Global Sensors Networks[1]

GSN Team

November 30, 2008

Contents

Chapter 1

Introduction

1.1 Terminology

- **Global Sensor Networks** (*GSN*) defines both the project and the software described in this document.

- A **Wrapper** (*Wrapper*) is a piece of Java code that does the data acquisition for a specific type of device.

- A **Virtual Sensor** (*VS*) is the main component in *GSN*. It receives data from one or more *Wrapper*. It can combine their data, process and finally store it. A *VS* is defined in a single *VSD* and combines different pieces of software

  - One *VSP*
  - Zero or Many *Wrapper (s)*

- A **Virtual Sensor Description file** (*VSD*) is an XML file that contains the selection and the parametrization of the *VSP* and *Wrapper* that compose a *VS*. This file also contains the SQL statements that connect them together.

- A **Virtual Sensor Processing class** (*VSP*) is a piece of Java code that process and stores the data upon reception from the *Wrapper*.

1.2 Quick Start

GSN (for Global Sensor Networks) is a software project that started in 2005 at EPFL in the LSIR Lab by Ali Salehi, under the supervision of Prof. Karl Aberer. The initial goal was to provide a reusable software platform for the processing of data streams generated by wireless sensor networks. The project was successful, and was later reoriented towards a generic stream processing platform.

GSN acquires data, filters it with an intuitive, enriched SQL syntax, runs customisable algorithms on the results of the query, and outputs the generated data with its notification subsystem.

GSN can be configured to acquire data from various data sources. The high number of data sources in GSN allows for sophisticated data processing scenarios. In the unlikely event that your data sources are not supported, it is very easy to write a wrapper to make your hardware work with GSN (you can find more information about this in chapter 5).

GSN offers advanced data filtering functionalities through an enhanced SQL syntax. It is assumed that the reader has some knowledge of the Standard Query Language (SQL). Using it for basic operations is fairly intuitive and you should be able to start using it from the examples provided in this document.

1.3   Installing GSN from binary

Due to the quick development cycle of GSN, you should install the latest version. It can always be found at http://gsn.sourceforge.net/download/.

Before installing GSN, please download the latest version (at least version 6) of the Java Development Kit from http://www.java.com. The default settings should be fine.

At the time of writing (14/2/2008), the version of the installer is 0.95 is dated 15/04/2007. The installer includes a Windows batch file, gsn-win-nogui.bat and a unix shell script, gsn-unix-nogui.sh, which will run the GSN server (com- mand line - no gui). You may need to modify the supplied configuration file conf/gsn.xml to select the database that you will be using. To use the in memory database ensure that the following line is uncommented:

```
\texttt{\begin{math}<\end{math}storage user="sa" password=""
   driver="org.hsqldb.jdbcDriver" url="jdbc:hsqldb:MEM:."/\begin{math}>\end{math}}
```

Listing 1.1: Database configuration in GSN.

This version includes no tools to run the GSN graphical user interface.

1.4   Installing GSN from source

If you wish to run the most recent version of GSN (or run the GUI), then you will need to run the source code version which is available from the SVN repository at http://gsn.svn.sourceforge.net/viewvc/gsn/.

1.4.1   To run from a command line

- Download the Jakarta apache ant version 1.7.x or higher.

- Add the full pathname to the ANT_HOME/bin folder to your PATH

- Ensure you have the latest version Sun JDK installed.

- Download and install TortoiseSVN (for Windows) or SmartSvn (OS independent) or a command line SVN client:

- svn co https://gsn.svn.sourceforge.net/svnroot/gsn/trunk gsn

- Check out the GSN source code from link - https://gsn.svn.sourceforge.net/svnroot/gsn

Task	Name Description
gsn	Starts the GSN server.
restart	Stops any running GSN server and starts it again.
stop	Stops the currently runing GSN server.
gui	Starts the GSN graphical user interface.
jar	Creates a jar ⎽le from the source.
clean	Removes the current build files and forces a rebuild. You may need to run this task once you TODO Something missing in existing pdf
cleandb	Removes the redundant tables which are create for holding GSN's internal states.

Table 1.1: List of Ant tasks for GSN.

To run any of the aforementioned ant tasks simply write: ant task_name.

1.4.2 Installing to Run and debug GSN in Eclipse.

The GSN code can also be installed to run and debug in the Eclipse environment.

- Download and install Eclipse SDK

- Start Eclipse;

- Download and install the Subclipse[1] (http://subclipse.tigris.org/install.html);

- File -> Import -> Other -> Checkout Projects from SVN;

  - Check "Create a new repository location";
  - Paste the repository location "https://gsn.svn.sourceforge.net/svnroot/gsn";
  - Select "trunk" and click "Next";
  - Select "Check out project configured using the New Projects Wizard" and click "Finish";
  - In the New Projects Wizard select "Java Project" and click "Next";
  - In the New Java Projects Wizard
    * enter the project name, select "Create new project in workspace
    * select "Create separate folders . . ." and click on "Configure default:
    * if the project doesn't contains a "src" directory (depends on the Eclipse version you are using), assure to create one by clicking on the "Create new source folder"
    * In Build Path preferences select "Folders", enter "build/classes" as the output folder name and click "OK:'
    * click "Finish";
  - Click "Ok" to confirm overwrite of non standard resources;
  - Wait for the files to download from the repository;

- To add library files to the build path:

  - Project -> Properties
  - In the Properties dialog select "Java Build Path";

---

[1]This guide works for Eclipse 3.2, for Eclipse 3.4 the steps have changed slightly.

– In the Java Build Path dialog, select the "Libraries" tab

– On the Libraries tab, click on "Add jars . . .";

– Add only the .jar files in the lib directory and its subfolders.

GSN is now ready to Run.

Refer also to the "How to Install Eclipse, Subclipse and GSN, Step-by-step walkthrough" on the GSN documentation page at: http://gsn.sourceforge.net/documentation/.

1.4.3 Configuring eclipse to run/debug GSN through Eclipse.

Step 1: Setting the Ant Home :

- Download the Ant binaries (apache-ant-1.7.x-bin.zip) from http://ant.apache.org/;

- Extract the folder apache-ant-1.7.x to a suitable location on your hard drive;

- Open Eclipse and do the following steps to set the Ant Home:

  – Go to Window Menu and select Preferences, on the left side, click on ANT and then select Runtime.

  – In the Classpath tab (opened on the center of the window) select Ant Home Entries, click on the Ant Home button and browse toward the directory that contains the files from the jakarta-ant archive (\apache-ant-1.7.x\bin).

  – Click on OK in the Browse window and again to exit the Preferences dialog.

Step 2: Setting an Ant Build System for your GSN project :

- Select the GSN project and on the Project menu click on properties tab.

- On the Properties sheet for the project, select Builders on the left side and click on "New" and select "Ant Builder".

- A "Builder Properties for . . . " window pops up which has several tabs :

  – Main Tab :
    * For build file, click on Browse Workspace and select the build.xml file
    * For base directory, click on the Browse Workspace and click on the project name which contains the gsn source code.
    * Leave the "Set an Input Handler" selected.

  – Targets Tab :
    * For After a Clean, click on Set Targets and select both Build and Bind.
    * Click on Ok.

  – Back in the Builders page
    * Select the new Ant Builder
    * De-select the existing Java Builder and click OK in the confirmation panel which will appear.
    * Move the new Ant Builder to the top of the list.

  – Click on OK.

1.4.4 Trying it out

- Build the project.

- Set the gsn-controller-port parameter:

  - Open build.xml, locate the gsn-controller-port value and copy it to the clipboard;
  - Open the Run dialog (Run -> Open Run Dialog . . .);
  - In the target Run Configuration (eg. Main):
    * Go to the "Arguments" tab;
    * Paste or type the gsn-controller-port value from build.xml;
    * Click "Close"
  - Click on the Run button on the toolbar
  - The GNS application should display "GSN Starting . . . " in the console;
  - Open a web browser and browse to http://localhost:22001 and verify that the GSN server is working.
  - Stop the running GSN using the ant Stop task;
  - Insert a breakpoint in the first line of the Main class;
  - Start the application from the "Debug" button on the Eclipse toolbar;
  - GSN should start in the Eclipse Debug perspective and pause at the breakpoint;

Now you can debug your virtual sensors in eclipse, try it by setting a break point on a line in the main file.

Chapter 2

GSN Architecture

2.1 Data Acquisition

Before filtering and processing data, GSN needs to receive it. GSN considers two types of data sources: event-based and polling-based. In the first case, data is sent by the source and a GSN method is called when it arrives. Serial ports, network (TCP or UDP) connections, wireless webcams fall in this case. In the latter one, GSN periodically asks the source for new data. This is the case of an RSS feed or a POP3 email account.

2.2 Safe Storage

Some acquisition systems do not internally store the data produced until GSN can process it. For instance the TinyOS Serial Forwarder simply sends a copy of the incoming messages to all its alive listeners. If GSN is down at that time, the data produced will be lost. Safe Storage is specialy developed to handle that case. Safe Storage runs in a separate process and it aims is to simply store persistently the incoming data until the GSN process can ask for them.

These two processes communicate together upon TCP sockets and could run on different machines. The Safe Storage process acts as a server and GSN processes as clients. This communication and the primary storage increase the delay between the data production and the final storage into GSN. This results in a tradeoff between the performances and the reliability. Moreover, GSN can run both standard wrappers and Safe Storage wrappers at the same time.

2.2.1 Software Architecture

Safe Storage is generic and could be added to all the already existing wrappers. However small modifications need to be done since the Safe Storage wrappers are split into the two following parts

- The acquisition part that runs in the Safe Storage process

- The processing part that runs in the GSN process

The wrapper class that executes on the Safe Storage should be kept as simple as possible and must extend the `gsn.acquisition2.wrappers.AbstractWrapper2` abstract class. An

implementation of this class must define the methods shown on the Listing 2.1.

```
public abstract boolean initialize ()
public abstract void finalize ()
public abstract String getWrapperName ()
public abstract void run()
```

Listing 2.1: Methods to implement for a Safe Storage Wrapper - Safe Storage Side

The class that executes on the GSN side should do the data processing and must extend the `gsn.acquisition2.wrappers.SafeStorageAbstractWrapper` abstract class. An implementation of this class must define the methods shown on the Listing 2.2

```
public DataField[] getOutputFormat()
public boolean messageToBeProcessed(DataMsg dataMessage)
```

Listing 2.2: Methods to implement for a Safe Storage Wrapper - GSN Side

2.2.2 Safe Storage configuration

All the wrappers that are based on Safe Storage must set the list of parameters defined in Listing B.7. The Safe Storage and Safe Storage controller port are defined in the `build.xml` file.

```
<property name="safe-storage-port" value="25000"/>
<property name="safe-storage-controller-port" value="25012"/>
```

Listing 2.3: Safe Storage ports

The Listing 2.4 shows an example of *VSD* that use a Safe Storage *Wrapper*.

```
<virtual-sensor name="ss_mem_vs" priority="10" >
   <processing-class>
      <class-name>gsn.vsensor.BridgeVirtualSensor</class-name>
      <output-structure>
         <field name="heap_memory_usage" type="bigint" />
         <field name="non_heap_memory_usage" type="bigint" />
         <field name="pending_finalization_count" type="int" />
      </output-structure>
   </processing-class>
   <description> Mem VS for Safe Storage Test </description>
   <life-cycle pool-size="10" />
   <addressing />
   <storage history-size="10"/>
   <streams>
      <stream name="data">
         <source alias="source" storage-size="1" sampling-rate="1">
            <address wrapper="ss_mem_processor">
               <predicate key="ss-host">localhost</predicate>
               <predicate key="ss-port">25000</predicate>
               <predicate key="continue-on-error">true</predicate>
               <predicate key="wrapper-name">mem2</predicate>
            </address>
            <query> select * from wrapper </query>
         </source>
         <query> select * from source </query>
      </stream>
   </streams>
</virtual-sensor>
```

Listing 2.4: Sample of Email Notification VSD file

2.2.3 Use Cases

When a virtual sensor that use a Safe Storage *Wrapper* is loaded into GSN, a request is made to Safe Storage for the *Wrapper*. If the *Wrapper* is not already created[1] this will create, initialize and run a new instance of the *Wrapper* in the Safe Storage process. If the *Wrapper* was already running, it will be reused.

If the GSN process fails or is shutted down, The Safe Storage *Wrapper* will continue to acquire the data. Once the GSN process is restarted, all the data will be fetched from the Safe Storage DB.

If the Safe Storage process fails or is shut down, the data are not stored anymore and the GSN process will try periodically to connect to the Safe Storage process until the latter has been restarted. When you restart the Safe Storage process, it will automatically create and start acquiring data for the wrappers that were loaded before.

2.2.4 Safe Storage Management

Safe Storage ANT Tasks	
Task Name	**Description**
start-acquisition	Start the Safe Storage process. The wrapper that were loaded during the last runs will be automatically resumed and will directly start acquiring data.
clean-acquisition	Delete all the Safe Storage permanent storage and flush the list of Wrappers to resume. Use this task with caution since it may delete unprocessed data.
stop-acquisition	Stop the Safe Storage process.
Use each of these tasks by typing in your terminal: `ant <Task Name>`	

Table 2.1: Safe Storage ANT Tasks

2.3 Virtual Sensors

2.4 Introduction

Virtual Sensor is the main abstraction used by GSN to represent a well structured data stream. By well structured we mean, the structure of the stream is known in advanced and it is not going to be changed while GSN is running.

2.5 Graphical Representation

2.6 Notes

2.6.1 What if the structure of the virtual sensor changes?

We are not changing the structure of the database automatically (adding or droping fields). If your virtual sensors structure changes and if you are using *permannet storages* such as MySQL or File-based HSQLDB, you have to change the structure of the table manually.

---

[1]In the current implementation, a Safe Storage *Wrapper* is identified by the following information: `<virtual-sensor-name>/<source-name>/<safe-storage-wrapper-short-name>`

2.6.2 Where is the DTD for the virtual sensor?

We are using JiBX Java-XML binding project. The structure of the virtual sensor descriptor file is defined in `conf/VirtualSensorDescription.xml`.

Have a look to the Chapter B.1.1.

2.6.3 Which databases we support ?

At the moment we are supporting HSqlDB and MySQL. Checkout the mailing list for the latest issues regarding the other databases and their support status.

2.6.4 Which projects are using GSN ?

There are over 10 EU/Swiss funding research projects using GSN as their core technology.

2.7 GSN.XML file

2.8 SafeStorage

2.8.1 Why we need it?

2.8.2 How to use it

2.8.3 How to write a wrapper for the safe storage

2.9 Wrappers

2.10 Their mandatory parameters

2.11 How it works

2.12 Introduction

2.13 GeoRSS

2.14 SQL Syntax

2.15 Introduction

2.16 Introduction

2.17 Introduction

2.18 Introduction

2.19 Introduction

2.20 Introduction

2.21 Introduction

2.22 Network communication

Looking inside the GSN infrastructure, there are at least half a dozen difference network communication channels are used. In this section I would like to dive in to the details of the

some major communication protocols designed and implemented in GSN.

2.22.1 Reusing Data Streams

One of the main ideas behind the virtual sensors is resuability. The resuability comes in two forms. First being able to recreate the same processing logic on different data streams. Second being able to reuse streaming data produced by other parties over the internet and possibly create a new data stream but instrumenting the original streams. In this section, I present the both high level and low level details associated with the second aspect of the reusability.

The virtual sensor descriptor file is the first place which specifies the intention of reusing streaming data from another virtual sensor. The source virtual sensor can be located anywhere as long as it is accessible through the network, this ofcourse includes the local machine and any other machine on the Internet.

In GSN, our vision is having an internet scale streaming world in which people can publish streaming data which can be produced directly using some sort of a measurement device which can range from a physical wireless sensor to stock ticks from a financial market.

2.23 GSN Notifications

2.23.1 Introduction

In GSN, virtual sensors can be configured to notify users of certain events, e.g. to send an Email notification to an user informing them that a particular event has occurred. To implement notifications in GSN is very straight forward. The basic principle is that once the virtual sensor query is answered as specified in the virtual sensor description file, e.g.

```
<query>SELECT temperature FROM s1 WHERE temperature >= 100</query>
```

a notification can be triggered by the java processing class

```
<class—name>gsn.vsensor.EmailVirtualSensor</class—name>
```

see examples in next section. Thus, any type of notifications, e.g. Email, SMS, SIP, Fax, MMS can be implemented easily in a virtual sensor processing class.

The technical details of implementing notifications are left to the designer. Below are three examples of some of the notification services already implemented in GSN.

Chapter 3

GSN in Nutshell

Abstract

With the price of wireless sensor technologies diminishing rapidly we can expect large numbers of autonomous sensor networks being deployed in the near future. These sensor networks will typically not remain isolated but the need of interconnecting them on the network level to enable integrated data processing will arise, thus realizing the vision of a global "Sensor Internet." This requires a flexible middleware layer which abstracts from the underlying, heterogeneous sensor network technologies and supports fast and simple deployment and addition of new platforms, facilitates efficient distributed query processing and combination of sensor data, provides support for sensor mobility, and enables the dynamic adaption of the system configuration during runtime with minimal (zero-programming) effort. This paper describes the Global Sensor Networks (GSN) middleware which addresses these goals. We present GSN's conceptual model, abstractions, and architecture, and demonstrate the efficiency of the implementation through experiments with typical high-load application profiles. The GSN implementation is available from http://gsn.sourceforge.net/.

Related Publications:

Different parts of the work presented in this chapter is published in the form of articles in international conferences and workshops. Parts of this chapter is also published in the form of internal technical reports.

- *Infrastructure for data processing in large-scale interconnected sensor networks*, Karl Aberer , Manfred Hauswirth , Ali Salehi. Mobile Data Management (MDM), Germany, 2007.

- *GSN, Quick and Simple Sensor Network Deployment*, Ali Salehi, Karl Aberer. European conference on Wireless Sensor Networks (EWSN), Netherlands, 2007.

- *Zero-programming Sensor Network Deployment*, Karl Aberer , Manfred Hauswirth , Ali Salehi. Next Generation Service Platforms for Future Mobile Systems (SPMS), Japan, 2007.

- *A middleware for fast and flexible sensor network deployment*, Karl Aberer , Manfred Hauswirth , Ali Salehi. Very Large Data Bases (VLDB) Seoul, Korea, 2006.

- *Middleware support for the "Internet of Things"*, Karl Aberer , Manfred Hauswirth , Ali Salehi. 5. GI/ITG KuVS Fachgesprch "Drahtlose Sensornetze", Universitt Stuttgart, 2006.

- *The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks*, Karl Aberer , Manfred Hauswirth , Ali Salehi. Technical Report, LSIR-2006-006.

- *Global Sensor Networks*, Karl Aberer , Manfred Hauswirth , Ali Salehi. Technical Report, LSIR-2006-001.
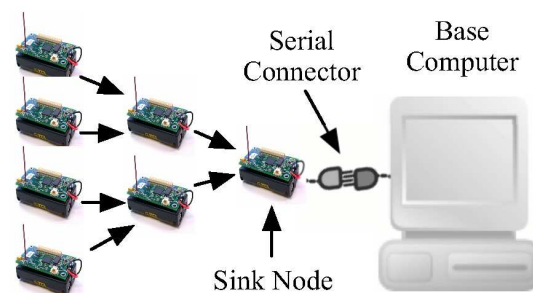
Figure 3.1: GSN model

3.1 Introduction

Until now, research in the sensor network domain has mainly focused on routing, data aggregation, and energy conservation inside a single sensor network while the integration of multiple sensor networks has only been studied to a limited extent. However, as the price of wireless sensors diminishes rapidly we can soon expect large numbers of autonomous sensor networks being deployed. These sensor networks will be managed by different organizations but the interconnection of their infrastructures along with data integration and distributed query processing will soon become an issue to fully exploit the potential of this "Sensor Internet." This requires platforms which enable the dynamic integration and management of sensor networks and the produced data streams.

The Global Sensor Networks (GSN) platform aims at providing a flexible middleware to accomplish these goals. GSN assumes the simple model shown in Figure 3.1: A sensor network internally may use arbitrary multi-hop, ad-hoc routing algorithms to deliver sensor readings to one or more sink node(s). A sink node is a node which is connected to a more powerful base computer which in turn runs the GSN middleware and may participate in a (large-scale) network of base computers, each running GSN and servicing one or more sensor networks.

We do not make any assumptions on the internals of a sensor network other than that the sink node is connected to the base computer via a software wrapper conforming to the GSN API. On top of this physical access layer GSN provides so-called *virtual sensors* which abstract from implementation details of access to sensor data and define the data stream processing to be performed. Local and remote virtual sensors, their data streams and the associated query processing can be combined in arbitrary ways and thus enable the user to build a data-oriented "Sensor Internet" consisting of sensor networks connected via GSN.

In the following we start with a detailed description of the virtual sensor abstraction in Section 3.2, discuss GSN's data stream processing and time model in Section 3.3, and present GSN's system architecture along with a discussion of essential implementation details in Section 3.4. We evaluate the performance of GSN in Section 3.7 and discuss related work in Section 3.8 before concluding.

3.2 Virtual sensors

The key abstraction in GSN is the *virtual sensor*. Virtual sensors abstract from implementation details of access to sensor data and correspond either to a data stream received directly from sensors or to a data stream derived from other virtual sensors. A virtual sensor can be any kind of data producer, for example, a real sensor, a wireless camera, a desktop computer, or any

combination of virtual sensors. A virtual sensor may have any number of input data streams and produces exactly one output data stream (with predefined format) based on the input data streams and arbitrary local processing. The specification of a virtual sensor provides all necessary information required for deploying and using it, including (1) metadata used for identification and discovery, (2) the details of the data streams which the virtual sensor consumes and produces (3) an SQL-based specification of the stream processing (filtering and integration) performed in a virtual sensor, (4) the processing class which preforms the more advanced and complex data processing (if needed) on the output stream before releasing it and (5) functional properties related to persistency, error handling, life-cycle, management, and physical deployment.

To support rapid deployment, the virtual sensors are provided in human readable declarative forms (XML). Figure 3.2 shows an example which defines a virtual sensor that reads two temperature sensors and in case both of them have the same reading above a certain threshold in the last minute, the virtual sensor returns the latest picture from the webcam in the same room together with the measured temperature.

A virtual sensor has a unique name (the `name` attribute in line 1) and can be equipped with a set of key-value pairs representing the logical addressing of the virtual sensor (lines 12–17), i.e., associated with metadata. The addressing information can be registered and discovered in GSN and other virtual sensors can use either the unique name or logical addressing based on the metadata to refer to a virtual sensor. We have defined certain addressing keys which are specifically used by the GSN's web interface. In GSN if a given virtual sensor has the addressing values for the both `latitude` (line 15) and `longitude` (line 16) keys, the default GSN web interface uses these geographical locations to show the sensor on the global map.

The example specification above defines a virtual sensor with three input streams which are identified by their metadata[1], i.e., by logical addressing. For example, the first temperature sensor is addressed by specifying two requirements on its metadata, namely that it is of type temperature sensor and at a certain physical certain location. By using multiple input streams Figure 3.2 also demonstrates GSN's ability to access multiple stream producers simultaneously. For the moment, we assume that the input streams (two temperature sensors and a webcam) have already been defined in other virtual sensor definitions (how this is done, will be described below).

In GSN data streams are temporal sequences of timestamped tuples (also known as `Stream Elements`). This is in line with the model used in most stream processing systems. The structure of the output data stream a virtual sensor produces is encoded in XML as shown in lines 6 – 9 (the `output-structure` part). The structure of the input streams is learned from the respective specifications of their virtual sensor definitions.

In GSN data stream processing is separated into three stages:

- processing applied to sources (lines 26, 33, and 40).

- processing for combining data from the different input streams and producing the temporary output stream (lines 43-46).

- producing the final output stream by passing the temporary output stream from a processing class (a processing logic represented in some programming languages). This part is presented by lines 3 – 10. Note that as the final output is produced by the processing class, the actual output structure of the virtual sensor should strictly conform the output format of the processing class [2].

---

[1]Note that the support for distributed directory/registry service had been removed from GSN's source code thus as of November 30, 2008, we only support physical addressing for identifying the data sources.

[2]As of November 30, 2008, the order and the types should be exactly match.

```
1  <virtual−sensor name="room−monitor" priority="10"
2          protected="false" >
3    <processing−class>
4       <class−name>gsn.vsensor.BridgeVirtualSensor</class−name>
5       <init−params/>
6       <output−structure>
7          <field name="image" type="binary:jpeg" />
8          <field name="temp"  type="int" />
9       </output−structure>
10   </processing−class>
11   <life−cycle pool−size="10" />
12   <addressing>
13       <predicate key="geographical">BC143</predicate>
14       <predicate key="usage">room monitoring</predicate>
15       <predicate key="latitude">46.5214</predicate>
16       <predicate key="longitude">6.5676</predicate>
17   </addressing>
18   <storage history−size="10h" />
19   <streams>
20       <stream name="cam">
21          <source name="cam"  storage−size="1" >
22             <address wrapper="remote">
23                <predicate key="geographical">BC143</predicate>
24                <predicate key="type">Camera</predicate>
25             </address>
26             <query>select * from WRAPPER</query>
27          </source>
28          <source name="temperature1" storage−size="1m" >
29             <address wrapper="remote">
30                <predicate key="type">temperature</predicate>
31                <predicate key="geographical">BC143−N</predicate>
32             </address>
33             <query>select AVG(temp1) as T1 from WRAPPER</query>
34          </source>
35          <source name="temperature2"  storage−size="1m" >
36             <address wrapper="remote">
37                <predicate key="type">temperature</predicate>
38                <predicate key="geographical">BC143−S</predicate>
39             </address>
40             <query>select AVG(temp2) as T2 from WRAPPER</query>
41          </source>
42          <query>
43             select cam.picture as image, temperature.T1 as temp
44             from   cam, temperature1
45             where  temperature1.T1 > 30 AND
46                    temperature1.T1 = temperature2.T2
47          </query>
48       </stream>
49   </streams>
50  </virtual−sensor>
```

Figure 3.2: A virtual sensor definition

To specify the processing of the sources we use SQL queries which refer to the actual data source by the reserved keyword `WRAPPER` (the data sources are logically represented as relational tables all of which called `wrapper`). The attribute `wrapper="remote"` indicates that the data stream is obtained through the network from another virtual sensor which can be located in any other GSN instance accessible through the network.

In the case of a directly connected local sensor, the `wrapper` attribute would reference the required wrapper[3]. For example, `wrapper="tinyos"` would denote a TinyOS-based sensor whose data stream is accessed via GSN's TinyOS wrapper [4]. GSN already includes wrappers for all major TinyOS platforms (Mica2, Mica2Dot, etc.), for wired and wireless (HTTP-based) cameras (e.g., AXIS 206W), several RFID readers (Texas Instruments, Alien Technology), Bluetooth devices, Shockfish, WiseNodes, epuck robots, etc. The implementation effort for wrappers is rather low, for example, the RFID reader wrapper has 50 lines of code (LOC), the TinyOS wrapper has 120 LOC, and the generic serial wrapper has 180 LOC.

In the given example the output stream joins the data received from two temperature sensors and returns a camera image if certain conditions on the temperature are satisfied (lines 43–46). To enable the SQL statement in lines 43–46 to produce the output stream, it needs to be able to reference the required sources which is accomplished by the `name` attribute (lines 21, 28, and 35) that defines a symbolic name for each stream source.

The definition of the structure of the output stream directly relates to the data stream processing that is performed by the virtual sensor's processing class and needs to be consistent with it. GSN provides multiple processing classes each of which are designed to perform different tasks (e.g., charts, network plots, filtering, ...). In our example we are using `gsn.vsensor.BridgeVirtualSensor` as our processing class. The `gsn.vsensor.BridgeVirtualSensor` class is special in the sense that unlike most of the other GSN's processing classes, this class does not perform any further processing on its input stream thus it does not alter the data nor the structure of its input.

Since the structure of the virtual sensor output is not altered through using the `gsn.vsensor.BridgeVirtualSensor` processing class hence the final structure of the virtual sensor's output is determined through the SQL statement at line 43, we need to make sure that, the data fields in the `select` clause matches the definition of the output structure in lines 6–9 (the order is important). It is recommended to use `gsn.vsensor.BridgeVirtualSensor` as long as the processing preformed in the virtual sensor through the SQL queries are sufficient enough and no further processing is required before publishing the sensor data to the outside.

In the design of GSN specifications we decided to separate the temporal aspects from the relational data processing using SQL. The temporal processing is controlled by various attributes provided in the input and output stream specifications, e.g., the attribute `storage-size` (lines 21, 28, and 35) defines the window size used for producing the input stream's data elements. Due to its specific importance the temporal processing will be discussed in detail in Section 3.3.

In addition to the specification of the data-related properties a virtual sensor also includes high-level specifications of functional properties: The `priority` attribute (line 1) controls the processing priority of a virtual sensor, the `<life-cycle>` element (line 11) enables the control and management of resources provided to a virtual sensor such as the maximum number of threads/queues available for processing, the `<storage>` element (line 18) allows the user to control how output stream data is persistently stored.

---

[3]As of November 30, 2008, all the wrappers has to written in Java language. The actual code for accessing the sensor can be written in any language as long as there is a possibility of communicating the data to the hardware through Java (e.g., interfacing Java to existing C code or the serial ports).

[4]In GSN, we have multiple TinyOS wrappers each corresponding to different versions and packet formats. Those details are out of the score of this chapter.

For example, in Figure 3.2 the `priority` attribute in line 1 assigns a priority of 10 to this virtual sensor (1 is the lowest priority and 20 the highest, default is 10), the `<life-cycle>` element in line 11 specifies a maximum number of 10 threads, which means that if the pool size is reached, data will be dropped (if no pool size is specified, it will be controlled by GSN depending on the current load), the `<storage>` element in line 18 defines that the output stream's data elements of the last 10 hours (`history-size` attribute) are stored to enable off-line processing. The `storage-size` attribute in line 21 defines the window size of 1 stream element. That's the most recent image taken by the webcam irrespective of the time it was taken.

In GSN, we can specify the set of values either by time or count. In the count based representation one only presents the values through integers. For instance `slide='2'` or `history-size='100'`. The count based representation is consists of an integer directly postfixed (without any space characters) with one of the time measurement units. As of November 30, 2008, we have `d,h,m,s` time measurement units which are corresponding to days, hours, minutes and seconds. As a time based example, we might have `storage-size='1m'`.

The `storage-size` attributes in lines 28 and 35 define a window of one minute for the amount of sensor readings subsequent queries will be run on, i.e., the `AVG` operations in lines 33 and 40 are executed on the sensor readings received in the last minute which of course depends on the rate at which the underlying temperature virtual sensor produces its readings. Note that when the `storage-size` is anything other than *1*, the virtual sensor author should be aware of the possibility of duplicated stream elements (discussed in more detail in section 3.3).

The query producing the output stream (lines 43–46) also demonstrates another interesting capability of GSN as it also mediates among three different flavors of queries: The virtual sensor itself uses continuous queries on the temperature data, a "normal" database query on the camera data and produces a result only if certain conditions are satisfied, i.e., a notification analogous to pub/sub or active rules.

Virtual sensors are a powerful abstraction mechanism which enables the user to declaratively specify sensors and combinations of arbitrary complexity. Virtual sensors can be defined and deployed to a running GSN instance at any time without having to stop the system. Also dynamic unloading is supported but should be used carefully as unloading a virtual sensor may have undesired (cascading) effects.

3.3 Data stream processing and time model

Data stream processing has received substantial attention in the recent years in other application domains, such as network monitoring or telecommunications. As a result, a rich set of query languages and query processing approaches for data streams exist on which we can build. A central building block in data stream processing is the time model as it defines the temporal semantics of data and thus determines the design and implementation of a system. Currently, most stream processing systems use a global reference time as the basis for their temporal semantics because they were designed for centralized architectures in the first place. As GSN is targeted at enabling a distributed "Sensor Internet," imposing a specific temporal semantics seems inadequate and maintaining it might come at unacceptable cost. GSN provides the essential building blocks for dealing with time, but leaves temporal semantics largely to applications allowing them to express and satisfy their specific, largely varying requirements. In our opinion, this pragmatic approach is viable as it reflects the requirements and capabilities of sensor network processing.

In GSN a data stream is a set of timestamped tuples also known as Stream Elements. The

order of the data stream is derived from the ordering of the timestamps and GSN provides basic support for managing and manipulating the timestamps. The following essential services are provided:

1. a local clock at each GSN Server

2. implicit management of a timestamp attribute (reserved field called `TIMED`)[5][6]

3. automatic timestamping of tuples upon arrival at the GSN in case the tuples (stream elements) don't have any timestamp (no `TIMED` field available)

4. a windowing mechanism which allows the user to define count- or time-based windows on data streams.

5. a sliding mechanism which allows the user to define count- or time-based sliding behaviors on data streams.

In this way it is always possible to trace the temporal history of data stream elements throughout the processing history. Multiple time attributes can be associated with data streams (as long as only one of them called `TIMED`) and can be manipulated through SQL queries. Thus sensor networks can be used as observation tools for the physical world, in which network and processing delays are inherent properties of the observation process which cannot be made transparent by abstraction. Let us illustrate this by a simple example: Assume a bank is being robbed and images of the crime scene taken by the security cameras are transmitted to the police. For the insurance company the time at which the images are taken in the bank will be relevant when processing a claim, whereas for the police report the time the images arrived at the police station will be relevant to justify the time of intervention. Depending on the context the robbery is thus taking place at different times.

As tuples (sensor readings) are timestamped, queries can also deal explicitly with time. For example, the query in lines 43–46 of Figure 3.2 could be extended such that it explicitly specifies the maximum time interval between the readings of the two temperatures and the maximum age of the readings. This would additionally require changes in the source definitions as the sources then must provide this information (more detailed example below), and also the averaging of the temperature readings (lines 33 and 40) would have to be changed to be explicit in respect to the time dimension.

In order to concretely show the time management inside GSN, we would like to simulate above scenario through two different virtual sensors (only the input stream parts presented). Say there exist a virtual sensor called *camera-vs* hosted on a GSN server which listens to port `80` on a machine with IP address of `1.2.3.4`. The virtual sensor used by the police and the one used by the insurance are depicted in figures 3.3 and 3.4. The stream specified in figure 3.3 has a query in line 7 for retrieving both the picture and the time stamp from the remote virtual sensor therefore the remote timestamp is used by GSN for the internal calculations. Now consider the stream specified in figure 3.4 which has a small change compared to the one in figure 3.3, the latter is not selecting the timestamp field hence GSN automatically adds the local reception time to every tuple it receives from the remote source.

In order to further elaborate the time management issue, consider the stream source specified in figure 3.5. This example combines both the local time and remote time in order to

---

[5]All timestamps in GSN are represented in milliseconds using 64-bit integers.

[6]As the timestamp (e.g., the `TIMED` field) is always present, it is not required to specify the `TIMED` field in the `output-structure` section of the virtual sensors. In fact, specifying the `TIMED` field in the output structure causes error therefore GSN refuses to load the virtual sensor.

```
1  <stream name="cam">
2      <source name="cam"  storage-size="1" >
3          <address wrapper="remote">
4              <predicate key="host">1.2.3.4</predicate>
5              <predicate key="port">80</predicate>
6              <predicate key="name">camera-vs</predicate>
7          </address>
8          <query>select PICTURE, TIMED from WRAPPER</query>
9      </source>
10      <query>
11          select PICTURE, TIMED from cam
12      </query>
13  </stream>
```

Figure 3.3: A stream using the remote timestamp.

```
1  <stream name="cam">
2      <source name="cam"  storage-size="1" >
3          <address wrapper="remote">
4              <predicate key="host">1.2.3.4</predicate>
5              <predicate key="port">80</predicate>
6              <predicate key="name">camera-vs</predicate>
7          </address>
8          <query>select PICTURE from WRAPPER</query>
9      </source>
10      <query>
11          select PICTURE, TIMED from cam
12      </query>
13  </stream>
```

Figure 3.4: A stream using the local (arrival) timestamp.

measure the latency associated with each tuple and uses the latency as a condition as the selection criteria (e.g., only accepting the tuples which are not delayed by the network for more than 5 milliseconds).

In order to deal with the streaming data, the standard way is to specify a query with at least two extra properties associated with it, window size and sliding value. The window size is used to limit the actual data used for the processing (execution) to a certain range in time or number of values. The sliding value is introduced to specify the execution condition for the query. The execution of the query is triggered whenever the sliding condition is satisfied implying a possibly infinitely long periodic execution of the query, therefore in stream processing systems, continuous queries are executed whenever the sliding occurs.

For instance, one can express the interest of obtaining the average of a temperature sensor over the last 10 minutes, and doing so periodically every 2 minutes, by simply providing the window size of 10 minutes and sliding value of 2 minutes to the stream processing engine. As indicated before, each time the sliding condition is satisfied (e.g., 2 minutes passed from the previous execution) the actual action, computing the average over the last 10 minutes, is performed. Note that in some research papers the execution of the action is also called *movement of the sliding window*.

The temporal processing in GSN is defined using the sliding and window values. Every data source in GSN can have at most one `slide`[7] and `storage-size`[8] attributes. Both values can be represented in the form of count-based or time-based values (described earlier in this

---

[7]Default value is 1, therefore this attribute can be omitted
[8]No default value defined

```
1  <stream name="cam">
2      <source name="cam"  storage-size="1" >
3          <address wrapper="remote">
4              <predicate key="host">1.2.3.4</predicate>
5              <predicate key="port">80</predicate>
6              <predicate key="name">camera-vs</predicate>
7          </address>
8          <query>select PICTURE, TIMED as REMOTE_TIMED from WRAPPER</query>
9      </source>
10     <query>
11         select PICTURE, REMOTE_TIMED AS TIMED from cam where
12         (cam.TIMED - cam.REMOTE_TIMED) < 5
13     </query>
14 </stream>
```

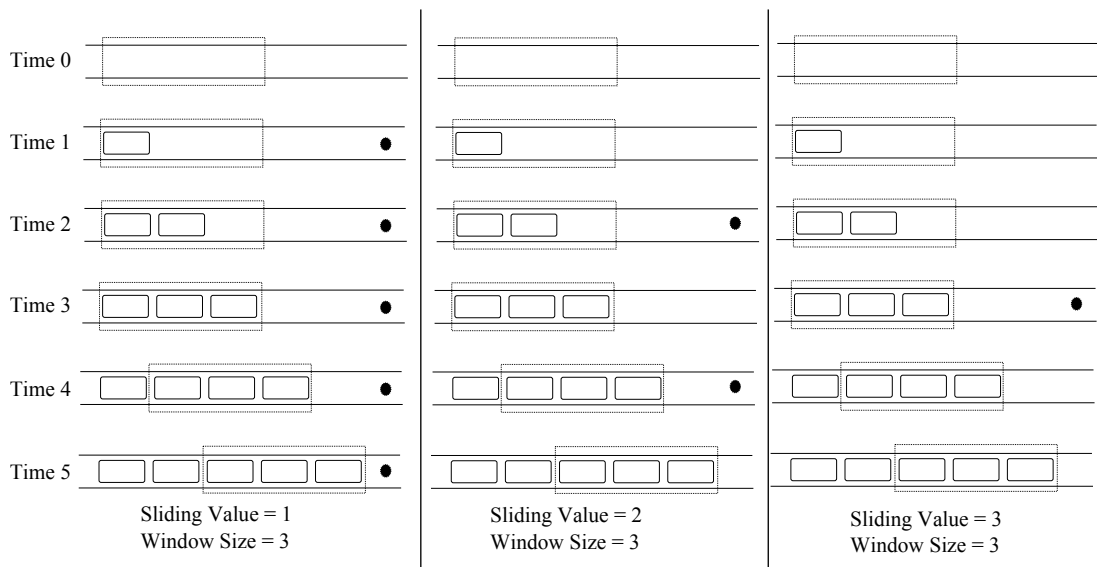Figure 3.5: A stream using both local and remote timestamps.

Figure 3.6: Illustration of the different sample sliding and window values.

section). Figure 3.6 visually represents the query execution inside GSN with different sliding and window values. We used a black dot in the figure to represent the triggering of execution. For instance, if both the window size and the sliding values are 3, and say we have received 5 stream elements in total, our continuous query have been executed only once (at the *Time 3*) during its life time. One can extend above paradigm to create virtual sensors to support the integration of continuous and historical data. For example, if the user wants to be notified when the temperature is 10 degrees above the average temperature in the last 24 hours, he/she can simply define two sources, getting data from the same wrapper but with different window sizes, i.e., 1 (count) and 24h (time), and then simply write a query specifying the original condition with these sources.

The production of a new output stream element of a virtual sensor is always triggered by the arrival of a data stream element from one of its input streams, thus processing is event-driven. As described before, a stream can have multiple sources. Once the window of one of the sources of a stream slides, the following processing steps are performed:

1. Based on the timestamps for each stream the stream elements are selected according to the definition of the time window and the resulting sets of relations are unnested into flat relations.

2. The queries defined on the source are evaluated and stored into temporary relations.

3. The stream query for producing the input of the processing class is executed based on the temporary relations.

4. The resulted stream elements are forwarded to the processing class.

5. The output of the processing class is stored and simultaneously forwarded (notification) to all consumers of the virtual sensor.

Figure 3.7 shows the logical data flow inside a GSN node.

Additionally, GSN provides a number of attributes in the virtual sensor file to control data rates. The values used for controlling these rates are usually float numbers between 0 to 1. The data rate controlling is useful whenever one wants to drop the stream elements with some random probability for load shedding. For instance, if one has a temperature source that keeps producing data with very high rate, one might want to sample the produced values thus making the processing lighter. For instance if one sets the sampling-rate to 0.75, any received stream element from the wrapper is going to be included in the window (the window and sliding values are explained above) with a probability of 75 out of 100. Thus, on average 25 random stream elements will be dropped out of the last 100 elements. In most of the cases one typically sets the rate control attributes to "1" to make sure nothing is dropped.

The rate control can be applied in the following three different levels (please refer to the virtual sensor quick reference for the syntactical information about different portions of the virtual sensor file):

- At the source level by providing `sampling-rate` attribute (float value in [0 .. 1]).

- At the stream level by providing `rate` attribute (integer value above zero).

- At the virtual sensor output level by providing `output-specification` → `rate` attribute (integer value above zero).
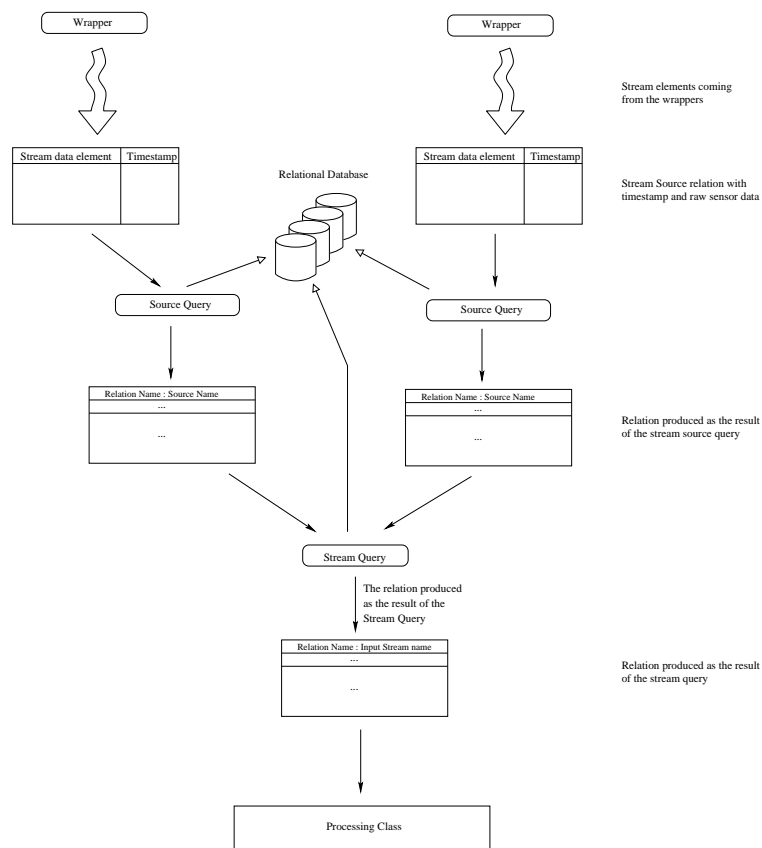
Figure 3.7: Conceptual data flow in a GSN node

As noted above, if the rate control is a positive integer, it defines the minimum allowed time difference between to successive stream elements. For instance, if one is interested in receiving an average of a given sensor once an hour but the sensor underneath can produce arbitrary number of stream elements (e.g., due to uncontrollable packet losses in the internal network), he can express this behavior by setting the rate attribute of the virtual sensor output (`output-specification → rate`) to "3600000" (one hour is 3,600,000 milliseconds).

To specify the data stream processing a suitable language is needed. A number of proposals exist already, so we compare the language approach of GSN to the major proposals from the literature. In the Aurora project [**?**] (http://www.cs.brown.edu/research/aurora/) users can compose stream relationships and construct queries in a graphical representation which is then used as input for the query planner. The Continuous Query Language (CQL) suggested by the STREAM project [**?**] (http://www-db.stanford.edu/stream/) extends standard SQL syntax with new constructs for temporal semantics and defines a mapping between streams and relations. Similarly, in Cougar [**?**] (http://www.cs.cornell.edu/database/cougar/) an extended version of SQL is used, modeling temporal characteristics in the language itself. The StreaQuel language suggested by the TelegraphCQ project [**?**] (http://telegraph.cs.berkeley.edu/) follows a different path and tries to isolate temporal semantics from the query language through external definitions in a C-like syntax. For example, for specifying a sliding window for a query a *for*-loop is used. The actual query is then formulated in an SQL-like syntax.

GSN's approach is related to TelegraphCQ's as it separates the time-related constructs from the actual query. Temporal specifications, e.g., the window size and rates, are specified in XML in the virtual sensor specification, while data processing is specified in SQL. Using this design, GSN can support SQL queries with the full range of operations allowed by the standard SQL syntax, i.e., joins, sub-queries, ordering, grouping, unions, intersections, etc. The advantage of using SQL is that it is well-known and SQL query optimization and planning techniques can be directly applied.

3.4   System architecture

GSN uses a container-based architecture for hosting virtual sensors. Similar to application servers, GSN provides an environment in which sensor networks can easily and flexibly be specified and deployed by hiding most of the system complexity in the GSN Server. Using the declarative specifications, virtual sensors can be deployed and reconfigured in GSN Servers at runtime. Communication and processing among different GSN Servers is performed in a peer-to-peer style through standard Internet and Web Services protocols. By viewing GSN Servers as cooperating peers in a decentralized system, we tried avoid some of the intrinsic scalability problems of many other systems which rely on a centralized or hierarchical architecture. Targeting a "Sensor Internet" as the long-term goal we also need to take into account that such a system will consist of "Autonomous Sensor Systems" with a large degree of freedom and only limited possibilities of control, similarly as in the Internet.

Figure 3.8 shows the layered architecture of a GSN Server.

Each GSN server hosts a number of virtual sensors it is responsible for. The virtual sensor manager (VSM) is responsible for providing access to the virtual sensors, managing the delivery of sensor data, and providing the necessary administrative infrastructure. The VSM has two subcomponents: The life-cycle manager (LCM) provides and manages the resources provided to a virtual sensor and manages the interactions with a virtual sensor (sensor readings, etc.). The input stream manager (ISM) is responsible for managing the streams, allocating resources to them, and enabling resource sharing among them while its stream quality manager
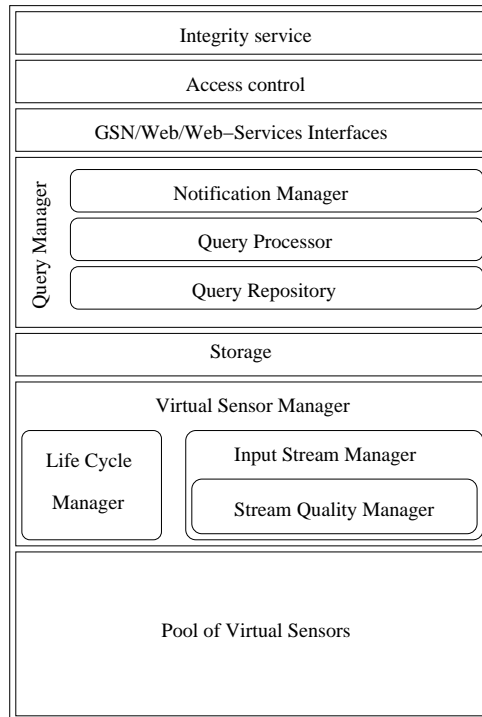
Figure 3.8: GSN Server architecture

subcomponent (SQM) handles sensor disconnections, missing values, unexpected delays, etc., thus ensuring the QoS of streams. All data from/to the VSM passes through the storage layer which is in charge of providing and managing persistent storage for data streams. Query processing in turn relies on all of the above layers and is done by the query manager (QM) which includes the query processor being in charge of SQL parsing, query planning, and execution of queries (using an adaptive query execution plan). The query repository manages all registered queries (subscriptions) and defines and maintains the set of currently active queries for the query processor. The notification manager deals with the delivery of events and query results to registered, local or remote virtual sensors. The notification manager has an extensible architecture which allows the user to largely customize its functionality, for example, having results mailed or being notified via SMS.

The top three layers of the architecture deal with access to the GSN server. The interface layer provides access functions for other GSN servers and via the Web (through a browser or via web services). These functionalities are protected and shielded by the access control layer providing access only to entitled parties and the data integrity layer which provides data integrity and confidentiality through electronic signatures and encryption. Data access and data integrity can be defined at different levels, for example, for the whole GSN server or at a virtual sensor level.

In connection with RFID tags this "plug-and-play" feature of GSN provides new and interesting types of mobility which we will investigate in future work. For example, an RFID tag may store queries which are executed as soon as the tag is detected by a reader, thus transforming RFID tags from simple means for identification and description into a GSN server for physically mobile queries which opens up new and interesting possibilities for mobile information systems.

3.5 Implementation

The GSN implementation consists of the GSN-CORE, implemented in Java, and the platform-specific GSN-WRAPPERS, implemented in Java, C, and Ruby, depending on the available toolkits for accessing specific types of sensors or sensor networks. The implementation currently has approximately 80,000 lines of code and is available from SourceForge (http://gsn.sourceforge.net/). GSN is implemented to be highly modular in order to be deployable on various hardware platforms from workstations to small programmable PDAs, i.e., depending on the specific platforms only a subset of modules may be used. GSN also includes visualization systems for plotting data and visualizing the network structure. In the following sections we are going to discuss some of the key aspects of the GSN implementation

3.5.1 Adding new sensor platforms

For deploying a virtual sensor the user only has to specify an XML document as described in Section 3.2, if GSN already includes software support for the concerned hardware/software. Adding a new type of sensor or sensor network can be done by supplying the name of the wrapper (specified in `/conf/wrappers.properties`) conforming to the GSN API. At the moment GSN provides the following wrappers:

HTTP generic wrapper is used to pull data from devices via HTTP GET or POST requests, for example, the AXIS206W wireless camera.

TinyOS wrapper enables interaction with TinyOS compatible motes (version 1.x and 2.x). This wrapper uses the serial forwarder which is the standard access tool for TinyOS provided in the TinyOS package.

USB camera wrapper is used for dealing with cameras connected via USB to the local machine. As USB cameras are very cheap, they are quite popular as sensing devices. The wrapper supports cameras with OV518 and OV511 chips (see http://alpha.dyndns. org/ov511/).

TI-RFID wrapper enables access to Texas Instruments Series 6000 S6700 multi-protocol RFID readers.

Generic UDP wrapper can be used for any device using the UDP protocol to send data.

Generic serial wrapper supports sensing devices which send data through the serial port.

Additionally, we provide template implementations for standard cases and frequently used platforms. If wrapper implementations are shared publicly this also facilitates building a reusable code base for virtually any sensor platform. The effort to implement wrappers is quite low.

New wrappers can be added to GSN without having to rebuild or modify the GSN server (plug-and-play). Upon startup GSN locates the wrapper mappings through reading the `/conf/wrapper.properties` file and loads each wrapper whenever needed by the system.

3.5.2 Dynamic resource management

The highly dynamic processing environment we target with GSN requires adaptive dynamic resource management to allow the system to quickly react to changing processing needs and environmental conditions. Dynamic resource management accomplishes three main tasks:
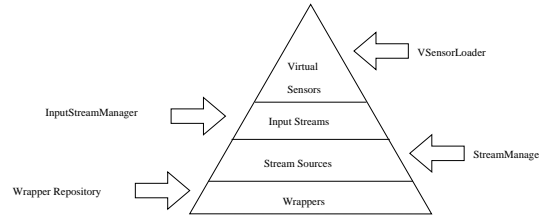
Figure 3.9: Hierarchical resource sharing in GSN

Resource sharing: As the user can modify/remove/add virtual sensors on-the-fly during runtime, the system needs to keep track of all resources used by the individual virtual sensors and enforce resource sharing among sensors (wrappers) where possible.

Failure management: If GSN detects a faulty virtual sensor or wrapper, e.g., by runtime exceptions, GSN undeploys it and releases the associated resources.

Explicit resource control: The user can specify explicit memory and processing requirements and restrictions. While restrictions are always enforced, requirements are handled depending of the globally available resources of the GSN instance. GSN tries to share the available resources in a fair way taking into account the explicitly specified resource requirements, if provided.

Dynamic resource management is performed at several levels in GSN as shown in Figure 3.9. Separating the resource sharing into several layers logically decouples the requirements and allows us to achieve a higher level of reuse of resources. In the following we will discuss the different levels.

Wrapper sharing. Wrappers communicate directly with the sensors which involves expensive I/O operations via a serial connection or wireless/wired network communication. To minimize the costs incurred by these operations GSN shares wrappers among virtual sensors accessing the same physical/virtual sensors. To do so each GSN node maintains a repository of active wrappers. If a new virtual sensor is deployed, the node first checks with the wrapper repository whether an identical wrapper already exists, i.e., wrapper name and initialization parameters (and their corresponding values) of the `<wrapper>` element in the virtual sensor definitions are identical. If a match is found, the new virtual sensor is registered to the existing wrapper as a consumer. If not, a new wrapper instance is created and registered with the wrapper repository. In the case of remote sensor accesses this strategy is applied at both the sending and receiving sides to maximize the sharing, i.e., multiple virtual sensors on one GSN node share a wrapper for the same remote sensor and on the node hosting the sensor the wrapper is shared among all nodes accessing it.

Data sharing. The raw input data produced by the wrappers is processed and filtered by the source queries to generate the actual input data for the input streams of a virtual sensor. For this purpose a source defines what part of the raw input data is used by the associated source query to produce the source's output data, i.e., by defining the available storage, sampling rates, and window sizes a view on the raw data is defined on which the source query is executed. In terms of the implementation each wrapper is assigned a storage holding the raw data and source queries are then defined as *SQL views* on this data store.

This has a number of advantages: (1) It minimizes the storage consumption as raw data is only stored once. Especially if the sensor data is large, e.g., image data, this is relevant. (2) If the sensor data comes from a power-constrained or slow device, power is conserved and processing is sped up. (3) Different processing strategies can be applied to the same data

without having to replicate it, for example, image enhancement algorithms and object detection can use the same raw image data.

In the same way as a wrapper can be shared by multiple sources, a source can also be shared among multiple streams at a higher level, and streams in turn are shared by multiple virtual sensors. In essence each of the layers in Figure 3.9 can be viewed as a resource pool where each of the individual resources in the pool can be shared among multiple resources at the next higher level. Conversely, each higher level resource can also use any number of lower level resources.

3.5.3 Query planning and execution

In GSN each virtual sensor corresponds to a database table and each sensor reading corresponds to a new tuple in the related table. As we use a standard SQL database as our low-level query processing engine, the question is how to represent the streaming logic in a form understandable for a standard database engine (as already described, GSN separates the stream processing directives from the query). We address this problem by using a query translator which gets an SQL query and the stream processing directives as provided in the virtual sensor definition as inputs and translates this into a query executable in a standard database. The query translator relies on special support functions which emulate stream-oriented constructs in a database. These support functions are dependent on the database used and are provided by GSN (currently we provide adapters for HSQLDB and MySQL). Translated queries are cached for subsequent use.

Upon deployment of a virtual sensor VS, all queries Q_i contained in its specification are extracted. Each query $Q_i(VS_1, \ldots, VS_n)$ accesses one or more relations $VS_1, \ldots, VS_n$ which correspond to virtual sensors. Then the query translator translates each $Q_i(VS_1, \ldots, VS_n)$ into an executable query $Q_i^t(VS_1, \ldots, VS_n)$ as described above and each $Q_i^t(VS_1, \ldots, VS_n)$ is declared as a view in the database with a unique identifier Id_i. This means whenever a new tuple, i.e., sensor reading, is added to the database, the concerned views will automatically be updated by the database. Additionally, a tuple (VS_j, Id_i, VS) for each $VS_j \in VS_1, \ldots, VS_n$ is added to a special view registration table. This procedure is done once when a virtual sensor is deployed.

With this setup it is now simple to execute queries over the data streams produced by virtual sensors: As soon a new sensor reading for a virtual sensor VS_d becomes available, it is entered into the according database relation. Then the database server queries the registration table using VS_d as the key and gets all identifiers Id_r registered for new data of VS_d. Then simply all views V_r affected by the new data item can be retrieved using the Id_r and all V_r can be queried using a `SELECT * FROM V_r` statement and the resulting data can be returned to the virtual sensor containing V_r (third column in the registration table). Since views are automatically updated by the database querying them is efficient. However, with many registered views (thousands or more) scalability may suffer. Thus GSN does not produce an individual query for each view but merges all queries into a large select statement, and the result will then be joined with the view registration table on the view identifier. Thus the result will hold tuples that identify the virtual sensor to notify of the new data. The reasons for applying this strategy are that (1) database connections are expensive, (2) with increasing number of clients and virtual sensor definitions, the probability of overlaps in the result sets increases which automatically will be exploited by the database's query processor, and (3) query execution in the database is expensive, so one large query is much less costly than many (possibly thousands) small ones.

Immediate notification of new sensor data is currently implemented in GSN and is an eager

strategy. As an alternative also a lazy strategy could be used where the query execution would only take place when the GSN instance requests it from the database, for example, periodically at regular intervals. In practice the former can be implemented using views or triggers and the latter can be implemented using inner selects or stored procedures.

3.6 GSN to GSN communication Protocol

In this section we would like to present the the low level details of GSN to GSN communication protocol. In order to enable data sharing and distributed collaborative data stream processing, we have introduced two special type of wrappers in GSN. First, the `local` wrapper, which enables data stream sharing among virtual sensors on the same machine. Second, the `remote` wrapper, which enables data stream sharing among multiple distributed virtual sensors each of which located on different machine accessible through the network.

In GSN whenever a virtual sensor wants to use another virtual sensor located on a different GSN server, the communication between two GSN server is triggered (during the loading process of the local virtual sensor). Once GSN notices that a remote virtual sensor is required by a local virtual sensor, GSN temporary suspends the local virtual sensor's loading process to confirm the existence of the remote virtual sensor. Therefore, GSN to GSN communication is initiated whenever a virtual sensor in a node A wants to use the data stream provided by another virtual sensor in a node B ($A \neq B$).

Using this kind of architecture, GSN mediates all the outgoing and incoming connections therefore the local virtual sensor does not interact directly with the remote virtual sensor (and vise versa). The packets exchanged between two GSN servers during GSN to GSN communication is depicted in figure 3.10 (all communications are implemented using XML-RPC calls). In the following we provide a brief description of each packet:

structure-request/structure is used by the local GSN server to discover the output structure of the remote virtual sensor. The response to this packet, confirms the existence and availability of the remote virtual sensor and contains the details of the output-structure of the remote virtual sensor.

register/confirm is used by the local GSN server to send the query and the contact address of the stream consumer. The query will be added to the notification list associated with the prospective virtual sensor at the stream producer, therefore whenever the remote virtual sensor produces a stream element, the query will be evaluated and the output of the evaluation(in case it is not empty) is delivered to the stream consumer. The remote virtual sensor uses the addressing information (received in the registration packet) to contact the stream consumer in order to deliver the stream elements. As there might be multiple virtual sensors at the stream consumer side be interested in one virtual sensor hosted at the stream producer, any register request has a UUID associated with it which is used by the stream producer whenever it wants to deliver stream elements to the stream consumer.

data represents the stream of tuples which are going to be delivered to the stream consumer. At the stream consumer, GSN server receives the data and based on the UUID of the tuples, GSN server disseminates the tuples to the appropriate local virtual sensors.

In order to make the GSN to GSN communication more concrete, we provide more system level details below. For using a remote virtual sensor, the first step is locating the *contact point* of the GSN server which hosts the prospective virtual sensor. By default, the contact point
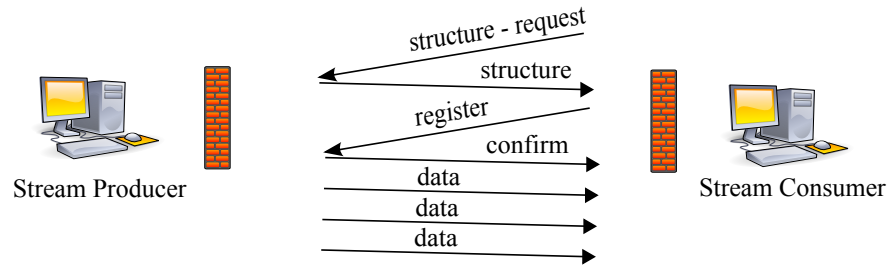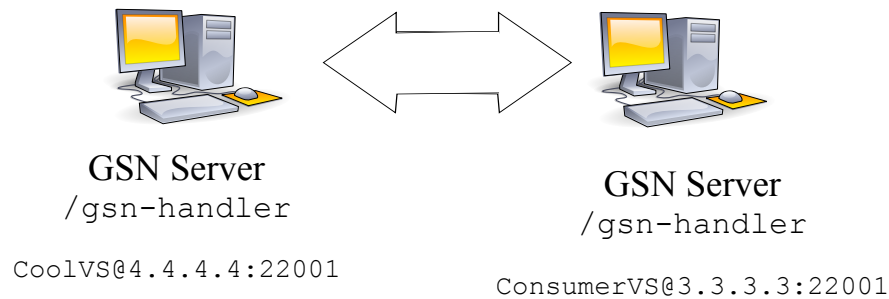
Figure 3.10: Experimental setup

Figure 3.11: Simple GSN to GSN communication

is `http://ip-address:gsn-port/gsn-handler`[9][10]. If the contact point is correctly identified, the response to a plain HTTP POST request returns a XML output.[11][12]

Correct identification of the contact point is crucial in success of using the remote virtual sensor. Once the contact points identified successfully, one can define a stream which consume data from the other data source. Note that consuming data from a remote virtual sensor doesn't require any kind of modifications at the remote host and in fact due to GSN's decoupled architecture, the remote virtual sensor is not even aware of its data consumers. In figure 3.11, the virtual sensor `ConsumerVS` running at the GSN server with the IP address of `3.3.3.3` under the port `22001` is interested in getting data from the `CoolVS` running at the GSN server with the IP address of `4.4.4.4` under the port `22001`. To enable this communication one has to use a source configuration similar to the one in figure 3.12.

In some deployments, GSN servers are hosted behind a NAT an apache web server[13] which can cause port and/or IP change. This can be true for both the GSN data stream consumer

---

[9]The port is specified in the `conf/gsn.xml` file.

[10]The exact mapping is specified in `webapp/WEB-INF/web.xml` file.

[11]The actual output represents an error as the request is not properly formatted.

[12]For sending plain HTTP POST requests to `http://ip-address:gsn-port/gsn-handler`, you may want to use http://code.google.com/p/rest-client/.

[13]The instruction for using GSN behind a apache web server is provided in appendix **??**.

```xml
1  <address wrapper="remote">
2    <predicate key="name">CoolVS</predicate>
3    <predicate key="host">4.4.4.4</predicate>
4    <predicate key="port">22001</predicate>
5  </address>
```

Figure 3.12: Source configuration for simple GSN to GSN communication.

Figure 3.13: Simple GSN to GSN communication

```
1  <address wrapper="remote">
2    <predicate key="name">CoolVS</predicate>
3    <predicate
        key="local−contact−point">http://1.1.1.1:82/gsn−2</predicate>
4    <predicate
        key="remote−contact−point">http://2.2.2.2:80/gsn−server</predicate>
5  </address>
```
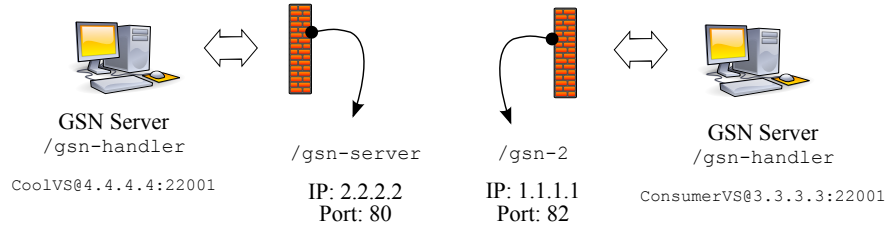
Figure 3.14: Source configuration for NATed GSN to GSN communication.

and GSN data stream producer. In these cases, one can use the more advanced form of the remote wrapper. Figure 3.13 presents a sample setup in which both of the GSN data stream consumer and data producer are behind firewall. The firewall at the consumer side has mapped `3.3.3.3:22001` into `1.1.1.1:82` and at the stream producer side firewall has mapped `4.4.4.4:22001` into `2.2.2.2:80`. To enable this kind of communication one has to use a source configuration similar to the one in figure 3.14.

3.6.1 `local` wrapper

The `local` wrapper is special version of `remote` wrapper (host = "127.0.0.1") which is optimized for communication among two different virtual sensors inside the same GSN server. By having the `local` wrapper optimized, we imply that most of the overhead associated with TCP/IP networking calls are eliminated by using internal GSN calls instead. The `local` wrapper is recommend whenever the end to end delay between two virtual sensors is important. In GSN, we have implemented the notification system so that the GSN server always gives priority to the local virtual sensors when it wans to disseminate the stream elements thus the local virtual sensors usually get notified earlier.

3.7 Evaluation[14]

GSN aims at providing a zero-programming and efficient infrastructure for large-scale inter-connected sensor networks. To justify this claim we experimentally evaluate the throughput of the local sensor data processing and the performance and scalability of query processing as the key influencing factors. As virtual sensors are addressed explicitly and GSN nodes communicate directly in a point-to-point (peer-to-peer) style, we can reasonably extrapolate the experimental results presented in this section to larger network sizes. For our experiments, we used the setup shown in Figure 3.15.

---

[14]The evaluation results in this section correspond to GSN release 0.90.

The GSN network consisted of 5 standard Dell desktop PCs with Pentium 4, 3.2GHz Intel processors with 1MB cache, 1GB memory, 100Mbit Ethernet, running Debian 3.1 Linux with an unmodified kernel 2.4.27. For the storage layer use standard MySQL 5.18. The PCs were attached to the following sensor networks as shown in Figure 3.15.

- A sensor network consisting of 10 Mica2 motes, each mote being equipped with light and temperature sensors. The packet size was configured to 15 Bytes (data portion excluding the headers).

- A sensor network consisting of 8 Mica2 motes, each equipped with light, temperature, acceleration, and sound sensors. The packet size was configured to 100 Bytes (data portion excluding the headers). The maximum possible packet size for TinyOS 1.x packets of the current TinyOS implementation is 128 bytes (including headers).

- A sensor network consisting of 4 Tiny-Nodes (TinyOS compatible motes produced by Shockfish, http://www.shockfish.com/), each equipped with a light and two temperature sensors with TinyOS standard packet size of 29 Bytes.

- 15 Wireless network cameras (AXIS 206W) which can capture 640x480 JPEG pictures with a rate of 30 frames per second. 5 cameras use the highest available compression (16kB average image size), 5 use medium compression (32kB average image size), and 5 use no compression (75kB average image size). The cameras are connected to a Linksys WRT54G wireless access point via 802.11b and the access point is connected via 100Mbit Ethernet to a GSN node.

- A Texas Instruments Series 6000 S6700 multi-protocol RFID reader with three different kind of tags, which can keep up to 8KB of data. 128 Bytes capacity.

The motes in each sensor network form a sensor network and routing among the motes is done with the surge multi-hop ad-hoc routing algorithm provided by TinyOS.

3.7.1 Internal processing time

In the first experiment we wanted to determine the internal processing time a GSN node requires for processing sensor readings, i.e., the time interval when the wrapper gets the sensor data until the data can be provided to clients by the associated virtual sensor. This delay depends on the size of the sensor data and the rate at which the data is produced, but is independent of the number of clients wanting to receive the sensor data. Thus it is a lower bound and characterizes the efficiency of the implementation.

We configured the 22 motes and 15 cameras to produce data every 10, 25, 50, 100, 250, 500, and 1000 milliseconds. As the cameras have a maximum rate of 30 frames/second, i.e., a frame every 33 milliseconds, we added a proxy between the GSN node and the WRT54G access point which repeated the last available frame in order to reach a frame interval of 10 milliseconds. All GSN instances used the Sun Java Virtual Machine (1.5.0 update 6) with memory restricted to 64MB.

The experiment was conducted as follows: All motes and cameras were set to the same rate and produced data for 8 hours and we measured the processing delay. This was repeated 3 times for each rate and the measurements were averaged. Figure 3.16 shows the results of the experiment for the different data sizes produced by the motes and the cameras.

High data rates put some stress on the system but the absolute delays are still quite tolerable. The delays drop sharply if the interval is increased and then converge to a nearly constant
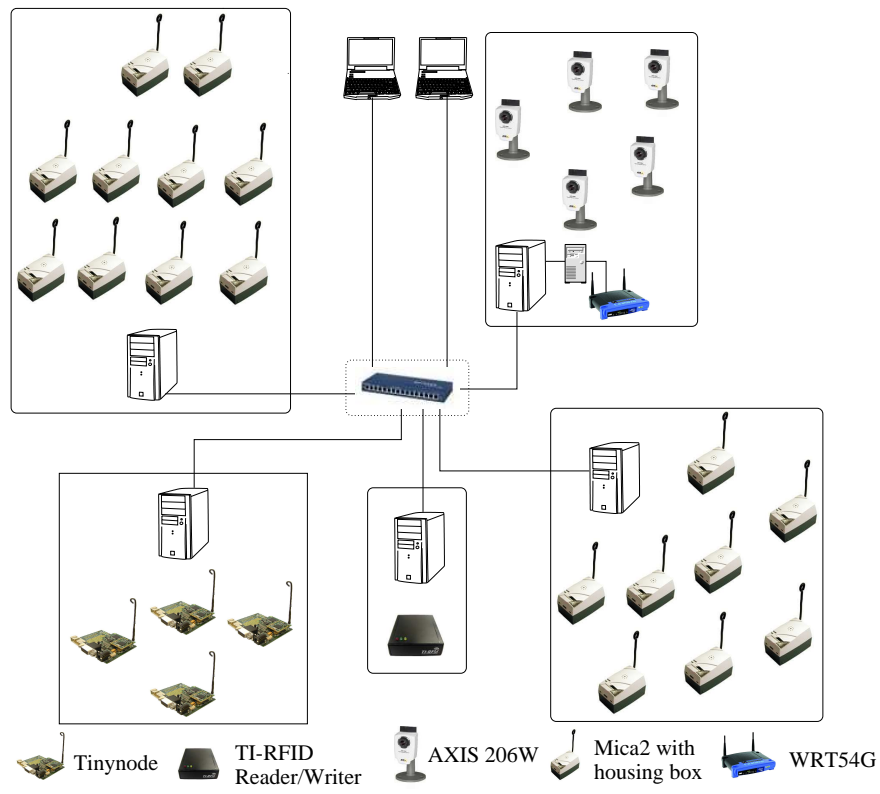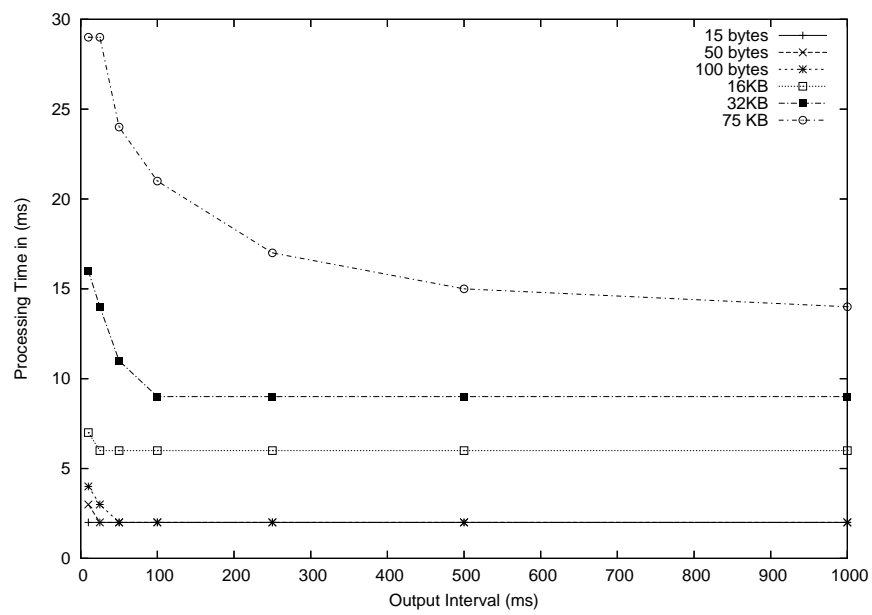
Figure 3.15: Experimental setup

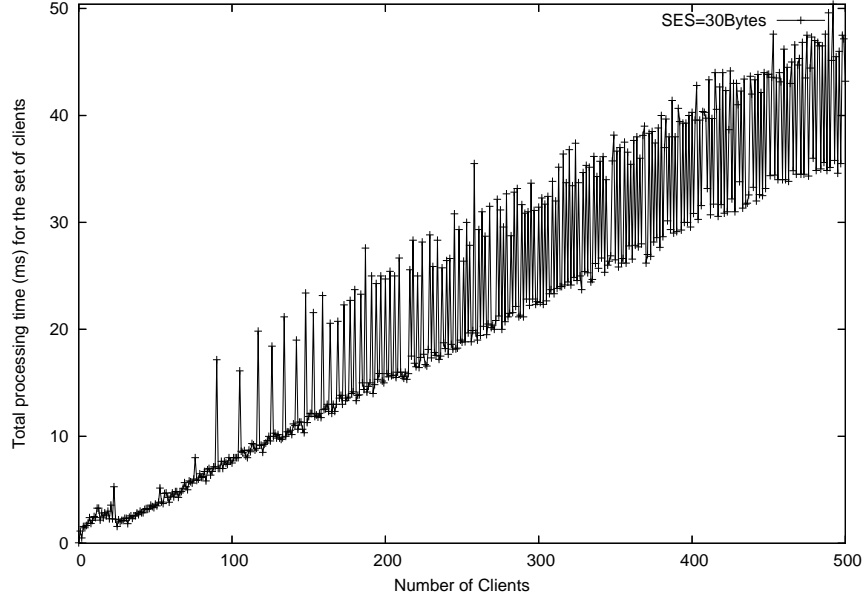Figure 3.16: GSN node under time-triggered load

Figure 3.17: Query processing latencies in a node

time at a rate of approximately 4 readings/second or less. This result shows that GSN can tolerate high rates and incurs low overhead for realistic rates as in practical sensor deployments lower rates are more probable due to energy constraints of the sensor devices while still being able to deal also with high rates.

3.7.2 Scalability in the number of queries and clients

In this experiment the goal was to measure GSN's scalability in the number of clients and queries. To do so, we used two 1.8 GHz Centrino laptops with 1GB memory as shown in Figure 3.15 which each ran 250 lightweight GSN instances. The lightweight GSN instance only included those components that we needed for the experiment. Each GSN-light instance used a random query generator to generate queries with varying table names, varying filtering condition complexity, and varying configuration parameters such as history size, sampling rate, etc. For the experiments we configured the query generator to produce random queries with 3 filtering predicates in the `where` clause on average, using random history sizes from 1 second up to 30 minutes and uniformly distributed random sampling rates (seconds) in the interval $[0.01, 1]$.

Then we configured the motes such that they produce a measurement each second but would deliver it with a probability $P < 1$, i.e., a reading would be dropped with probability $1 - P > 0$. Additionally, each mote could produce a burst of R readings at the highest possible speed depending on the hardware with probability $B > 0$, where R is a uniformly random integer from the interval $[1, 100]$. I.e., a burst would occur with a probability of $P * B$ and would produce randomly 1 up to 100 data items. In the experiments we used $P = 0.85$ and $B = 0.3$. On the desktops we used MySQL as the database with the recommended configuration for large memory systems. Figure 3.17 shows the results for a stream element size (SES) of 30 Bytes. Using SES=32KB gives the same latencies. Due to space limitations we do not include this figure.

The spikes in the graphs are bursts as described above. Basically this experiment measures the performance of the database server under various loads which heavily depends on the used database. As expected the database server's performance is directly related to the number of the clients as with the increasing number of clients more queries are sent to the database and also the cost of the query compiling increases. Nevertheless, the query processing time is reasonably low as the graphs show that the average time to process a query if 500 clients issue queries is less than 50ms, i.e., approximately 0.5ms per client. If required, a cluster could be used to the improve query processing times which is supported by most of the existing databases already.

In the next experiment shown in Figure 3.18 we look at the average processing time for a client excluding the query processing part. In this experiment we used $P = 0.85$, $B = 0.05$, and R is as above.

We can make three interesting observations from Figure 3.18:

1. GSN only allocates resources for virtual sensors that are being used. The left side of the graph shows the situation when the first clients arrive and use virtual sensors. The system has to instantiate the virtual sensor and activates the necessary resources for query processing, notification, connection caching, etc. Thus for the first clients to arrive average processing times are a bit higher. CPU usage is around 34% in this interval. After a short time (around 30 clients) the initialization phase is over and the average processing time decreases as the newly arriving clients can already use the services in place. CPU usage then drops to around 12%.

2. Again the spikes in the graph relate to bursts. Although the processing time increases considerably during the bursts, the system immediately restores its normal behavior with low processing times when the bursts are over, i.e., it is very responsive and quickly adopts to varying loads.

3. As the number of clients increases, the average processing time for each client decreases. This is due to the implemented data sharing functionalities. As the number of clients increases, also the probability of using common resources and data items grows.

3.8 Related work

So far only few architectures to support interconnected sensor networks exist. Sgroi et al. [?] suggest basic abstractions, a standard set of services, and an API to free application developers from the details of the underlying sensor networks. However, the focus is on systematic definition and classification of abstractions and services, while GSN takes a more general view and provides not only APIs but a complete query processing and management infrastructure with a declarative language interface.

Hourglass [?] provides an Internet-based infrastructure for connecting sensor networks to applications and offers topic-based discovery and data-processing services. Similar to GSN it tries to hide internals of sensors from the user but focuses on maintaining quality of service of data streams in the presence of disconnections while GSN is more targeted at flexible configurations, general abstractions, and distributed query support.

HiFi [?] provides efficient, hierarchical data stream query processing to acquire, filter, and aggregate data from multiple devices in a static environment while GSN takes a peer-to-peer perspective assuming a dynamic environment and allowing any node to be a data source, data sink, or data aggregator.
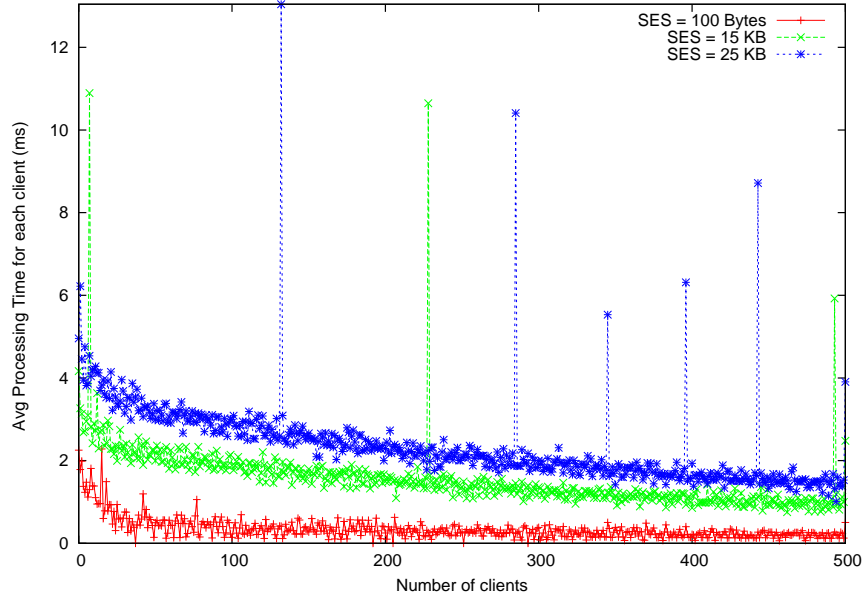
Figure 3.18: Processing time per client

IrisNet [?] proposes a two-tier architecture consisting of sensing agents (SA) which collect and pre-process sensor data and organizing agents (OA) which store sensor data in a hierarchical, distributed XML database. This database is modeled after the design of the Internet DNS and supports XPath queries. In contrast to that, GSN follows a symmetric peer-to-peer approach as already mentioned and supports relational queries using SQL.

Rooney et al. [?] propose so-called EdgeServers to integrate sensor networks into enterprise networks. EdgeServers filter and aggregate raw sensor data (using application specific code) to reduce the amount of data forwarded to application servers. The system uses publish/-subscribe style communication and also includes specialized protocols for the integration of sensor networks. While GSN provides a general-purpose infrastructure for sensor network deployment and distributed query processing, the EdgeServer system targets enterprise networks with application-based customization to reduce sensor data traffic in closed environments.

Besides these architectures, a large number of systems for query processing in sensor networks exist. Aurora [?] (Brandeis University, Braun University, MIT), STREAM [?] (Stanford), TelegraphCQ [?] (UC Berkeley), and Cougar [?] (Cornell) have already been discussed and related to GSN in Section 3.3.

In the Medusa distributed stream-processing system [?], Aurora is being used as the processing engine on each of the participating nodes. Medusa takes Aurora queries and distributes them across multiple nodes and particularly focuses on load management using economic principles and high availability issues. The Borealis stream processing engine [?] is based on the work in Medusa and Aurora and supports dynamic query modification, dynamic revision of query results, and flexible optimization. These systems focus on (distributed) query processing only, which is only one specific component of GSN, and focus on sensor heavy and server heavy application domains.

Additionally, several systems providing publish/subscribe-style query processing comparable to GSN exist, for example, [?].

3.9 Conclusions

The full potential of sensor technology will be unleashed through large-scale (up to global scale) data-oriented integration of sensor networks. To realize this vision of a "Sensor Internet" we suggest our Global Sensor Networks (GSN) middleware which enables fast and flexible deployment and interconnection of sensor networks. Through its virtual sensor abstraction which can abstract from arbitrary stream data sources and its powerful declarative specification and query tools, GSN provides simple and uniform access to the host of heterogeneous technologies. GSN offers zero-programming deployment and data-oriented integration of sensor networks and supports dynamic configuration and adaptation at runtime. Zero-programming deployment in conjunction with GSN's plug-and-play detection and deployment feature provides a basic functionality to enable sensor mobility. GSN is implemented in Java and C/C++ and is available from SourceForge at http://gsn.sourcefourge.net/. The experimental evaluation of GSN demonstrates that the implementation is highly efficient, offers very good performance and throughput even under high loads and scales gracefully in the number of nodes, queries, and query complexity.

List of Figures

List of Tables

Listings

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *Data-Stream Management: Processing High-Speed Data Streams*, chapter STREAM: The Stanford Data Stream Management System. Springer, 2006.

[3] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[4] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable Distributed Stream Processing. In *CIDR*, 2003.

[5] M. Franklin, S. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-in Systems: The HiFi Approach. In *CIDR*, 2005.

[6] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing*, 2(4), 2003.

[7] A. J. G. Gray and W. Nutt. A Data Stream Publish/Subscribe Architecture with Self-adapting Queries. In *International Conference on Cooperative Information Systems (CoopIS)*, 2005.

[8] Sean Rooney, Daniel Bauer, and Paolo Scotton. Techniques for Integrating Sensors into the Enterprise Network. *IEEE eTransactions on Network and Service Management*, 2(1), 2006.

[9] firstname1 secondname1 and firstname2 secondname2. Dummy Article. *Dummy Journal*, 20 August 2008.

[10] M. Sgroi, A. Wolisz, A. Sangiovanni-Vincentelli, and J. M. Rabaey. A service-based universal application interface for ad hoc wireless sensor and actuator networks. In *Ambient Intelligence*. Springer Verlag, 2005.

[11] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical Report TR-21-04, Harvard University, EECS, 2004. http://www.eecs.harvard.edu/~syrah/hourglass/papers/tr2104.pdf.

[12] Yong Yao and Johannes Gehrke. Query Processing in Sensor Networks. In *CIDR*, 2003.

[13] Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Cetintemel, Magdalena Balazinska, and Hari Balakrishnan. The Aurora and Medusa Projects. *Bulletin of the Technical Committe on Data Engineering, IEEE Computer Society*, 2003.

Appendix A

Developer's Guide

A.1 How to develop a wrapper

A.1.1 How to develop a Standard Wrapper

A.1.2 How to develop a Safe Storage Wrapper

This section describes step by step the development of wrappers that support the Safe Storage feature described in the Chapter 2.2. As an example to picture this development we use the Safe Storage Memory Monitor wrapper (`ss_mem_wrapper`), based on the standard one (`memory-usage`).

1. In the package `gsn.acquisition2.wrappers`, create the class that will execute in the Safe Storage process. By convention we name these classes like the following: `<Old wrapper name>2.java` (for our example `MemoryMontoringWrapper2.java`). This class must extend the abstract class `gsn.acquisition2.wrappers.AbstractWrapper2`.

2. Add a short name link for this new class in the file:
   `conf/safe_storage_wrappers.properties`.
   For our example we added:
   `mem2=gsn.acquisition2.wrappers.MemoryMontoringWrapper2`.

3. Create the class that will execute on the GSN process. By convention we name these classes like the following: `<Old wrapper name>Processor.java` (for our example `MemoryWrapperProcessor`). This class must extend the abstract class `gsn.acquisition2.wrappers.SafeStorageAbstractWrapper`.

4. Add a short name definition to this new class in the file:
   `conf/wrappers.properties`.
   For our example we added:
   `ss_mem_processor=gsn.acquisition2.wrappers.MemoryWrapperProcessor`.

5. Create a Virtual Sensor Description file for the sensor you will use to test your wrapper. For our example we created the file:
   `virtual-sensors/safe-storage/ss_mem_vs.xml`.
   Your *VSD* file must contain at least the following predicates which are mandatory for Safe Storage feature as shown in the Table B.7.

```xml
<address wrapper="GSN_SHORT_NAME (eg. ss_mem_processor)">
  <predicate key="ss-host">SAFE_STORAGE_HOST (default:
      localhost)</predicate>
  <predicate key="ss-port">SAFE_STORAGE_PORT (default: 25000)</predicate>
  <predicate key="wrapper-name">SAFE_STORAGE_SHORT_NAME (eg.
      mem2)</predicate>
</address>
```

6. Edit the wrapper class that runs on the Safe Storage process (for our example: `MemoryMontoringWrapper2`). Four methods have to be implemented

   - `boolean initialize ()`
     This method is called after the instanciation of the wrapper class. It is used to create the resources according to the parameters set in the configuration file. Your implementation must return false if some mandatory parameters were missing or if any error arise during this phase. In the other case, this method must return true.

   - `void finalize ()`
     This method should be used to free the wrapper ressources. It is called when the wrapper is unloaded.

   - `String getWrapperName ()`
     This method returns the SafeStorage wrapper's name.

   - `void run ()`
     Each wrapper runs in a separate thread. Use the run method to get data from your device and store them to the Safe Storage database with the method `postStreamElement (Serializable[])`. Notice that you can pass has many parameters has you want to this method but you must set the current time as the last parameter. This parameter will tag the creation of the packet in SafeStorage and is only necessary for running SafeStorage.
     For our example:
     `postStreamElement(heapMemoryUsage,nonHeapMemoryUsage,pendingFinalizationCount,System.c`

7. Edit the wrapper class that runs on the GSN part (in our case `MemoryWrapperProcessor`). Two methods have to be implemented

   - `DataField[] getOutputFormat ()`
     This method return an array of fields (names, and types) that are produced by the *Wrapper*.

   - `boolean messageToBeProcessed (DataMsg dataMessage)`
     This method is called upon reception of a new message from the SafeStorage. You can access the data (an array of Serializable objects) with the instance method `dataMessage.getData()`. The last element in this array is still the TimeStamp that you added before.
     Once you have parsed your data, you must use one of the `postStreamElement()` method to store data into GSN.

8. If you need to get some parameters from the VS XML configuration file, to initialize or finalize your wrapper, you can override the superclass methods:

   - `boolean initialize()`
   - `void finalize()`

A.2   How to cutomize the GSN Reports

Appendix B

Quick Reference Guide

B.1   Virtual Sensors (*VS*)

B.1.1   *VSD* DTD

All the *VS* are configured with an XML Virtual Sensor Description file (*VSD*). A graphical representation of the *VSD* Document Type Definition (DTD) is available on the Figure B.1. The description of all these tags are given in the Table B.1.
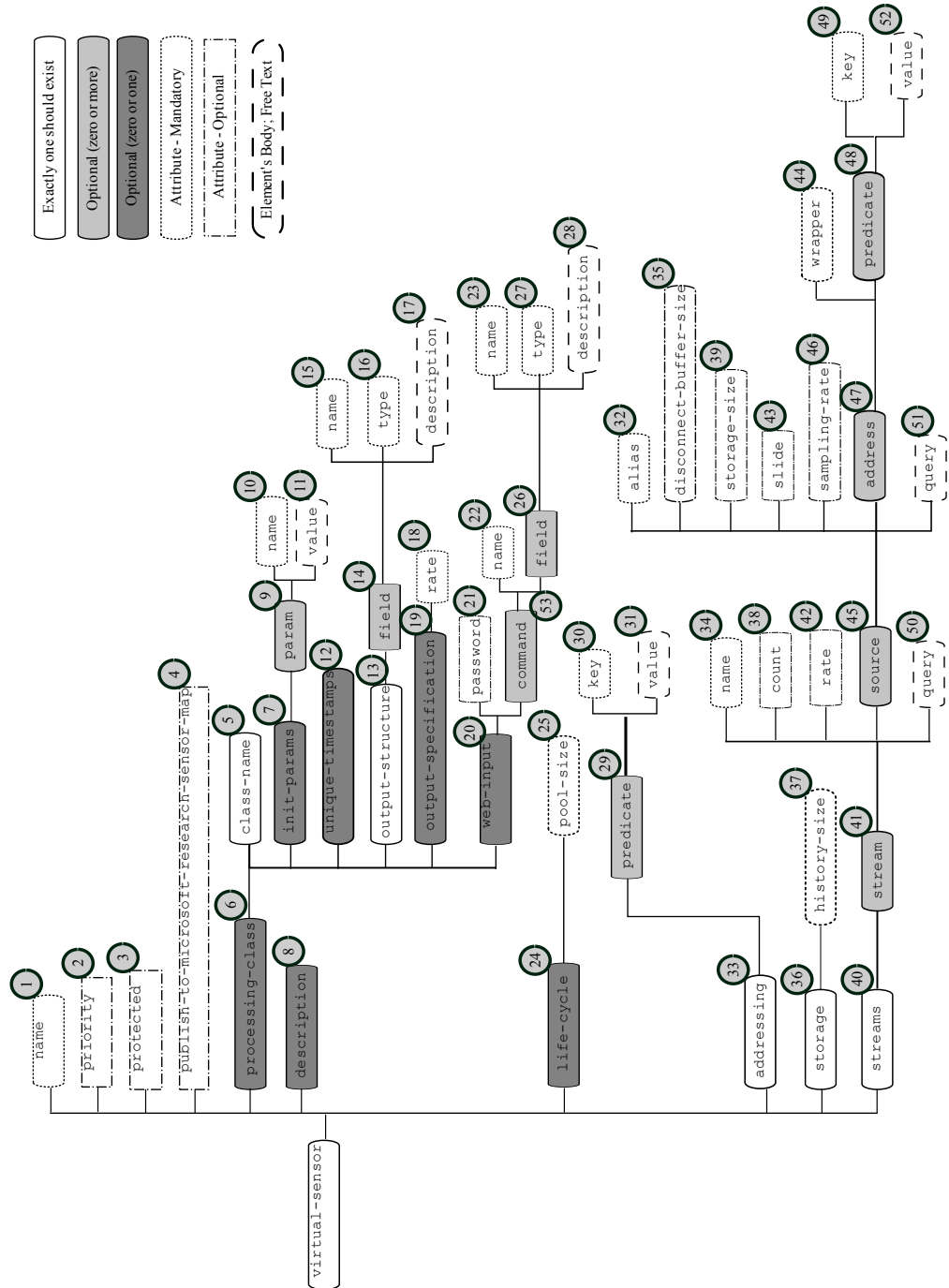
Figure B.1: *VSD* DTD Quick Reference Card

VSD DTD Quick Reference Card Description	
Tag Number	**Description**
1	TODO
2	TODO
3	TODO
4	TODO
5	TODO
6	TODO
7	TODO
8	TODO
9	TODO
10	TODO
11	TODO
12	TODO
13	TODO
14	TODO
15	TODO
16	TODO
17	TODO
18	TODO
19	TODO
20	TODO
21	TODO
22	TODO
23	TODO
24	TODO
25	TODO
26	TODO
27	TODO
28	TODO
29	TODO
30	TODO
31	TODO
32	TODO
33	TODO
34	TODO
35	TODO
36	TODO
37	TODO
38	TODO
39	TODO
40	TODO
41	TODO
42	TODO
43	TODO
44	TODO
45	TODO
46	TODO
47	TODO
48	TODO
49	TODO
50	TODO
51	TODO
52	TODO

Table B.1: *VSD* DTD Quick Reference Card Description

B.1.2   Email Notification *VS*

This virtual sensor implements an Email notification system for *GSN*. This *VS* is based on the Chapter B.2.1 *VSP* and the Chapter B.3.2 *Wrapper*. The Listing B.1 shows an example of *VSD* for this *VS*.

```xml
<virtual-sensor name="email" priority="10">
    <processing-class>
        <class-name>gsn.vsensor.EmailVirtualSensor</class-name>
        <init-params>
            <param name="RECEIVER">John Connor</param>
            <param name="receiver-email">john.connor@gmail.com</param>
            <param name="sender-email">admin@sensorinternet.com</param>
            <param name="mail-server">smtp.gmail.com</param>
            <param name="subject">Abnormal Temperature Detected</param>
            <param name="MESSAGE">Sensor 114 has a value of 100 C.</param>
        </init-params>
        <output-structure>
            <field name="temp" type="double" />
        </output-structure>
    </processing-class>
    <description>Send an Email Notification</description>
    <life-cycle pool-size="10" />
    <addressing />
    <storage history-size="10m" />
    <streams>
        <stream name="in1">
            <source alias="s1" sampling-rate="1" storage-size="1">
                <address wrapper="multiformat">
                    <predicate key="HOST">localhost</predicate>
                    <predicate key="PORT">22001</predicate>
                </address>
                <query>SELECT * FROM wrapper</query>
            </source>
            <query>SELECT temperature FROM s1 WHERE temperature >= 100</query>
        </stream>
    </streams>
</virtual-sensor>
```

Listing B.1: Sample of Email Notification VSD file

B.1.3 SMS Notification *VS*

This virtual sensor implements SMS (Short Message Service) notification. This virtual sensor is very similar to the email virtual sensor an email is sent to a Mobile phone operator or SMS gateway provider based on the user's mobile account and the email is converted and send as a SMS to the given phone number. This *VS* is based on the Chapter B.1.3 *VSP* and the Chapter B.3.2 *Wrapper*. The Listing B.2 shows an example of *VSD* for this *VS*.

```xml
<virtual-sensor name="sms" priority="10">
   <processing-class>
      <class-name>gsn.vsensor.SMSVirtualSensor</class-name>
      <init-params>
         <param name="phone-number">004413243545</param>
         <param name="password">3524</param>
         <param name="sms-server">vodafone.co.uk</param>
         <param name="message-format">Temperature: $TEMP$</param>
      </init-params>
      <output-structure>
         <field name="temp" type="double" />
      </output-structure>
   </processing-class>
   <description>Send a SMS Notification</description>
   <life-cycle pool-size="10" />
   <addressing />
   <storage history-size="10m" />
   <streams>
      <stream name="in1">
         <source alias="s1" sampling-rate="1" storage-size="1">
            <address wrapper="multiformat">
               <predicate key="HOST">localhost</predicate>
               <predicate key="PORT">22001</predicate>
            </address>
            <query>SELECT * FROM wrapper</query>
         </source>
         <query>SELECT temperature FROM s1 WHERE temperature >= 100</query>
      </stream>
   </streams>
</virtual-sensor>
```

Listing B.2: Sample of SMS Notification VSD file

Note that this virtual sensor will only work if you have an account with a mobile phone operator or an internet SMS gateway provider.

B.1.4   Voip Notification *VS*

This virtual sensor implements a phone call notification. The virtual sensor makes a phone call when a condition in the virtual sensor query is triggered. The virtual sensor is configured to work with Asterisk[1]. So, you will need to install[2] or have an existing asterisk server running. To configure the sensor with asterisk you will need to add the following line: *"#include extensions_gsn.conf"* to the *extensions_custom.conf* file in asterisk (/etc/asterisk/extensions_custom.conf). Once the virtual sensor is deployed, it automatically creates a dial plan in asterisk and registers extensions needed to make the phone calls, minimal configuration is required. This will include the any extensions created by the virtual sensor in the file *extensions_gsn.conf*. Depending on the number of times the virtual sensor is deployed, an extension with the virtual sensor name and a random extension number (internal to asterisk) will be created. The virtual sensor uses the Manager API to connect (login) to asterisk and execute remote commands (e.g. load dial plan, make phone call, call forward). To loging to the asterisk manager, you have to create an account and add your IP address in the *manager.conf* file in asterisk (/etc/asterisk/manager.conf):

```
[testuser]
secret = mypassword
permit=192.168.12.101/255.255.0.0
read = system,call,log,verbose,command,agent,user
write = system,call,log,verbose,command,agent,user
```

Listing B.3: Settings for Manager.conf file in Asterisk

Where, *testuser* is the username and *secret* is the password. In *permit* add your IP address (otherwise asterisk will block you). As part of asterisk, you will also need to install the *festival* text to speech system[3] and the *sox* utility[4]. These programs are used by the virtual sensor to convert the notification message (string) to speech (audio) to be played by asterisk. If using linux/ubuntu, do an *"apt-get install festival sox"*.

This *VS* is based on the Chapter B.1.4 *VSP* and the Chapter B.3.2 *Wrapper*. The Listing B.6 shows an example of *VSD* for this *VS*.

```xml
<virtual-sensor name="phone" priority="10">
   <processing-class>
   <class-name>gsn.vsensor.VoipVirtualSensor</class-name>
      <init-params>
         <param name="username">testuser</param>
         <param name="password">mypassword</param>
         <param name="host">asterisk-server.com</param>
         <param name="number">0041786831809</param>
         <param name="message">The temperature in the lab is too high.</param>
      </init-params>
      <output-structure>
         <field name="temperature" type="double" />
      </output-structure>
   </processing-class>
   <description>Makes a phone call when a query in the virtual sensor is
      satisfied.</description>
   <life-cycle pool-size="10" />
   <addressing>
      <predicate key=""></predicate>
   </addressing>
   <storage history-size="1h" />
  <streams>
```

---

[1]http://www.asterisk.org/
[2]http://www.trixbox.org/downloads
[3]http://www.cstr.ed.ac.uk/projects/festival/
[4]http://sox.sourceforge.net/

```
    <stream name="input1">
    <source alias="source1" sampling-rate="1" storage-size="1">
      <address wrapper="multiformat">
          <predicate key="HOST">localhost</predicate>
          <predicate key="PORT">22001</predicate>
      </address>
      <query>SELECT temperature, timed FROM wrapper WHERE packet_type=2</query>
    </source>
    <query>SELECT temperature, timed FROM source1</query>
    </stream>
  </streams>
</virtual-sensor>
```

Listing B.4: Sample of VoIP VSD file

Note that this virtual sensor will only work if you have an account with a mobile phone operator or an internet SMS gateway provider.

B.1.5   R *VS*

The software architecture of GSN allows the integration of third-party numerical packages such as Matlab, Mathematica, and more recently the R statistical package. In this section, we describe how to integrate and perform data analysis using R and GSN.

Installation and Setup

R is a programming language and a software package for statistical computing and analysis. R provides a range of techniques for statistical analysis such as liner and nonlinear modeling, classical statistical test, time-series analysis. The R software platform is composed of an engine that can interpret and execute R scripts or programs (in a similar way to GNU Octave or Matlab). The R engine can be invoked from the console or over the network using a TCP/IP server called Rserve. In this section, we describe how to configure the R and the Rserve server.

Installation

First, we have to install R. R is available in several platforms, depending on your platform you will have to follow specific instructions. We recommend to read the documentation from the official project website: http://www.r-project.org/.

Rserve TCP/IP Server

Rserve is a TCP/IP server which allows other programs to use invoke R from various languages without the need to initialize R or link against R library. Rserve can be downloaded from the following website: http://www.rforge.net/Rserve/. Rserve is written in Java and it has bindings for other languages. Rserve comes as a R package, therefore, to install and run Rserve, the library and the package have to be invoked within R as follows:

```
user@host\# R
R version 2.6.2 (2008−02−08)
Copyright (C) 2008 The R Foundation for Statistical Computing

> library(Rserve)
> Rserve()
```

Listing B.5: Running R and starting Rserve server.

This will load the Rserve library and start the Rserve TCP/IP server on the localhost. The default port is 6311. To read more about how to invoke R using Rserve, please refer to the Rserve documentation website: http://www.rforge.net/Rserve/.

This *VS* is based on the Chapter B.1.4 *VSP* and the Chapter B.3.2 *Wrapper*. The Listing **??** shows an example of *VSD* for this *VS*.

```xml
<virtual−sensor name="plot" priority="10">
   <processing−class>
   <class−name>gsn.vsensor.RVirtualSensor</class−name>
      <init−params>
      <param name="plot">2D</param>
         <param name="device">jpeg</param>
         <param name="operation">x<−rnorm(10);plot(x,$temperature$);</param>
      </init−params>
      <output−structure>
         <field name="plot" type="binary:image/jpeg"/>
      </output−structure>
```

```
  </processing-class>
  <description>Plots a graph using R.</description>
  <life-cycle pool-size="10"/>
  <addressing>
     <predicate key=""></predicate>
  </addressing>
  <storage history-size="1h"/>
 <streams>
   <stream name="input1">
   <source alias="source1" sampling-rate="1" storage-size="1">
     <address wrapper="multiformat">
        <predicate key="HOST">localhost</predicate>
        <predicate key="PORT">22001</predicate>
     </address>
     <query>SELECT temperature, timed FROM wrapper WHERE packet_type=2</query>
   </source>
   <query>SELECT temperature, timed FROM source1</query>
   </stream>
 </streams>
</virtual-sensor>
```

Listing B.6: Sample of R VSD file

B.1.6 GPS *VS*

This virtual sensor integrates Generic NMEA GPS Devices with *GSN*. This *VS* is based on the Chapter B.2.5 *VSP* and the Chapter B.3.4 *Wrapper*. The Listing B.7 shows an example of *VSD* for this *VS*.

```xml
<virtual-sensor name="gpsd" priority="10">
   <processing-class>
      <class-name>gsn.vsensor.BridgeVirtualSensor</class-name>
      <init-params/>
      <output-structure>
         <field name="Latitude" type="double"/>
         <field name="Longitude" type="double"/>
         <field name="Altitude" type="double"/>
         <field name="RateOfClimb" type="double"/>
         <field name="SpeedOverGround" type="double"/>
         <field name="GpsDeviceName" type="char(255)"/>
         <field name="GpsProtocol" type="char(255)"/>
         <field name="CountOfSatellites" type="int"/>
         <field name="VerticalDOP" type="double"/>
         <field name="HorizontalDOP" type="double"/>
         <field name="PositionalDOP" type="double"/>
         <field name="TimeDOP" type="double"/>
         <field name="GeometricDOP" type="double"/>
      </output-structure>
   </processing-class>
   <description>Send a SMS Notification</description>
   <life-cycle pool-size="10" />
   <addressing />
   <storage history-size="10m" />
   <streams>
      <stream name="input1">
         <source alias="source1" sampling-rate="1" storage-size="1">
            <address wrapper="gpsd">
               <predicate key="ListenerPort">2947</predicate>
               <predicate key="SamplingRate">2000</predicate>
               <predicate key="Timeout">22000</predicate>
            </address>
            <query>select * from wrapper</query>
         </source>
         <query>select * from source1</query>
      </stream>
   </streams>
</virtual-sensor>
```

Listing B.7: Sample of VSD using the Gpsd Wrapper

B.1.7 TinyOS *VS*

This virtual sensor integrates TinyOS[5] data sources with *GSN*. This *VS* is based on the Chapter B.2.5 *VSP* and the Chapter B.3.3 *Wrapper*. The Listing B.8 shows an example of *VSD* for this *VS*.

```xml
<virtual-sensor name="ss_migwrapper" priority="10" >
    <processing-class>
        <class-name>gsn.vsensor.BridgeVirtualSensor</class-name>
        <unique-timestamps>true</unique-timestamps>
         <output-structure>
          <field name="SYSVOLTAGE" type="INTEGER" />
          <field name="REALSYSVOLTAGE_V" type="DOUBLE" />
          <field name="SDIVOLTAGE" type="INTEGER" />
          <field name="REALSDIVOLTAGE_V" type="DOUBLE" />
          <field name="TEMPERATURE" type="INTEGER" />
          <field name="REALTEMPERATURE_C" type="DOUBLE" />
          <field name="MOISTURE" type="INTEGER" />
          <field name="APPROXMOISTURE_REL" type="DOUBLE" />
          <field name="REALMOISTURE_REL" type="DOUBLE" />
          <field name="TIMESTAMP" type="BIGINT" />
        </output-structure>
    </processing-class>
    <description> TODO Add a description</description>
    <life-cycle pool-size="10" />
    <addressing/>
    <storage />
    <streams>
    <stream name="data">
      <source alias="source" storage-size="1" sampling-rate="1">
        <address wrapper="ss_tinyos-mig">
                <predicate key="ss-host">localhost</predicate>
                <predicate key="ss-port">25000</predicate>
                <predicate key="continue-on-error">true</predicate>
                <predicate key="wrapper-name">mig2</predicate>
                <predicate key="wrapper-keep-processed-ss-entries">false</predicate>
                <predicate key="source">sf@permafrozer.ethz.ch:9001</predicate>
                <predicate
                    key="message-classname">ch.ethz.permafrozer.DozerDataMsg</predicate>
                <predicate key="message-length">23</predicate>
                <predicate key="getter-prefix">get</predicate>
          </address>
        <query> select * from wrapper </query>
      </source>
      <query> select * from source</query>
    </stream>
  </streams>
</virtual-sensor>
```

Listing B.8: Sample of VSD

---

[5]http://www.tinyos.net/

B.2 Virtual Sensor Processing classes (*VSP*)

B.2.1 gsn.vsensor.EmailVirtualSensor *VSP*

Parameters for gsn.vsensor.EmailVirtualSensor *VSP*				
Parameter Name	**Type**	**Mandatory**	**Default**	**Description**
RECEIVER	String	Yes	None	Name of the email recipient
receiver-email	String	Yes	None	Email address of the recipient
sender-email	String	Yes	None	Email address of the sender
mail-server	String	Yes	None	URL for the email (SMTP) server
subject	String	Yes	None	Subject of the email
message	String	Yes	None	Email message

Table B.2: Parameters for gsn.vsensor.EmailVirtualSensor *VSP*

B.2.2 gsn.vsensor.SMSVirtualSensor *VSP*

Parameters for gsn.vsensor.SMSVirtualSensor *VSP*				
Parameter Name	**Type**	**Mandatory**	**Default**	**Description**
phone-number	String	Yes	None	The mobile phone number to send the message
password	String	Yes	None	The password to login to the mobile operator or SMS gateway provider
sms-server	String	Yes	None	URL address of the mobile phone operator or SMS gatewat provider
message-format	String	Yes	None	The format of the SMS message in StringTemplate-syntax, e.g. Temperature: $TEMP$ where TEMP has some value given from the GSN StreamElement

Table B.3: Parameters for gsn.vsensor.SMSVirtualSensor *VSP*

B.2.3 gsn.vsensor.VoipVirtualSensor *VSP*

Parameters for gsn.vsensor.VoipVirtualSensor *VSP*				
Parameter Name	**Type**	**Mandatory**	**Default**	**Description**
username	String	Yes	None	Username to login to the asterisk server.
password	String	Yes	None	Password to login to the asterisk server.
host	String	Yes	None	DNS address of the asterisk server.
number	String	Yes	None	Telephone number to dial.
message	String	Yes	None	Notification message.

Table B.4: Parameters for gsn.vsensor.VoipVirtualSensor *VSP*

B.2.4 gsn.vsensor.RVirtualSensor *VSP*

B.2.5 gsn.vsensor.BridgeVirtualSensor *VSP*

Parameters for gsn.vsensor.RVirtualSensor *VSP*				
Parameter Name	**Type**	**Manda- tory**	**De- fault**	**Description**
phone- number	String	Yes	None	The mobile phone number to send the message
password	String	Yes	None	The password to login to the mobile operator or SMS gateway provider
sms-server	String	Yes	None	URL address of the mobile phone operator or SMS gatewat provider
message- format	String	Yes	None	The format of the SMS message in StringTemplate-syntax, e.g. Temperature: $TEMP$ where TEMP has some value given from the GSN StreamElement

Table B.5: Parameters for gsn.vsensor.RVirtualSensor *VSP*

Parameters for gsn.vsensor.BridgeVirtualSensor *VSP*				
Parameter Name	**Type**	**Mandatory**	**Default**	**Description**
TODO	TODO	TODO	TODO	TODO

Table B.6: Parameters for gsn.vsensor.BridgeVirtualSensor *VSP*

B.3 Wrappers

B.3.1 Safe Storage Wrappers Default parameters

The parameters shown on the Listing B.7 must be added to each *Wrapper* that support the Safe Storage feature. An example of *Wrapper* that use this feature is described in Chapter B.3.3.

Safe Storage Parameters				
Parameter Name	**Type**	**Manda- tory**	**De- fault**	**Description**
ss-host	String	Yes	None	The machine host name that runs the Safe Storage
ss-port	Inte- ger	Yes	None	The server port on which Safe Storage listen for con- nections
wrapper-name	String	Yes	None	The Safe Storage side wrap- per full classname (must extends `gsn.acquisition2.wrappers.AbstractWrapper2`) or the short name from the `conf/safe\_storage\ _wrappers.properties` file.
wrapper-keep- processed-ss-entries	Boolean	No	true	If this option is set to `true`, all the entries (processed or not) kept into the Safe Storage storage. If this option is set to `false`, the processed entries are removed from the Safe Storage storage once processed.
continue-on-error	Boolean	No	true	Not yet implemented

Table B.7: Safe Storage Parameters

B.3.2 multiformat *Wrapper*

B.3.3 ss_tinyos-mig *Wrapper*

The TinyOS *Wrapper* can receive data from both version 1.x and 2.x TinyOS based networks. This *Wrapper* can interract with any TinyOS compatible Base Station and any type of TinyOS Packet Source. For instance The TinyOS wrapper can interact with the serial forwarder (pro-

multiformat *Wrapper* Parameters				
Parameter Name	**Type**	**Mandatory**	**Default**	**Description**
TODO	TODO	TODO	TODO	TODO
Support Safe Storage		No		
GSN *Wrapper* Classname		gsn.wrappers.MultiFormatWrapper (multiformat)		

Table B.8: multiformat *Wrapper* Parameters

multiformat *Wrapper* Output Structure		
Name	**Type**	**Description**
TODO	TODO	TODO

Table B.9: multiformat *Wrapper* Output Structure

vided by TinyOS distribution) which inturn presents a sensor network. In order to use the TinyOS wrapper with a sensor network you need to first generate the java representation of the message structures used in the network (in TinyOS they are defined in the `.h` files). NesC language provides a tool called `mig` (message interface generator for nesC) for this purpose. The tool has a standard man page documentation in addition to being described in the lession 4 of the TinyOS 2 tutorial [6].

Both TinyOS 1.x and 2.x are using very same package structures and class names which generates conflict when you have both TinyOS1.x and 2.x jar files in your classpath (which is the case for *GSN*). To solve this issue, *GSN* provides a slightly modified version of the TinyOS 1.x java tools by renaming the `net.tinyos.xxx` package to `net.tinyos1x.xxx`. Thus the messages classes generated with MIG must be configured depending on the version of TinyOS your are using. The Table B.10 shows the class that your MIG generated messages must extend (directly or not directly).

TinyOS version	**Must extends**
1.x	`net.tinyos1x.message.Message`
2.x	`net.tinyos.message.Message`

Table B.10: Superclass for TinyOS messages classes

This *Wrapper* infers the output structure from the methods names contained into the MIG generated message class and the superclasses. To filter these methods, This *Wrapper* applies a prefix pattern matching on the methods names. The mapping between the TinyOS types and the types to use in your *VSD* are shown on the Table B.12.

B.3.4 GpsdWrapper *Wrapper*

GpsdWrapper is a GSN wrapper for communicating with gpsd and supports querying GPS devices compliant to

- NMEA 0183 protocol

- Rockwell binary protocol

- TSIP binary protocol

---

[6]Further informations can be found at
http://www.tinyos.net/tinyos-1.x/doc/nesc/mig.html
http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/lesson4.html
http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson6.html

ss_tinyos-mig *Wrapper* Parameters				
Parameter Name	**Type**	**Manda- tory**	**Default**	**Description**
source	String	Yes	None	The TinyOS data source eg. sf@serial.forwarder.url:9001
message- classname	String	Yes	None	The full package path to the message class generated by MIG eg. `ch.ethz.permafrozer.DozerDataMsg`
message- length	In- te- ger	No	`DEFAULT_MESSAGE_SIZE`	Override the default size of the messages received
getter-prefix	String	No	get_	The methods of the message class that contain this prefix will be added in the output structure. Keep the default value for the messages generated with MIG.
Support Safe Storage SS *Wrapper* Classname GSN *Wrapper* Classname	Yes (Parameters listed on Table B.7 must be added.) gsn.acquisition2.wrappers.MigMessageWrapper2 (mig2) gsn.acquisition2.wrappers.MigMessageWrapperProcessor (ss_tinyos-mig)			

Table B.11: ss_tinyos-mig *Wrapper* Parameters

ss_tinyos-mig *Wrapper* Output Structure			
nesC	**Java**	***VSD***	**Description**
nx_int8_t	byte	TINYINT	8-bit signed
nx_uint8_t	short	SMALLINT	8-bit unsigned
nx_int16_t	short	SMALLINT	16-bit signed
nx_uint16_t	int	INTEGER	16-bit unsigned
nx_int32_t	int	INTEGER	32-bit signed
nx_uint32_t	long	BIGINT	32-bit unsigned
NOT SUPPORTED	float	DOUBLE	32-bit floating point number
NOT SUPPORTED	double	DOUBLE	64-bit floating point number
Arrays of the listed types are also supported.			

Table B.12: ss_tinyos-mig *Wrapper* Output Structure

- SiRF protocol

- Garmin binary protocol

- Evermore binary protocol

Where are the source files?

GpsdWrapper *Wrapper* Parameters				
Parameter Name	**Type**	**Mandatory**	**Default**	**Description**
HostName	String	No	localhost	The hostname of GPS Daemon
ListenerPort	Integer	No	2947	The listener port of GPS Daemon
Timeout	Long	No	20000	The timeout for Telnet session to GPS Daemon in ms
SamplingRate	Long	No	2000	the rate of GPS sampling in ms
Support Safe Storage		No		
GSN *Wrapper* Classname		gsn.acquisition.wrappers.?? (short name)		

Table B.13: GpsdWrapper *Wrapper* Parameters

GpsdWrapper Output Structure		
Name	**Type**	**Description**
Latitude	type?	Current latitude in degrees
Longitude	type?	Current longitude in degrees
Altitude	type?	Current altitude in meters above sea level
RateOfClimb	type?	Current rate of climb in meters per second
SpeedOverGround	type?	Current speed over ground in meters per second
GpsDeviceName	type?	Active GPS device name on the GPS host
GpsProtocol	type?	GPS protocol in use
VerticalDOP	type?	Vertical dilution of precision
HorizontalDOP	type?	Horizontal dilution of precision
PositionalDOP	type?	Positional dilution of precision
TimeDOP	type?	Time dilution of precision
GeometricDOP	type?	Geometric dilution of precision

Table B.14: GpsdWrapper *Wrapper* Output Structure

Connecting a GPS device to gpsd

gpsd is a Linux daemon that monitors one or more GPS devices attached to a host computer through serial or USB ports. All data of the GPS devices is made available to be queried on TCP port 2947 of the host computer. With gpsd, multiple GPS client applications (such as navigational and wardriving software) can share access to GPS devices without contention or loss of data. Also, gpsd responds to queries with a format that is substantially easier to parse than the NMEA 0183 emitted by most GPS devices.

Using Bluetooth

We are given a Bluetooth GPS device that we want to connect to a Linux host. In this tutorial, it's a HOLUX GPSlim 236.

First, we need to start bluetooth services:

```
sudo /etc/init.d/bluetooth start
```

Now, let's scan for bluetooth devices:

```
hcitool scan
```

This should return a list of devices like

```
00:0B:0D:85:77:79 HOLUX GPSlim236
00:16:4E:D7:AE:5F Nokia N70
00:12:62:AF:C0:6E Nino
00:11:67:80:41:96 BT-GPS
```

As already mentioned, we are going to use the HOLUX GPSlim 236. We want to map the HOLUX GPSlim 236 to a emulated RS-232 serial port. To this end, we use the Bluetooth protocol RFCOMM. That's pretty simple and goes as follows. First we create a config file for the RFCOMM:

```
sudo nano /etc/bluetooth/rfcomm.conf
```

and add an entry for our HOLUX GPSlim 236 to this file

```
rfcomm0 {
bind yes;
device ''00:0B:0D:85:77:79'';
channel 1;
comment "Your comment here";
}
```

This way, we are mapping the HOLUX to a emulated RS-232 serial port

```
/dev/rfcomm0
```

by using the shell command

```
sudo rfcomm connect 0
```

We should get the following return

```
Connected /dev/rfcomm0 to 00:0B:0D:85:77:79 on channel 1
Press CTRL-C for hangup
```

Great, now we open a second terminal and connect the gpsd to our /dev/rfcomm0

```
sudo gpsd -b -N -D 4 /dev/rfcomm0
```

You can telnet into the gpsd to play around and check if it's working correctly

```
telnet localhost 2947
```

Cool, now we have a Bluetooth GPS device connected to a Linux host! A GSN server can use a GpsdWrapper to connect to this machine and read the GPS data.

Using SSH reverse tunneling

As you might already guess, the GpsdWrapper uses telnet to connect to gpsd.

If you are concerned about security, or can not telnet into the GPS host machine, e.g. a mobile phone, or just want to be fancy, let's do some SSH reverse tunneling! By the way, this also would allow access to the GPS host if it were behind a firewall.

In this scenario, we connect a HOLUX GPSlim 236 via Bluetooth to a Nokia N810 which runs gpsd by default. Usually, the N810 doesn't have a static IP address and looking up the IP address and manually typing it in is annoying so we set up SSH reverse tunneling.

On the N810, we simply execute

```
ssh -N -R 1234:localhost:2947 user@gsn-server.com
```

This forwards the port 1234 on gsn-server.com to the default gpsd port 2947 on the N810.

On the GSN host, we can telnet into gpsd and play around by

```
telnet localhost 1234
```

B.4 GSN ANT Tasks

GSN ANT Tasks	
Task Name	**Description**
start-all	Start both the Safe Storage and the GSN processes.
stop-all	Stop both the Safe Storage and the GSN processes.
start-acquisition	Start the Safe Storage process. The wrapper that were loaded during the last runs will be automatically resumed and will directly start acquiring data.
clean-acquisition	Delete all the Safe Storage permanent storage and flush the list of Wrappers to resume. Use this task with caution since it may delete some unprocessed data forever.
stop-acquisition	Stop the Safe Storage process.
gsn	Start the GSN process. You also have to start the Safe Storage process if you are using Safe Storage wrappers.
stop	Stop the GSN process. The Safe Storage processes if any will continue running and acquire the data.
restart	Stop and restart the GSN process.
compile-reports	Compile the Jasper Reports located in the `gsn-reports` directory. Must be called after modification of any Jasper report configuration file (.jrxml).
Use each of these tasks by typing in your terminal: `ant <Task Name>`	

Table B.15: GSN ANT Tasks

Appendix C

LaTeX Examples

Parameters for example Wrapper (that support safe storage)				
Parameter Name	**Type**	**Mandatory**	**Default**	**Description**
param1	String	Yes	None	param1 description
This Wrapper supports Safe Storage. Parameters listed on Table B.7 must be added.				

Table C.1: Parameters for example Wrapper (that support safe storage)

Parameters for example Wrapper (that doesn't support safe storage)				
Parameter Name	**Type**	**Mandatory**	**Default**	**Description**
param1	String	Yes	None	param1 description
param2	String	Yes	None	param2 description

Table C.2: Parameters for example Wrapper (that doesn't support safe storage)

```ruby
# A method
def mymethod
   @foo.each { |bar| bar.to_s }
   @@foobar = "a string"
end
```

Listing C.1: Ruby Code Example

```java
/**
 * Java Class
 */
public class Test {
   private int test;
   public Test () {}
   public Test (int level) {
      // A comment
      String b = "my test string";
      this.test = test;
   }
}
```

```xml
<!-- one comment -->
<tag attribute="val1">
   <tag2 p="5" />
</tag>
```

Listing C.2: XML Code Example

```html
<!-- one comment -->
<html>
    <head>
        <title>Test Page</title>
    </head>
    <body>
        <p style="display: none;">Test</p>
    </body>
</html>
```

citation [1]

http://www.google.ch www.google.ch

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *Data-Stream Management: Processing High-Speed Data Streams*, chapter STREAM: The Stanford Data Stream Management System. Springer, 2006.

[3] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[4] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable Distributed Stream Processing. In *CIDR*, 2003.

[5] M. Franklin, S. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-in Systems: The HiFi Approach. In *CIDR*, 2005.

[6] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing*, 2(4), 2003.

[7] A. J. G. Gray and W. Nutt. A Data Stream Publish/Subscribe Architecture with Self-adapting Queries. In *International Conference on Cooperative Information Systems (CoopIS)*, 2005.

[8] Sean Rooney, Daniel Bauer, and Paolo Scotton. Techniques for Integrating Sensors into the Enterprise Network. *IEEE eTransactions on Network and Service Management*, 2(1), 2006.

[9] firstname1 secondname1 and firstname2 secondname2. Dummy Article. *Dummy Journal*, 20 August 2008.

[10] M. Sgroi, A. Wolisz, A. Sangiovanni-Vincentelli, and J. M. Rabaey. A service-based universal application interface for ad hoc wireless sensor and actuator networks. In *Ambient Intelligence*. Springer Verlag, 2005.

[11] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical Report TR-21-04, Harvard University, EECS, 2004. http://www.eecs.harvard.edu/~syrah/hourglass/papers/tr2104.pdf.

[12] Yong Yao and Johannes Gehrke. Query Processing in Sensor Networks. In *CIDR*, 2003.

[13] Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Cetintemel, Magdalena Balazinska, and Hari Balakrishnan. The Aurora and Medusa Projects. *Bulletin of the Technical Committe on Data Engineering, IEEE Computer Society*, 2003.