ) Hydrology
( ) Chemistry
( ) Geophysics
( ) Ecology
( ) Microbiology
( ) Soil science
( ) Other: ...


2. Do you have some experience with programming languages?
[ ] Java
[ ] C#
[ ] C/C++
[ ] Ruby
[ ] Python
[ ] Matlab scripts
[ ] R scripts
[ ] Bash
[ ] Perl
[ ] PHP
[ ] Other: ...


3. Do you have some experience with the following query languages?
[ ] SQL
[ ] SPARQL
[ ] Wiki Query Language
[ ] XQuery
[ ] Xpath


=============================================
 First query example
=============================================


Please consider the following query (explained in detail during the presentation):

GET timed, position, value OF Hourly_Avg_Dts1
INCLUDE class ANNOTATIONS OF Temp_Classifier_Dts1
LIMIT RESULTS TO channel IN [1..1] AND value IN (3..7)
AND ANNOTATION class IN ("cold","hot")
LIMIT WINDOW TO (NOW-1h..NOW+1h]
FORMAT RESULTS AS CSV WITH timed AS "Measure date", position AS "point",
value AS "Temperature (C)"
ORDER BY timed ASC


4. Do you find that this is a clear way to request sensor data?
( ) Very unclear
( ) Unclear
( ) Neutral
( ) Clear
( ) Very clear


5. Would you be able to formulate such a query yourself?
( ) Yes
( ) No


6. Which parts of the query are unclear to you?
[ ] Nothing
[ ] The overall structure of the query
[ ] The projection: GET timed, position, value OF Hourly_Avg_Dts1
[ ] Inclusion of annotations: INCLUDE class ANNOTATIONS OF
     Temp_Classifier_Dts1
[ ] Data selection: LIMIT RESULTS TO channel IN [1..1] AND value IN [3..7]
   AND ANNOTATION class IN ["cold","hot"]
[ ] Windowing: LIMIT WINDOW TO [NOW-1h..NOW+1h]
[ ] Output format: FORMAT RESULTS AS CSV WITH timed AS "Measure date",
  position AS "point", value AS "Temperature [C]"
[ ] Ordering the data: ORDER BY timed ASC
[ ] Other: ...


7. This example is very verbose, meaning that it contains a lot of extra words to
improve its readability. What do you think of this level of verbosity?
( ) Too verbose
( ) A little too verbose
( ) Neutral
( ) Like it a little
( ) I prefer this level of verbosity


=============================================
 Second query example
=============================================


Please consider the following query (explained in detail during the presentation):

data = Hourly_Avg_Dts1.get(timed|position|value, window => (NOW-1h..NOW+1h],
                           conditions => channel:[1..1], value:(3..7),
                           order =>  timed, ASC) +
Temp_Classifier_Dts1.annotation(class, conditions => class:("cold","hot"));

data.format('csv', labels =>  timed: "Measure date", position:"point",
value:"Temperature (C)");


8. Do you find that this is a clear way to request sensor data?
( ) Very unclear
( ) Unclear
( ) Neutral
( ) Clear
( ) Very clear


9. Would you be able to formulate such a query yourself?
```

```
( ) Yes
( ) No


10. Which parts of the query are unclear to you?
[ ] Nothing
[ ] The overall structure of the query
[ ] The projection: Hourly_Avg_Dts1.get[timed|position|value, ...]
[ ] Inclusion of annotations: + Temp_Classifier_Dts1.annotation[class,
    conditions => class:["cold","hot"]]
[ ] Data selection: conditions => channel:[1..1], value:[3..7]
[ ] Windowing: window => [NOW-1h..NOW+1h]
[ ] Output format: data.format['csv', labels =>  timed: "Measure date",
     position:"point", value:"Temperature [C]"
[ ] Ordering the data: order =>  timed, ASC
[ ] Other: ...

11. This query example is very concise, which makes it shorter but also less
readable. What do you think of this conciseness?
( ) Too concise
( ) A little too concise
( ) Neutral
( ) Like it a little
( ) I prefer this level of conciseness

=============================================
 Questions for both examples
=============================================

12. The first query example is very verbose and human readable, while the other
one is mathematical and concise. Which notation do you prefer?

( ) The verbose notation
( ) The mathematical notation


13. Why do you prefer that notation over the other?
[ ] I prefer its verbosity
[ ] I prefer its conciseness
[ ] It is the most logical way
[ ] It is easier to understand for me
[ ] The notation looks familiar
[ ] The other notation does not appeal to me
[ ] Other: ...
[ ] Other: ...
```

The examples described use the same interval notation:

- An interval is specified by specifying the left and right boundary, separated bytwo dots: (1..5]
- Inclusion of a boundary itself can be specified by square brackets; in the interval (1..5], the right boundary 5 is included in the interval
- Exclusion of a boundary can be specified by parenthesis; the left boundary 1 of (1..5] is not included in the interval
- The interval (1..5] therefore represents the set { 2,3,4,5 }.

14. Are you familiar with the interval notation that has been used?
( ) Yes
( ) No

15. Does this notation seem to be a logical choice to you?
( ) Yes, because it is the usual way of defining intervals
( ) Yes, because it is easily understandable
( ) No, because it doesn't follow the mathematical standard
( ) No, because I use a different notation for my research
( ) No, because the notation doesn't make sense to me
( ) No, because an SQL BETWEEN clause seems more logical to me

```
=============================================
 Thank You!
=============================================
```

16. Thank you for filling in this survey! Now that you understand what is possible, do you have any final ideas/remarks?

...
...
...

Appendix B

ANTLR V3 Lexer/Parser Grammar

```
grammar pasnql;

options {
    output=AST;
}

@rulecatch {
       catch ( RecognitionException ex ) {
           //System.err.println("Syntax error at or near token '" + ex.token.getText() + "' (line " +
ex.line + ", column " + ex.charPositionInLine + "). Please verify your query is correct and try again!");
           //throw new RuntimeException(getErrorHeader(ex) + " : " + getErrorMessage(ex, tokenNames), ex);
           throw new RuntimeException("Syntax error at or near token '" + ex.token.getText() + "' (line " +
ex.line + ", column " + ex.charPositionInLine + "). Please verify your query is correct and try again!");
       }
   }

query : command+ ;

command : alias=IDENTIFIER ASSIGN assign_method SEMICOLON -> ^(ASSIGN_CMD $alias assign_method)
| show_method SEMICOLON!
;

show_method : 'show' LPAREN pe=IDENTIFIER COMMA out=STRING (COMMA opts=non_bracket_hash)? RPAREN
-> ^(SHOW_CMD $pe $out $opts?);

assign_method  : 'data' LPAREN pe=IDENTIFIER (COMMA 'attributes' ARROW attributes)? (COMMA conditions)? RPAREN
-> ^(DATA_CMD $pe attributes? conditions?)
| 'project' LPAREN pe=IDENTIFIER COMMA 'attributes' ARROW attributes (COMMA window_predicates)? RPAREN
-> ^(PROJECT_CMD $pe attributes window_predicates?)
| 'select' LPAREN pe=IDENTIFIER COMMA conditions (COMMA window_predicates)? RPAREN
-> ^(SELECT_CMD $pe conditions window_predicates?)
| 'union' LPAREN IDENTIFIER (PIPE IDENTIFIER)+ (COMMA window_predicates)? RPAREN
-> ^(UNION_CMD IDENTIFIER+ window_predicates?)
| 'join' LPAREN pe1=IDENTIFIER PIPE pe2=IDENTIFIER COMMA join_predicate (COMMA window_predicates)? RPAREN
-> ^(JOIN_CMD $pe1 $pe2 join_predicate window_predicates?);

attributes : IDENTIFIER (PIPE IDENTIFIER)* -> ^(ATTS IDENTIFIER*);

conditions : 'conditions' ARROW hash -> ^(CONDS hash);

join_predicate : 'on' ARROW v=IDENTIFIER (COMMA 'delta' ARROW delta_value)?  -> ^(JOIN_PRED $v delta_value?);

delta_value : val=NUMBER time_letter? -> ^(DELTA $val time_letter?);

window_predicates : window_kv_pair (COMMA! trigger_kv_pair)?
  | trigger_kv_pair (COMMA! window_kv_pair)?;
```

```
window_kv_pair : 'window' ARROW window_value -> ^(WINDOW window_value);

window_value : window_interval | window_hash ;

trigger_kv_pair : 'trigger' ARROW 'LAST' op=COMPARISON_OPERATOR trigger_expression
-> ^(TRIGGER $op trigger_expression);

window_expression : 'NOW' (OPERATOR time_literal)? -> ^(WINDOW_EXPR 'NOW' (OPERATOR time_literal)?)
  | 'MAX_ID' (OPERATOR NUMBER)? -> ^(WINDOW_EXPR 'MAX_ID' (OPERATOR NUMBER)?)
  | STRING
  ;

trigger_expression : 'NOW' (OPERATOR time_literal)? -> ^(TRIGGER_EXPR 'NOW' (OPERATOR time_literal)?)
  | 'MAX_SID' (OPERATOR NUMBER)? -> ^(TRIGGER_EXPR 'MAX_SID' (OPERATOR NUMBER)?)
  | STRING
  ;

time_literal : NUMBER time_letter ;

hash : LCURLY! non_bracket_hash RCURLY! ;

non_bracket_hash : single_kv_pair (COMMA single_kv_pair)* -> ^(HASH single_kv_pair single_kv_pair*);

window_hash : LCURLY window_ival_kv_pair (COMMA window_ival_kv_pair)* RCURLY -> ^(HASH window_ival_kv_pair+ );

window_ival_kv_pair : pe=IDENTIFIER ARROW window_interval -> ^(KVPAIR $pe window_interval);

single_kv_pair : k=IDENTIFIER ARROW v=value -> ^(KVPAIR $k $v);

value : collection
| hash
| single_value
;

collection : left_bracket interval right_bracket -> ^(COLLECTION left_bracket interval right_bracket);

interval : single_value (RANGE single_value -> ^(RANGE_IVAL single_value single_value) | (COMMA single_value)*
-> ^(CSV_IVAL single_value+ ));

window_interval : left_bracket window_expression RANGE window_expression right_bracket
-> ^(RANGE_IVAL left_bracket window_expression window_expression right_bracket);

single_value : STRING | NUMBER ;

left_bracket : LPAREN | LBRACKET;

right_bracket :  RPAREN | RBRACKET;

time_letter: 'h' | 'm' | 's' ;

IDENTIFIER : LETTER (LETTER | DIGIT | '_')* ;
STRING : QUOTE (~'\'')* QUOTE ;
NUMBER : DIGIT+
  (
    ('.' DIGIT) => '.' DIGIT+
    |
  )
  ;
COMPARISON_OPERATOR : '<' | '<=' | '=';
OPERATOR : '+' | '-' ;
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+  { $channel = HIDDEN; } ;

fragment DIGIT: '0'..'9';
fragment LETTER: 'a'..'z' | 'A'..'Z';
```

Appendix C

ANTLR V3 Tree Walker Grammar

```
tree grammar pasnqlTree;

options {
  // We're going to process an AST whose nodes are of type CommonTree.
  ASTLabelType = CommonTree;
  tokenVocab = pasnql;
}

@header {
  import nl.utwente.ewi.stream.pasnql.commands.*;
  import nl.utwente.ewi.stream.pasnql.helpers.*;
  import nl.utwente.ewi.stream.intervals.*;
  import java.util.HashMap;
  import java.util.Map;
  import java.util.EmptyStackException;
  import java.text.ParseException;
}

// We want to add some fields and methods to the generated class.
@members {
  private ArrayList<Command> commandList = new ArrayList<Command>();

  public List<Command> getCommandList() {
return commandList;
  }

  // This just shortens the code for print calls.
  private static void out(Object obj) {
    System.out.print(obj);
  }

  // This just shortens the code for println calls.
  private static void outln(Object obj) {
    System.out.println(obj);
  }

  public static String strip(String value){
    return value.substring(1, value.length()-1);
  }

} // @members

query : command+ ;

command : ^(ASSIGN_CMD alias=IDENTIFIER cmd=assign_method) { ((AssignableCommand) $cmd.result)
.setName($alias.text); commandList.add($cmd.result); }
| cmd=show_method {commandList.add($cmd.result);}
```

```
;

show_method returns [Command result]
: ^(SHOW_CMD pe=IDENTIFIER out=STRING non_bracket_hash?) {
$result = new ShowCommand($pe.text, strip($out.text), $non_bracket_hash.result); };

assign_method returns [Command result]
: ^(DATA_CMD pe=IDENTIFIER attributes? conditions?) {
$result = new DataCommand($pe.text, $attributes.result, $conditions.result); }
| ^(PROJECT_CMD pe=IDENTIFIER attributes window_predicates?) {
$result = new ProjectCommand($pe.text, $attributes.result, $window_predicates.result); }
| ^(SELECT_CMD pe=IDENTIFIER conditions window_predicates?) {
$result = new SelectCommand($pe.text, $conditions.result, $window_predicates.result); }
| ^(JOIN_CMD pe1=IDENTIFIER pe2=IDENTIFIER join_predicate window_predicates?) {
$result = new JoinCommand($pe1.text, $pe2.text, $join_predicate.result, $window_predicates.result); }
| ^(UNION_CMD identifiers window_predicates?) {
$result = new UnionCommand($identifiers.result, $window_predicates.result); }
;

identifiers returns [List result]
scope { List current; }
@init { $identifiers::current = new ArrayList(); }
 : identifier+ { $result = $identifiers::current; }
    ;

identifier
: ident=IDENTIFIER { $identifiers::current.add($ident.text); }
;

attributes returns [List result]
scope { List current; }
@init { $attributes::current = new ArrayList(); }
 : ^(ATTS atts=attribute*){ $result = $attributes::current; }
    ;

attribute
: ident=IDENTIFIER { $attributes::current.add($ident.text); }
;

conditions returns [Map result]
: ^(CONDS hash) { $result = $hash.result; }
;

join_predicate returns [Map result]
: ^(JOIN_PRED v=IDENTIFIER delta_value?)
{
Map map = new HashMap();
map.put($v.text, $delta_value.result);
$result = map;
}
;

delta_value returns [double result]
: ^(DELTA val=NUMBER tl=time_letter?)
{
double dval = Double.parseDouble($val.text);
if(tl == null) $result = dval;
else  {
String tletter = $tl.text;
if(tletter.equals("s")) $result = dval * 1000;
else if(tletter.equals("m")) $result = dval * 1000 * 60;
else $result = dval * 1000 * 60 * 60;
}};

window_predicates returns [WindowPredicate result]
: w=window_kv_pair (t=trigger_kv_pair)?
```

```
{
WindowPredicate wp = new WindowPredicate();
wp.setBufferPredicates($w.result);
if(t != null) wp.setTriggerPredicate($t.result);
$result = wp;
}
| t=trigger_kv_pair (w=window_kv_pair)?
{
WindowPredicate wp = new WindowPredicate();
wp.setTriggerPredicate($t.result);
if(w != null) wp.setBufferPredicates($w.result);
$result = wp;
}
;

window_kv_pair returns [Map<String, BufferPredicate> result]
: ^(WINDOW w=window_value) { $result = $w.result; }
        ;

window_value returns [Map<String, BufferPredicate> result]
scope { Map<String, BufferPredicate> current; }
@init { $window_value::current = new HashMap<String,BufferPredicate>(); }
: w=window_interval { $window_value::current.put("", $w.result); $result = $window_value::current; }
| window_hash { $result = $window_value::current; }
;

trigger_kv_pair returns [TriggerPredicate result]
scope { TriggerPredicate current; }
@init { $trigger_kv_pair::current = new TriggerPredicate(); }
: ^(TRIGGER op=COMPARISON_OPERATOR trigger_expression)
{
$trigger_kv_pair::current.setCompOp($op.text);
$trigger_kv_pair::current.setExpression($trigger_expression.result);
$result = $trigger_kv_pair::current;
}
;

hash returns [Map<String, Object> result]
 : non_bracket_hash { $result = $non_bracket_hash.result; }
   ;

non_bracket_hash returns [Map<String, Object> result]
scope { Map<String, Object> current; }
@init { $non_bracket_hash::current = new HashMap<String, Object>(); }
: ^(HASH single_kv_pair single_kv_pair*) { $result = $non_bracket_hash::current; }
;

window_ival_kv_pair
: ^(KVPAIR pe=IDENTIFIER w=window_interval) { $window_value::current.put($pe.text, $w.result); }
;

single_kv_pair
: ^(KVPAIR k=IDENTIFIER v=value) { $non_bracket_hash::current.put($k.text, $v.result); }
        ;

value returns [Object result]
: collection { $result = $collection.result; }
| hash { $result = $hash.result; }
| single_value  { $result = $single_value.result; }
;

collection returns [Interval result]
 : ^(COLLECTION left_bracket interval right_bracket)
 {
Interval ival = $interval.result;
if($left_bracket.text.equals("(")) ival.setLeftBoundaryIncluded(false);
if($right_bracket.text.equals(")")) ival.setRightBoundaryIncluded(false);
$result = ival;
 }
 ;

interval returns [Interval result]
  scope { List values; }
@init { $interval::values = new ArrayList(); }
 : ^(RANGE_IVAL left=single_value right=single_value)
 {
  try {
// STRING
if($left.result instanceof String) {
// STRING TIMESTAMP ?
try {
$result = new TimeInterval((String)$left.result,(String)$right.result, true, true);
} catch(Exception e) {
$result = new StringInterval((String)$left.result,(String)$right.result, true, true);
}
}
//NUMBER
else $result = new NumberInterval((Double)$left.result,(Double)$right.result, true, true);
} catch(IntervalInvalidException ex){
throw new RecognitionException();
}
 }
 | ^(CSV_IVAL single_value+ )
 {
Sequence s = new Sequence($interval::values);
$result = s;
 }
 ;

single_value returns [Object result]
: STRING
{
 String val = strip($STRING.text);
 $result = val;
 try {
  $interval::values.add(val);
 } catch(EmptyStackException e){}
}
| NUMBER
{
 String val = $NUMBER.text;
 $result = Double.parseDouble(val.replaceAll(",","."));
 try {
  $interval::values.add(val);
 } catch(EmptyStackException e){}
}
;

window_interval returns [BufferPredicate result]
: ^(RANGE_IVAL left_bracket left=window_expression right=window_expression right_bracket)
{
BufferPredicate bp = new BufferPredicate();

boolean leftBoundaryIncluded = true, rightBoundaryIncluded = true;
if($left_bracket.text.equals("(")) leftBoundaryIncluded = false;
if($right_bracket.text.equals(")")) rightBoundaryIncluded = false;
WindowExpression lft = $left.result;
WindowExpression rgt = $right.result;

if(lft.getType2() == WindowExpression.Type.ABSOLUTE && rgt.getType2() == WindowExpression.Type.ABSOLUTE){
// FIXED (TIME) INTERVAL
bp.setType(WindowPredicate.Type.TIME_BASED);
bp.setType2(BufferPredicate.Type.FIXED);
try {
       bp.setValue(new TimeInterval((String) lft.getValue(), (String) rgt.getValue(),
leftBoundaryIncluded, rightBoundaryIncluded));
      } catch (ParseException ex) {
          throw new RecognitionException();
        }
}
else if(lft.getType2() == WindowExpression.Type.ABSOLUTE && rgt.getType2() == WindowExpression.Type.RELATIVE){
// LANDMARK
bp.setType(WindowPredicate.Type.TIME_BASED);
bp.setType2(BufferPredicate.Type.LANDMARK);
bp.setValue(lft.getValue());
}
else if(lft.getType2() == WindowExpression.Type.RELATIVE && rgt.getType2() == WindowExpression.Type.RELATIVE){
// SLIDE
bp.setType2(BufferPredicate.Type.SLIDE);

if(lft.getType() == WindowPredicate.Type.TIME_BASED && rgt.getType() == WindowPredicate.Type.TIME_BASED){
bp.setType(WindowPredicate.Type.TIME_BASED);
}
else if(lft.getType() == WindowPredicate.Type.TUPLE_BASED && rgt.getType() == WindowPredicate.Type.TUPLE_BASED){
bp.setType(WindowPredicate.Type.TUPLE_BASED);
}
else throw new RecognitionException();

bp.setOp(lft.getOp());
bp.setValue(lft.getValue());
}
else throw new RecognitionException();

$result = bp;
}
;

window_expression returns [WindowExpression result]
@init { WindowExpression we = new WindowExpression(); }
: ^(WINDOW_EXPR 'NOW' (op=OPERATOR time_literal)?)
{
   we.setType(WindowPredicate.Type.TIME_BASED);
   we.setType2(WindowExpression.Type.RELATIVE);
   if($op != null) {
   we.setOp($op.text);
   we.setValue($time_literal.result);
   }
   $result = we;
}
| ^(WINDOW_EXPR 'MAX_ID' (op=OPERATOR val=NUMBER)?)
{
we.setType(WindowPredicate.Type.TUPLE_BASED);
   we.setType2(WindowExpression.Type.RELATIVE);
   if($op != null) {
   we.setOp($op.text);
   we.setValue(Double.parseDouble($val.text));
   }
   $result = we;
}
| STRING
{
   we.setType(WindowPredicate.Type.TIME_BASED);
   we.setType2(WindowExpression.Type.ABSOLUTE);
   we.setValue($STRING.text);
   $result = we;
}
;
```

```
trigger_expression returns [TriggerExpression result]
@init { TriggerExpression te = new TriggerExpression(); }
: ^(TRIGGER_EXPR 'NOW' (op=OPERATOR time_literal)?)
{
te.setType(WindowPredicate.Type.TIME_BASED);
te.setType2(TriggerExpression.Type.RELATIVE);
if($op != null) {
   te.setOp($op.text);
   te.setValue($time_literal.result);
   }
   $result = te;
}
| ^(TRIGGER_EXPR 'MAX_SID' (op=OPERATOR val=NUMBER)?)
{
te.setType(WindowPredicate.Type.TUPLE_BASED);
te.setType2(TriggerExpression.Type.RELATIVE);
if($op != null) {
   te.setOp($op.text);
   te.setValue(Double.parseDouble($val.text));
   }
   $result = te;
}
| STRING
{
   te.setType(WindowPredicate.Type.TIME_BASED);
   te.setType2(TriggerExpression.Type.ABSOLUTE);
   te.setValue($STRING.text);
   $result = te;
}
;

time_literal returns [double result]
: val=NUMBER tl=time_letter
{
double dval = Double.parseDouble($val.text);
if(tl == null) $result = dval;
else  {
String tletter = $tl.text;
if(tletter.equals("s")) $result = dval * 1000;
else if(tletter.equals("m")) $result = dval * 1000 * 60;
else $result = dval * 1000 * 60 * 60;
}};

window_hash : ^(HASH window_ival_kv_pair+ );
left_bracket : LPAREN | LBRACKET;
right_bracket :  RPAREN | RBRACKET;
time_letter: 'h' | 'm' | 's' ;
```

Bibliography

[1] K. Aberer, M. Hauswirth, and A. Salehi, "A Middleware for Fast and Flexible Sensor Network Deployment," in *Proceedings of the 32nd international conference on Very large data bases*, pp. 1199–1202, 2006.

[2] 52North, *52North - Exploring Horizons*. `http://52north.org`, 2005.

[3] X. Chu, "Open Sensor Web Architecture: Core Services," Master's thesis, 2005.

[4] Y. Simmhan, J. Futrelle, B. Plale, S. Miles, C. Goble, P. Missier, and R. Barga, "The Open Provenance Model (v1. 01)," 2008.

[5] J. Selker, L. Thévenaz, H. Huwald, and A. Mallet, "Distributed fiber optic temperature sensing for hydrologic systems," *Water Resources Research, Vol. 42*, 2006.

[6] W. Horré, N. Matthys, S. Michiels, W. Joosen, and P. Verbaeten, "A survey of middleware for wireless sensor networks," 2007.

[7] L. Golab and M. T. Özsu, "Issues in data stream management," *ACM SIGMOD Record*, vol. 32, pp. 5–14, juni 2003.

[8] M. Botts, G. Percivall, C. Reed, and J. Davidson, "OGC Sensor Web Enablement: Overview and High Level Architecture (OGC 07-165)," 2007.

[9] A. Sheth, C. Henson, and S. S. Sahoo, "Semantic Sensor Web," *IEEE Internet Computing*, vol. 12, pp. 78–83, juli 2008.

[10] *The Book of GSN*, 2008.

[11] D. Abadi, Y. Ahmad, M. Balazinska, and U. Cetintemel, "The design of the borealis stream processing engine," in *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA*, 2005.

[12] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 1, pp. 1—-44, 2008.

[13] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik, "Retrospective on Aurora," *The VLDB Journal*, vol. 13, pp. 370–383, 2004.

[14] A. Na and M. Priest, "Sensor Observation Service," tech. rep., 2006.

[15] *A SensorWeb Middleware with Stateful Services for Heterogeneous Sensor Networks*, 2007.

[16] M. Corporation, *SensorMap*. http://atom.research.microsoft.com/sensewebv3/sensormap/, 2007.

[17] C. F. E. N. Sensing, *SensorBase*. http://sensorbase.org/, 2007.

[18] T. G. Computing and D. Systems, *The SensorWeb Project: Unifying Sensor Networks And Grid Computing With The World-Wide Web*. http://www.gridbus.org/sensorweb/, 2008.

[19] A. Salehi, *GSN*. http://sourceforge.net/apps/trac/gsn/, 2009.

[20] K. Aberer, M. Hauswirth, and A. Salehi, "Global Sensor Networks," tech. rep., Ecole Polytechnique Fédérale de Lausanne (EPFL), 2006.

[21] O. G. Consortium, *Sensor Observation Service (SOS) Introduction*. http://www.ogcnetwork.net/SOS_Intro, 2008.

[22] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, (New York), p. 1123, ACM Press, 2008.

[23] J. Liu and F. Zhao, "Towards semantic services for sensor-rich information systems," *2nd International Conference on Broadband Networks, 2005.*, pp. 44–51, 2005.

[24] K. Whitehouse, F. Zhao, and J. Liu, "Semantic streams: A framework for composable semantic interpretation of sensor data," *Lecture Notes in Computer Science*, vol. 3868, p. 5, 2006.

[25] P. Gibbons, B. Karp, and Y. Ke, "Irisnet: An architecture for a worldwide sensor web," *IEEE Pervasive Computing*, pp. 22—-33, 2003.

[26] O. G. Consortium, *Official OGC SOS Schema*. http://schemas.opengis.net/sos/, 2008.

[27] CCES, *RECORD*. http://www.cces.ethz.ch/projects/nature/Record, 2009.

[28] C. C. Environment and Sustainability, *The Swiss Experiment*. http://www.swiss-experiment.ch, 2009.

[29] T. Schmid and H. Dubois-Ferriere, "SensorScope: Experiences with a Wireless Building Monitoring Sensor Network," *Workshop on Real-World Wireless Sensor Networks (REALWSNâ05)*, 2005.

[30] I. Talzi, A. Hasler, S. Gruber, and C. Tschudin, "PermaSense: Investigating permafrost with a WSN in the Swiss Alps," *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, 2007.

[31] J. Beutel, S. Gruber, S. Gubler, A. Hasler, M. Keller, and R. Lim, "The PermaSense Remote Monitoring Infrastructure," *International Snow Science Workshop Davos 2009*, 2009.

[32] B. Glavic and L. Dittrich, "Data Provenance: A Categorization of Existing Approaches," in *12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW 2007)*, vol. 7, pp. 227—-241, 2007.

[33] R. Bose, I. Foster, and L. Moreau, "Report on the International Provenance and Annotation Workshop:(IPAW'06)," *ACM SIGMOD Record*, 2006.

[34] L. Moreau, *Provenance Challenge Wiki*. `http://twiki.ipaw.info/bin/view/Challenge/WebHome`, 2009.

[35] P. Groth, S. Miles, and L. Moreau, "PReServ: Provenance Recording for Services," in *Proceedings of the UK OST e-Science second All Hands Meeting*, University of Southampton [School of Electronics and Computer Science] [http://eprints.ecs.soton.ac.uk/perl/oai2] (United Kingdom), 2005.

[36] P. Groth, S. Jiang, S. Miles, S. Munroe, V. Tan, S. Tsasakou, and L. Moreau, "An Architecture for Provenance Systems," 2006.

[37] J. Ledlie, C. Ng, and D. Holland, "Provenance-Aware Sensor Data Storage," *21st International Conference on Data Engineering Workshops (ICDEW'05)*, pp. 1189–1189, april 2005.

[38] J. Widom, "Trio: A System for Integrated Management of Data, Accuracy, and Lineage," in *Proceedings of the 2005 CIDR Conference*, 2005.

[39] I. Foster, J. Vöckler, M. Wilde, and Y. Zhao, "Chimera: A virtual data system for representing, querying, and automating data derivation," *doi.ieeecomputersociety.org*, 2002.

[40] J. Gehrke and S. Madden, "Query processing in sensor networks," *IEEE Pervasive Computing*, 2004.

[41] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: a distributed, scalable platform for data mining," in *DMSSP '06: Proceedings of the 4th international workshop on Data mining standards, services and platforms*, (New York, NY, USA), pp. 27–37, 2006.

[42] D. Abadi, D. Carney, U. Cetintemel, and M. Cherniack, "Aurora: a data stream management system," *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 666—-666, 2003.

[43] A. Wombacher, "Data Workflow - A Unified Time Controlled E-Science Processing Model." 2009.

[44] H. Mailvaganam, *Introduction to OLAP*. `http://www.dwreview.com/OLAP/Introduction_OLAP.html`, 2007.

[45] U. S. A. Force, M. Keith, C. Ross, and E. Al., *Air University Sampling and Surveying Handbook: Guidelines for Planning, Organizing, and Conducting Surveys*. University Press of the Pacific.

[46] D. Watt and D. Brown, *Programming language processors in Java: compilers and interpreters*. Pearson Education, 2000.

[47] T. O. Group, *ArchiMate 1.0 Specification*. Van Haren Publishing, Zaltbommel, 2009.

[48] J. Futrelle, *Tupelo Server*. `http://tupeloproject.ncsa.uiuc.edu/`, 2009.

[49] S. M. Yacoub, "Composite Filter Pattern," 2001.

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: elements of reusable object-oriented software. 1995*. Addison-Wesley Publishing Co., 1995.

[51] T. Parr, J. Lilly, P. Wells, R. Klaren, M. Illouz, and J. Mitchell, "ANTLR reference manual," *MageLang Institute, document version*, 2003.

[52] T. Parr and J. Bovet, *Introduction to OLAP*. `http://www.dwreview.com/OLAP/Introduction_OLAP.html`, 2009.

[53] Nokia, *The URI Query Agent Model, a Semantic Web Enabler*. `http://sw.nokia.com/uriqa/URIQA.html`, 2008.

[54] J. Hendler, O. Lassila, and T. Berners-Lee, "The semantic web," *Scientific American*, 2001.