The Book of GSN

June 24, 2008

Contents

List of Tables

Part I

User's Guide

Chapter 1

Introduction

1.1 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 functionnalities 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.

1.2 Installing GSN (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 (command 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:

<storage user="sa" password="" driver="org.hsqldb.jdbcDriver" url="jdbc:hsqldb:MEM:." />

This version includes not tools to run the GSN Gui.

1.3 Installing GSN (source)

You wish to run the most up to 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/.

1.3.1 To run from a command line

- Download the jakarta apache ant version 1.7.x or higher.
- add ANT_HOME/bin 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
- Here are the list of the ant tasks:

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 Gui
`jar`	Creates a jar file from the source.
`clean`	Removes the current build files and forces a rebuild. You may need to run this task once yo
`cleandb`	Removes the redundant tables which are create for holding GSN's internal states. You can r

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

1.3.2 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
- Download or import the GSN source code.
- Execute Eclipse
- Go to File -> New -> Project
- Choose "Java Project from Existing Ant Buildfile" -> Next
- Browse -> go to folder gsn/trunk/ -> choose file build.xml

- The project name will automatically appeard as gsn. In order to be able to use the debugging features of Eclipse with GSN, the following additional configuration is required\footnote{As GSN uses the Jibx framework for reading the xml documents, it is not straightforward how to configure your eclipse environment to run/debug your virtual sensors.}:
- Step 1: Setting the Ant Home for Eclipse:
  - Go to Window Menu and select Preferences, on the left side, click on the 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. Click on OK.
- Step 2: Setting an Ant Build System for your GSN project :
  - On the Project menu and click on properties tab.
  - Select Build on the left side and click on new and select Ant.
  - A new window pops up which has several tabs :
    * Main Tab:
      1. For build file, click on Browse Workspace and select the build.xml file
      2. For base directory, click on the Browse Workspace and click on the project name which contains the gsn source code.
      3. Let the Set an Input Handler selected.
    * Targets Tab:
      1. For After a Clean, click on Set Targets and select both Build and Bind.
      2. Click on Ok.

    You'll be returned back to the Builders page, make sure both Java Builder AND New Builder (or what ever you named it in the previous window) are selected. Click on OK.

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

Configuring and running GSN

2.1 The basics

GSN comes preconfigured with many virtual sensors activated.

It is possible to run GSN from a graphical user interface, or from the command line interface. The graphical interface is self explanatory ; you only need to run it and to click on the Start button. It also allows to easily analyze GSN log output.

The command to start GSN from the command line interface is:

```
ant gsn
```

This will generate a very verbose output. Don't worry if the text goes too fast: all the information is stored in a log file (logs/gsn.log) that you can view with your favorite text editor. You can configure the log level by editing the file `conf/log4j.properties`. There are several log levels, from DEBUG (maximum logging) to WARN (minimum logging, maximum performance). If you want to reduce the log verbosity, change the log4j.rootLogger to INFO or WARN. The conf/log4j.properties file specifies two destinations for the output log, one of them is called "console" which presents the standard output and the other one called file which points to logs/gsn.log file. The log4j.rootLogger=WARN,console,file sets the logging level of both detinations to WARN level.

If you want to debug your configuration, simply do the opposite. This parameter can also be set from the graphical user interface.

GSN does much more than generating log statements. You can access the web interface of your gsn server on `http://127.0.0.1:22001`.

You should see something like Figure 2.1.

2.2 Some example configuration scenarios

As can be seen on Figure 2.2, GSN is built around three subsystems: data acquisition, data processing and data output. Most of the time these three functionalities can run on the same computer.

However, more complex scenarios can be imagined, and GSN servers can collaborate so that each of them only deals with one of these three functions.

Figure 2.1: GSN Web Interface

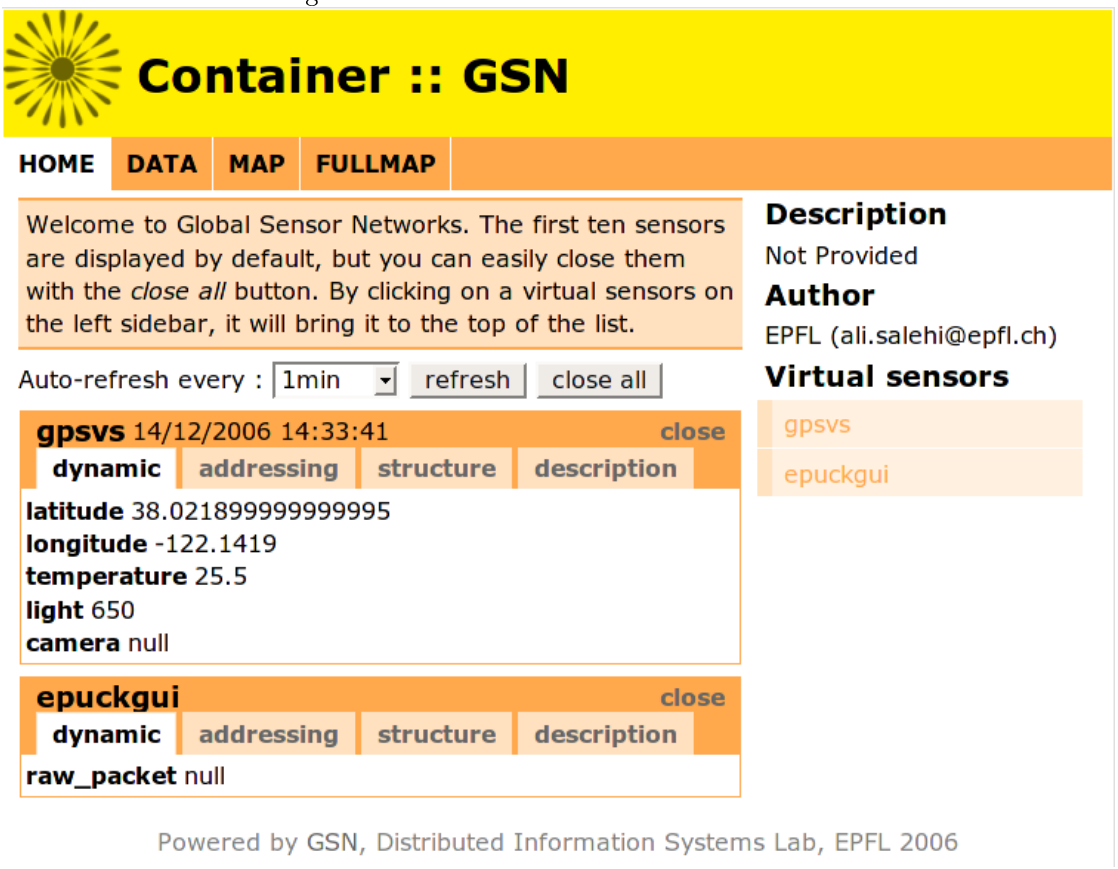


Figure 2.2: The three autonomous components of GSN

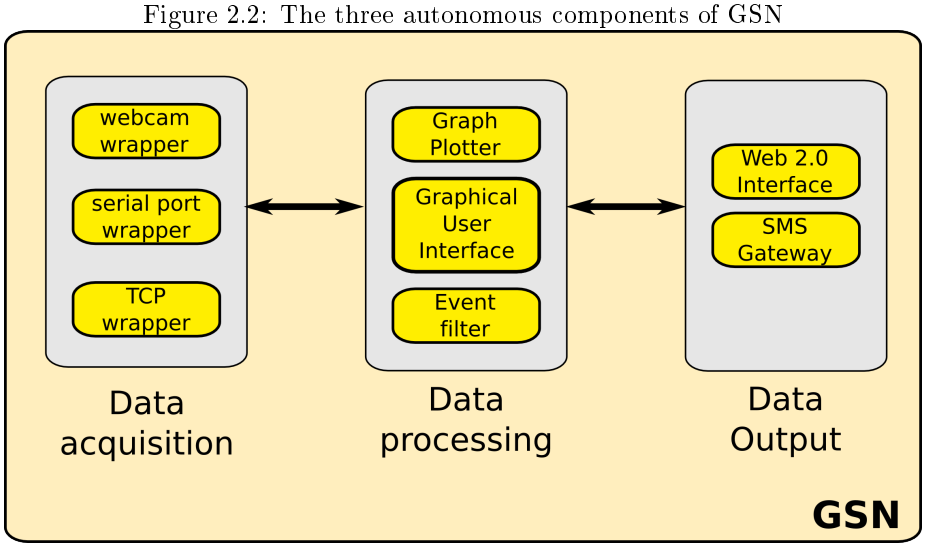
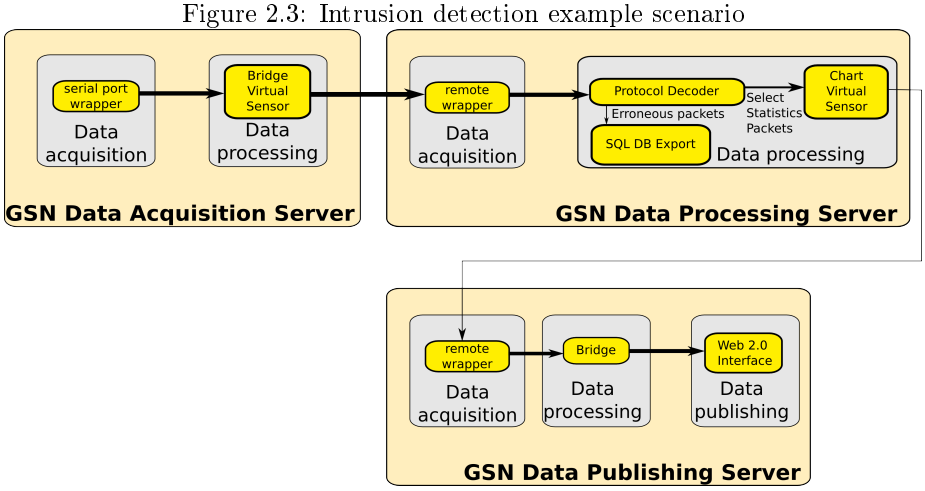


Figure 2.3: Intrusion detection example scenario



This can be done to distribute the load, to deploy GSN on remote, resource-constrained devices or to interconnect many data sources into complex data processing scenarios.

For example, a user may deploy a network of wireless sensors in a building for intrusion detection. Data is collected through multiple, small computers connected to one of these sensors. Such data collection stations are often called *sinks*. Each sink receives data from the neighbouring sensors. To reduce hardware costs and power consumption, these sinks are resource constrained: no hard disk, only a serial port connector and a ethernet or wireless local area network connection. Each sink can run GSN configured to only acquire data.

A GSN data processing server runs in the basement of the building or even in a remote location, possibly hosted in a colocation facility. This server knows about each of the data acquisition servers and registers to them as a data consumer. It will receive their data, analyzes the network packets, logs all erroneous packets in a separate database for off-line manual analysis and sends all statistics packets to the graph generator.

A third GSN server gets the refined data and also runs the Apache web server for performance reasons. It allows identified web users to see plots of the number of erroneous messages in the last 24 hours (for application debugging and traffic spoofing detection), plots with the number of openings for each door and the time at which they occured and the photos of the last 25 persons who entered the building. It also shows a zoomable satellite picture of the enterprise campus to locate more easily the possible source of intrusion.

An idea on how to configure GSN to do that is shown in Figure 2.3. For the sake of simplicity, only one instance of each server type is represented.

Chapter 3

GSN Architecture

3.1 Generalities

GSN is composed of three parts: data acquisition, data processing, and output dispatching (also called notification subsystem). Most users will probably focus on the second part, data processing.

3.2 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.

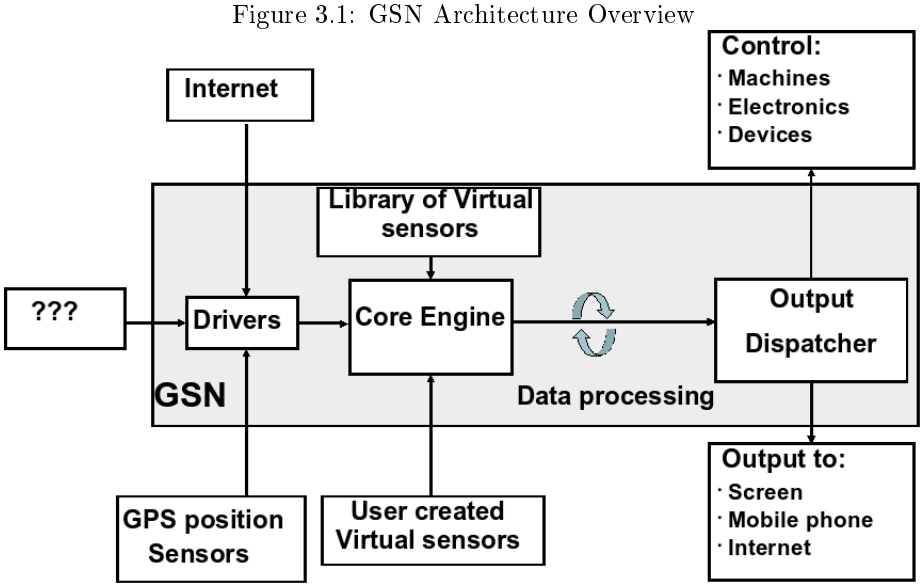
3.2.1 GSN Wrappers

GSN can receive data from various data sources. This is done by using so called *wrappers*. They are used to encapsulate the data received from the data source into the standard GSN data model, called a *StreamElement*. A StreamElement is an object representing a row of a SQL table.

Each wrapper is a Java class that extends the *AbstractWrapper* parent class. Usually a wrapper initializes a specialized third-party library in its constructor. It also provides a method which is called each time the library receives data from the monitored device. This method will extract the interesting data, optionnally parse it, and create one or more StreamElement(s) with one or more columns. From this point on, the received data has been mapped to a SQL data structure with fields that have a name and a type. GSN is then able to filter this using its enhanced SQL-like syntax. You will learn more about that in section 3.3.1.

A wrapper is implemented in a Java class. For simplicity, GSN uses short names to refer to these wrappers. These associations are defined in the file `conf/wrappers.properties`. For now on it is assumed that you use the default names provided at installation time.

Figure 3.1: GSN Architecture Overview



Each of the standard wrappers is described below with some documentation on how to use it.

You can refer to Table 14 for a quick overview of the available wrappers.

3.2.2 Remote Wrapper

A virtual sensor can get data from another virtual sensor (Inter-GSN communication). If this other virtual sensor is running on another GSN server, the communication between these two servers is made with the XML-RPC protocol. This means that you can actually feed data to GSN from your own custom software written in any language supporting XML-RPC.

The Remote Wrapper is a special wrapper that allows GSN to treat a remote virtual sensor (located on anthor machine in the network) as a data source for another virtual sensor. This data source can be configured on the same GSN server (in this case, we say that they are defined in the same container), or it can be run on another computer (hence the ip and port parameters of this wrapper).

This wrapper takes three mandatory parameters. `HOST` specifies the network address (DNS name or IP address) on which is running the GSN instance we want to connect to (when connecting to the same GSN instance, use the `localhost` name or the loopback address `127.0.0.1`). `PORT` is the port number on which this GSN instance is listening (the standard port for GSN is `22001`). `name` is the name of the virtual sensor to which we want to connect. Note that this wrapper uses the very same XML-RPC style communication used by GSN.

3.2.3 Local Wrapper (InVMPipeWrapper)

This is a special form of the Remote wrapper which always points to the GSN instance running on the same machine. Using this wrapper you can combine

Table 3.1: Description of GSN wrappers

Wrapper	Short name	Description	page
Remote wrapper	`remote`	Enables a virtual sensor to use another virtual sensor as its data source.	13
TinyOS 1.x	`tinyos1x`	Communicates with any device running the TinyOS 1.x operating system.	15
TinyOS 2.x	`tinyos2x`	Communicates with any device running the TinyOS 2.x operating system.	15
Serial Wrapper	`serial`	Reads and send data on the serial port (RS-232 interface), real or virtual (e.g. Serial Port Profile over BlueTooth wireless link).	17
UDP Wrapper	`udp`	Opens a UDP socket on a configured port and reads data on it.	18
System Time	`system-time`	Generates events from system clock every t milliseconds, with the system time at which the event was generated.	19
HTTP Get handler	`http-get`	Polls sensor readings from a remote web server (e.g., AXIS wireless camera).	19
USB Webcams	`usb-cam`	Polls a USB camera[1] periodically to grab a picture.	19
Memory Monitor	`memory-usage`	Periodically generates memory usage statistics. Current version gives information on Heap, non-heap and the number of objects awaiting finalization in the virtual machine running GSN.	20

Table 3.2: Parameters for Remote wrapper

Parameter name	Type	Mandatory	Description	Default
HOST	Network name or IP address	Yes	The network address on which the GSN instance that we want to connect to is running.	None
PORT	Integer between 1 and 65535	Yes	The TCP port on which the GSN instance that we want to connect to is running.	None
name	String	Yes	The name of the virtual sensor name we want to get data from.	None

two or more virtual sensors together and make a chain of data processing. For using this wrapper both virtual sensor should be located on a very same GSN instance. This wrapper accepts only one parameter which is called "name" and points to the name of the virtual sensor acting as a data source. Note that if both virtual sensors are on the same machine you can use both the remote and local wrapper but the latter perform much faster than the former. The remote wrapper suffers from the network communication overhead while the local wrapper is desgined to bypass this layer hence making it faster.

3.2.4 TinyOS

With default GSN installation we provide sample wrappers to interact with TinyOS based sensor networks (both version 1.x and 2.x). The TinyOS wrapper is actually a java program that interacts with the serial forwarder (provided 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.

Note for TinyOS 1.x users, due to strange java package management in TinyOS 2.x distribution, 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 resolve this, you need to edit the generated message file and modify it so that it only uses class files from the TinyOS 1.x (all classes must be imported from net.tinyos1x package).

Once you generated the Java representation of the communication message structures, you need to write a wrapper class for your message. Assuming you have a message class called MyMessage which has three fields, node_id, temperature and light, here is a sample wrapper class to bind your network

with GSN :

```
//import net.tinyos1x.message.Message; // Only for TinyOS 1.x
//import net.tinyos1x.message.MessageListener; // Only for TinyOS 1.x
//import net.tinyos1x.message.MoteIF; // Only for TinyOS 1.x
import net.tinyos.message.Message; // Only for TinyOS 2.x
import net.tinyos.message.MessageListener; // Only for TinyOS 2.x
import net.tinyos.message.MoteIF; // Only for TinyOS 2.x
import net.tinyos.packet.BuildSource; // Only for TinyOS 2.x
import net.tinyos.packet.PhoenixSource; // Only for TinyOS 2.x
public class MyWrapper extends AbstractWrapper implements MessageListener {
private final DataField [] structure = new DataField[] {
new DataField("node_id","int","The node id the mote."),
new DataField("temperature","double","Presents the temperature sensor."),
new DataField("light","double","Presents the light sensor.")
};
public boolean initialize ( ) {
AddressBean addressBean = getActiveAddressBean( );
String host = addressBean.getPredicateValue( "host" ); // The ip of the machine running serial forwarder.
if ( host == null || host.trim( ).length( ) == 0 ) {
logger.warn( "The >host< parameter is missing from the RemoteWrapper wrapper." );
return false; //initialization failed, the host name is not provided in the parameters.
}
int port; // The port of the machine running serial forwarder.
String portRaw = addressBean.getPredicateValue( "port" );
if ( portRaw == null || portRaw.trim( ).length( ) == 0 )
{ logger.warn( "The >port< parameter is missing from the RemoteWrapper wrapper." );
return false;// initialization failed, the port no. is not provided in the parameters.
} try {
port = Integer.parseInt( portRaw );
if ( port > 65000 || port <= 0 ) throw new Exception( "Bad port No" + port );
```

```
} catch ( Exception e ) {
logger.warn( "The >port< parameter is not a valid integer for the
    RemoteWrapper wrapper." );
return false; // initialization failed, the port no. is not valid.
}
if ( logger.isDebugEnabled( ) )
logger.debug( "The SFWrapperDS connects to the Serial Forwarder
    interface at *" + host + ":" + port + "*" );
setName( "TinyOS-My-Wrapper") ;
try {
PhoenixSource reader=BuildSource.makePhoenix( BuildSource.makeSF(
    host , port ) , null );// Only for TinyOS 2.x
reader.start( ); // TinyOS 2.x
moteif = new MoteIF( reader ); // Only for TinyOS 2.x
// moteif = MoteIF (host,port); // Uncomment this, only for TinyOS
    1.x
} catch ( Exception e ) {
logger.error( e.getMessage( ) , e );
return false; // Serial Forwarder client couldn't establish the con-
    nection.
}
moteif.registerListener( new MyMessage( ) , this );
return true; //initialization is successful.
}
public synchronized void finalize ( ) {
moteif.deregisterListener( new SensorScopeDataMsg( ) , this );
}
public void messageReceived ( int toAddr , Message message ) {
MyMessage msg = (MyMessage) message;
// (If needed) extract/parse/convert the sensor measurements to
    some meaningful values
postStreamElement( nodeid,sensor1,sensor2 );
}
public DataField [] getOutputFormat ( ) { return structure; }
```

3.2.5 Serial Wrapper

Todo : Jaggi has added several options to this wrapper, I'll combine them in a few days.

The short name of this wrapper is `serial`.

The mandatory parameters are `HOST`, `PORT` and `serialport`. The latest one specifies on which serial port the wrapper should listen. The syntax is platform dependent: on Windows systems it will be `COM1`, `COM2`... and on Unix systems this will be `/dev/ttyS0`, `/dev/ttyS1`...

In its default mode of operation, each time it receives a byte sequence, it will be published in a StreamElement under the name `RAW_PACKET(varchar)` and the type `BINARY`. It is up to the virtual sensor getting data from the wrapper to analyze the meaning of the received bytes.

If the optional parameter *inputseparator* is defined, then the received data is not immediately published. Instead, the wrapper splits the data using the parameter value as a separator. The separator is never published. If there is remaining data after the last separator, it is stored in a temporary buffer. The next byte sequence to be received by the wrapper will be concatenated to the temporary buffer before attempting to split it.

As an example, if the separator is `:` and the received data is `abc:def:g` , then two StreamElements (two rows) are generated. The first one contains `abc` in the `RAW_PACKET` field and the other one contains `def`. `g` is stored in the temporary buffer. Imagine that the wrapper now receives `hijklm:` . Then only one StreamElement will be generated, with the value `ghijklm` and the temporary buffer will be emptied.

Table 3.3: Parameters for serial wrapper

Parameter name	Type	Mandatory	Description	Default
`serialport`	String	Yes	The serial port on which the wrapper should listen to.	None
`baudrate`	Number	No	The speed at which the data is arriving on the port, in bits per second.	9600
`inputseparator`	String	No	If this parameter is defined, the serial port wrapper will split incoming data using this value as a separator. The separator is not published.	None

3.2.6 UDP Wrapper

The short name of this wrapper is `udp`. It allows GSN to receive arbitrary data on a UDP port of the machine on which it is running. There is only one parameter, named `port`. Received data is published under the type `BINARY` and the name `RAW_PACKET`.

Table 3.4: Parameters for UDP wrapper

Parameter name	Type	Mandatory	Description	Default
`port`	Integer between 1 and 65535	Yes	The UDP port on which the wrapper should listen to.	None

3.2.7 System Time

The short name for this wrapper is `system-time`. It generates a StreamElement object every `clock_periods` milliseconds, with the timestamp at which the object was generated.

Table 3.5: Parameters for System Time wrapper

Parameter name	Type	Mandatory	Description	Default
`clock_periods`	Integer	No	Time interval between two events.	1000 ms

3.2.8 Http-Get Wrapper Wireless Camera

This wrapper polls a sensor data from a web server using http get requests. This can be used with any kind of sensor providing access to its sensor data via a web server. As a sample, we provide a netcam virtual sensor which uses this wrapper to interact with Axis 206W networked camera. This wrapper has the following parameters :

Table 3.6: Parameters for Http Get wrapper

Parameter name	Type	Mandatory	Description	Default
`rate`	Integer	No	Time interval between two events.	2000 ms
url	String	Yes	The URL of the web server serving sensor readings. Becareful about the especial characters in XML.	N/A

3.2.9 USB Webcams

Because of a license incompatibility, the wrapper for OV511/518 wired usb cameras is not distributed with the standard installation of GSN.

You can find the wrapper (the java files) from the svn repository of GSN at `http://www.sourceforge.net/projects/gsn`.

To use the ovcam511/518 wrapper with linux:

1. Get the driver from `http://ovcam.org/ov511/`, compile it and install it.

2. Download the Java Media Framework for your platform and install it. At this stage, the JMF should detect your webcam. If it does not, most probably

the ov511 driver was not correctly installed. Please refer to the documentation of your distribution.

3. Once the webcam is detected by the JMF, On linux : sudo rmmod ov511; sudo modprobe ovcamchip;sudo modprobe ov511

4. The system environment or the java execution environment (in Eclipse) must be configured:

```
LD_LIBRARY_PATH=/home/ali/download/JMF-2.1.1e/lib:
  /usr/local/java/jdk1.5.0/jre/lib/i386:
  /usr/local/java/jdk1.5.0/jre/lib/i386/client:
  /usr/local/java/jdk1.5.0/jre/lib/i386/xawt
LD_PRELOAD=/usr/local/java/jdk1.5.0/jre/lib/i386/libjawt.so
```

The short name for this wrapper is `usb-cam`. You can configure it to run in live mode or not (mandatory parameter `live-view`).

Table 3.7: Parameters for OV511 USB webcams

Parameter name	Type	Mandatory	Description	Default
`live-view`	Boolean (as String: `true` or `false`)	Yes	Sets the web cam in live mode.	None.

3.2.10 Memory Monitor

This wrapper queries periodically the virtual machine in which gsn is running. It produces statistics on memory usage. The query rate can be configured with the optional parameter rate. The short name for this wrapper is memory-usage.

Table 3.8: Parameters for memory monitoring wrapper

Parameter name	Type	Mandatory	Description	Default
`rate`	Integer	No	Defines the rate at which data is produced.	1000

3.3 Data filtering and processing

GSN provides two complementary mechanisms to work on your data.

The first one is based on a SQL syntax enhanced with specialized semantics for timed sliding windows and event counting.

The second one allows to manipulate data with specialized programs called *virtual sensors*. GSN comes with a library of virtual sensors that you can use without programming. If you have more sophisticated needs, you can write your own virtual sensors (See the Developer's Guide, chapter 4).

GSN always processes the data according to a virtual sensor configuration. If you only want to use the SQL filtering mechanism, without any data transformation, you can use the *BridgeVirtualSensor* (see section 3.3.2.3).

If you don't want to use the SQL filtering mechanism, simply select all data from the wrapper.

3.3.1 The SQL syntax

3.3.2 Virtual Sensors

3.3.2.1 Introduction

Virtual sensors are small Java programs that register to GSN with a specific SQL query for their data input. This query is configured by the user. When GSN receives data that matches the filter at the entry of a virtual sensor, this data is sent to the virtual sensor, which usually performs some sort of operation depending of the received data, and finally publishes some data (it may also produce nothing).

Virtual sensors are configured in the virtual-sensors directory. You can edit the configuration of a virtual sensor online while GSN is running, because GSN periodically scans this directory for updates. This can be very useful when you are learning how to use GSN: you can immediately see the effect of modifying a query.

3.3.2.2 The virtual-sensor xml configuration file

A virtual sensor configuration file starts with the `virtual-sensor` tag. It takes three parameters, `name`, `priority` and `password`. The first one is mandatory and arbitrary (the only constraint is that the name should be unique in this GSN configuration), and the two others are optional. 0 is the highest priority and 20 is the lowest. The default priority is 10.

The first tag inside virtual-sensor is usually `processing-class`. The first tag inside this group is `class-name`, and gives the class name of the Virtual Sensor to be used in this configuration. After this comes an optional `init-params` section.

We will work with an example to better understand these concepts. We present here a configuration for the ChartVirtualSensor that gets data from another virtual sensor, named `MemoryMonitorVS`. These two configurations can be found in the default installation of GSN under the names *memoryDataVS.xml* and *memoryPlotVS.xml*.

```
<virtual-sensor name="MemoryPlotVS" >
  <processing-class>
     <class-name>gsn.vsensor.ChartVirtualSensor</class-name>
        <init-params>
            <param name="input-Stream">DATA</param>
            <param name="title">GSN Memory Usage</param>
            <param name="type">ANY</param>
            <param name="height">200</param>
            <param name="width">300</param>
            <param name="vertical-axis">Sensor Readings</param>
```

Table 3.9: GSN built-in virtual sensors

Virtual Sensor	Class name	Description	page
Bridge	gsn.vsensor.BridgeVirtualSensor	This VS only acts as a bridge and does not modify data. It can be used to forward data directly from a wrapper to the notification system, or to only use SQL filtering.	26
Chart	gsn.vsensor.ChartVirtualSensor	Generates graphs of received data.	26
Stream Exporter	gsn.vsensor.StreamExporterVirtualSensor	Exports received data to any supported database.	26
Web Interaction	gsn.vsensor.WebInteractiveVirtualSensor		
Host Controller Interface Graphical User Interface	gsn.vsensor.HCIProtocolGUIVS	A Graphical User Interface for sending commands to a hardware device. Two protocols are currently supported.	

```
            <param name="history-size">100</param>
        </init-params>
      <output-structure>
        <field name="DATA" type="binary:image/jpeg"/>
      </output-structure>
    </processing-class>
    <description>My chart virtual sensor<description>
    <life-cycle pool-size="10"/>
    <addressing>
      <predicate key="geographical">BC building EPFL</predicat>
    </addressing>
    <storage history-size="1" />
    <streams>
       <stream name="DATA" rate="100">
         <source alias="source1" storage-size="1 sampling-rate="1">
            <address wrapper="remote">
               <predicate key="HOST">
                  localhost
               </predicate>
               <predicate key="PORT">
                  22001
               </predicate>
               <predicate key="NAME">
                  MemoryMonitorVS
               </predicate>
            </address>
            <query>
              select HEAP,NON_HEAP,
              PENDING_FINALIZATION_COUNT,TIMED
              from wrapper
            </query>
          </source>
          <query>
           select * from source1
          </query>
        </stream>
      </streams>
  </virtual-sensor>
```

Let us analyze this long file piece by piece. The first part defines which class to use and configures it:

```
<processing-class>
    <class-name>gsn.vsensor.ChartVirtualSensor</class-name>
    <init-params>
         <param name="input-Stream">DATA</param>
         <param name="title">GSN Memory Usage</param>
         <param name="type">ANY</param>
         <param name="height">200</param>
         <param name="width">300</param>
```

```
            <param name="vertical-axis">Sensor Readings</param>
            <param name="history-size">100</param>
        </init-params>
      <output-structure>
        <field name="DATA" type="binary:image/jpeg"/>
      </output-structure>
  </processing-class>
```

This virtual sensor generates graphs from the data it receives. We see here that the graph has a name, "GSN Memory Usage", that its size is 200*300 pixels, that the vertical axis shows the data readings and that the history-size is set to 100. This means that only the latest 100 received values will be plotted. The output-structure describes the data type produced by the virtual sensor. It defines a name and a type. Please consult the documentation of the virtual sensor that you want to use for more information. In this case the ChartVirtualSensor produces a its plots under the name "DATA" which is a jpeg image hence type="binary:image/jpeg".

```
<description>My chart virtual sensor</description>
<life-cycle pool-size="2"/>
<addressing>
   <predicate key="geographical">...</predicate>
</addressing>
```

The description field is shown to the user through the web interface, and you can also use it to help you remember what this specific configuration does.

The `life-cycle pool-size` option is a performance parameter. It is usually safe to keep the default value. It defines the maximum number of instances of this virtual sensor (with this configuration). This can happen when the processing method of the virtual sensor takes a long time to complete, and / or when data arrives at high speed. If all instances are busy, then the data will be dropped.

The `addressing` section is used to describe the location of the sensor (physical and logical address). This can be used to annotate a virtual sensor. In GSN we have two special addressing keys, latitude and longitude, which are used by GSN's web interface to visualize the phyiscal location of the virtual sensor on the google map interface. You can use them like below :

<addressing>
<predicate key="type">mica sensor temperature light</predicate>
<predicate key="LATITUDE">37.4</predicate>
<predicate key="LONGITUDE">-122.1</predicate>
</addressing>

Note that the addressing part is optional and can be omitted.

```
  <storage history-size="1" />
```

`storage`'s `history-size` defines the number of streamElements produced by this virtual sensor that we want to store in the database. It does not impact the logical processing of data streams. If the history-size is not provided, GSN stores all the sensor data in the specified database (the gsn.xml file). Note that, if you use in memory database you must put upper limit on the history size otherwise the Java virtual machine will run out of memory at some point.

```
<streams>
<stream name="DATA" rate="100">
<source alias="source1" storage-size="1" sampling-rate="1">
<address wrapper="local">
<predicate key="HOST">localhost</predicate>
<predicate key="PORT">22001</predicate>
<predicate key="NAME">MemoryMonitorVS</predicate>
</address>
<query>
  select HEAP,NON_HEAP,
    PENDING_FINALIZATION_COUNT,TIMED
  from wrapper
</query>
</source>
<query>select * from source1</query>
</stream>
</streams>
```

The streams section is very important. It tells GSN what data it should send to the virtual sensor you are using (how to feed the virtual sensor). A virtual sensor can receive data from one or more streams. In this case there is only one stream, that we name `DATA`. The `rate` parameter is a performance tuning parameter. It defines the minimum interval in milliseconds between two calls to this virtual sensor. If there is data available for the virtual sensor in less than this value, then the data is silently dropped hence it is better to remove it when debuging.

This xml structure allows to perform `JOIN` SQL operations on the incoming data streams, if more than one source is defined. In the stream queries, we can refer to a source by the value of `alias` attribute of that source. The combination of `storage-size` and `slide` attributes defines a sliding window over the data. The `storage-size` attribute specifies the size of the window and the `slide` attibute specifies the sliding predicate. Both of these attributes can be either time-based or count-based. A time-based value can be specified by adding a *s*, *m*, or *h* character after the numeric value.

Here there is only one source, and we select everything from it in `select * from source1`. Our source gets its data from the `remote` wrapper. This wrapper is used to get data from another virtual sensor. This other virtual sensor can be running in the local GSN instance or on a remote GSN server, hence the wrapper name. In this case we use the local wrapper which points to the local GSN instance. This wrapper gets one mandatory parameter : Name which is the name of the virtual sensor on the local box. We could also use the remote wrapper to do the same job (but less efficiently). For using the remote wrapper we need to provide three mandatory parameters: `HOST` and `PORT` tells how to contact the GSN instance and `NAME` specifies the virtual sensor from which we want to receive data from. The query specified after the addressing section is sent to the remote GSN server. Here we select some fields that are produced by the virtual sensor.

3.3.2.3 Bridge Virtual Sensor

This virtual sensor immediately publishes any data that it receives without modifying it. The StreamElement object is not modified.

It does not take any parameter. Its main use is to filter and join sensor data with SQL without any other kind of post processing operation on the data. For using this wrapper the output structure of the virutal sensor must match strictly with the structure of the query.

3.3.2.4 Chart Virtual Sensor

This virtual sensor generates graphs from the data it receives. This is a complex VS because it can be computationally intensive, especially if it is frequently called. This virtual sensor using the JFreeChart library to plot the data.

3.3.2.5 Stream Exporter

This virtual sensor exports the data it receives to the database of your choice. This can be interesting when debugging your GSN configuration or to easily back up critical data on an independent machine. It can also be used to log unexpected events for later off-line, manual analysis. It can receive any number of input streams. Each one will be saved into a separate table named after the input stream.

It requires a JDBC URL and a user name and password so that it knows where is the database server and how to authenticate. This virtual sensor inserts the data into the specified table. If the table doesn't exist, GSN first create ths table and if the table exists, GSN first verifies the structure of the table with the structure of the produced stream elements. If the structure matches, GSN starts inserting data into the table otherwise it stops with an error message to inform the user about the uncompatible strucutres.

3.4 Data publishing

3.4.1 Web Interface

GSN ships with an elegant and easy to use web interface. The only thing you have to do is to open a web browser and go the following address: `http://127.0.0.1:22001` .

3.4.1.1 GoogleMaps integration

GSN can associate your data with GPS positions and then display these on a world map retrieved from Google's GoogleMaps service. You need a special identification key from Google. For more information, please refer to the documentation file `doc/README.txt`, section "*How to use GoogleMaps with GSN*".

3.4.2 Email notifications

In order to use email or SMS (Short Messaging System, text messages for GSM phones) notifications, you need to download two third-parties libraries that we are not allowed to distribute.

Table 3.10: Parameters for Plotter VS

Parameter name	Type	Mandatory	Description	Default
`input-stream`	String	Yes	The name of the stream to plot.	None
`title`	String	Yes	The graph title.	None
`vertical-axis`	String	Yes	A legend for the vertical axis.	None
`width`	Integer	No	Graph width in pixels	640
`height`	Integer	No	Graph height in pixels	480

Table 3.11: Parameters for Stream Exporter

Parameter name	Type	Mandatory	Description	Default
`url`	String	Yes	A JDBC url that specifies how to connect to the database server.	None
`user`	String	Yes	The user name for authentication with database server.	None
`password`	String	Yes	The password for authentication with database server.	None
`table`	String	No	The name of the table into which GSN will store the sensor data.	None

The first one is the *JavaBeans Activation Framework* (JAF), which you can find at `http://java.sun.com/products/javabeans/jaf/index.jsp`. After downloading it, you have to copy the file `activation.jar` in the `lib` directory.

The second one is *JavaMail*. This can be found at `http://java.sun.com/products/javamail/downloads/index.html`. All included JAR files should be copied in the same `lib` directory.

3.4.3 SMS notifications

Part II

Developer's Guide

Chapter 4

Writing a Virtual Sensor Processing Class

In GSN, a virtual sensor processing class is a piece of Java code which performs the final stage of the processing. The inputs and the output structure of the virtual sensor is specified in the virtual sensor description file (VSD). Once a virtual sensor is loaded, GSN prepares an object pool for the virtual sensor. This pool used for both improving the performance and enforcing the maximum resource consumption in the highly loaded environemnts. You can specify the number of instances per virtual sensor using the <life-cycle pool-size="X"/> element. Every virtual sensor processing class (VSPC) extends the AbstractVirtualSensor class.

4.1 The AbstractVirtualSensor class

All virtual sensors are subclass `of the AbstractVirtualSensor` (package `gsn.vsensor`). It requires its subclasses to implement the following three methods:

- `public boolean initialize()`
- `public void dataAvailable(String inputStreamName, StreamElement se)`
- `public void finalize()`

`initialize()` is the first method to be called after object creation. It should configure the virtual sensor according to its parameters, if any, and return `true` in case of success, `false` if otherwise. If this method returns false, GSN will generate an error message in the logs and stops using the virtual sensor.

`finalize()` is called when GSN destroys the virtual sensor. It should release all system resources in use by this virtual sensor. This method is typically called when we want to shutdown the GSN instance.

`dataAvailable` is called each time that GSN has data for this virtual sensor, according to its configuration. If the virtual sensor produces data, it should encapsulate this data in a `StreamElement` object and deliver it to GSN by calling `dataProduced(StreamElement se)`.

Note that a Virtual Sensor should always use the same StreamElement structure for producing its data. Changing the structure type is not allowed and trying to do so will result in an error. However, a virtual sensor can be configured at initialization time what kind of StreamElement it will produce. This allows to produce different types of StreamElement by the same VS depending on its usage. But one instance of the VS will still be limited to produce the same structure type. If a virtual sensor really needs to produce several different stream elements, user must provide the set of all possibily fields in the stream elements and provide Null whenever the data item is not applicable.

4.2 Reading Initialization Parameters

As you noticed in the Chart virtual sensor, a virtual sensor can receive parameters from the virtual sensor description file and use them in the initialization process. An initialization parameter has a name and a value both are in the form of String. The parameter name is cases insensitive.

For example, in order to read the value of the "my-parameter" parameter form the following code snippets

...
<processing-class>
<class-name>gsn.vsensor.ChartVirtualSensor</class-name>
<init-params>
<param name="my-parameter">123456</param>
</init-params>
...
You can use the following java code in the initialize method :
TreeMap < String , String > params = getVirtualSensorConfiguration( ).getMainClassInitialParams( );
String value = params.get("my-parameter");
if (value ==null){ // parameter is missing.
// use default value or return false with an error message.
}

4.3 The StreamElement class

A StreamElement is a GSN class that encapsulates data. It has a data types structure (a DataField array), a data values structure (a Serializable array) and a timestamp.

4.4 Writing your own graphical user interface

A virtual sensor is not limited to raw data processing. You can call any other Java library, including Swing classes. An introduction to GUI programming is outside the scope of this document. You can have a look at the HCIProtocolGUIVS class to see how such an interface can be implemented.

A simple way to go is to create the graphical components (like a JFrame) in the `initialize()` method and at the same time define the events logic (`eventListeners`...). In the `dataAvailable()` method, received data can be

sent to graphical components to present the information to the user. Beware that there may be concurrency problems since your GUI is running with the Swing event thread while your virtual sensor is run by a GSN thread.

4.5 Feedback channel to a Virtual Sensor

public boolean dataFromWeb (String action,String[] paramNames, Serializable[] paramValues)

Chapter 5

Writing a Wrapper

5.1 A quick how-to

All wrappers subclass `gsn.wrapper.AbstractWrapper`. Subclasses must implement four methods:

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

Each wrapper is a thread in the GSN. If you want to do some kind of processing in a fixed time interval, you can override the `run()` method. The run method is useful for time driven wrappers in which the production of a sensor data is triggered by a timer.

Optionally, you may wish to override the following one:

- `boolean sendToWrapper(String action, String[] paramNames, Object[] paramValues)`

5.1.1 initialize()

This method is called after the wrapper object creation. For more information on the life cycle of a wrapper, see section 5.3 on page 38. The complete method prototype is `public boolean initialize()`.

In this method, the wrapper should try to initialize its connection to the actual data producing/receiving device(s) (e.g., wireless sensor networks or cameras). The wrapper should return `true` if it can successfully initialize the connection, `false` otherwise.

GSN provides access to the wrapper parameters through the `getActiveAddressBean().getPredicateValue("parameter-name")` method call.

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

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

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

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

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

```
setUsingRemoteTimestamp(true);
```

5.1.2 finalize()

In the `public void finalize()` method, you should release all the resources you acquired during the initialization procedure or during the life cycle of the wrapper. Note that this is the last chance for the wrapper to release all its reserved resources and after this call the wrapper instance virtually won't exist anymore.

For example, if you open a file in the initialization phase, you should close it in the finalization phase.

5.1.3 getWrapperName()

`public String getWrapperName()` returns a name for the wrapper.

5.1.4 getOutputFormat()

`public abstract DataField[] getOutputFormat()` returns a description of the data structure produced by this wrapper.

This description is an array of `DataField` objects. A `DataField` object can be created with a call to the constructor `public DataField(String name, String type, String Description)`. The name is the field name, the type is one of GSN data types (`TINYINT, SMALLINT, INTEGER, BIGINT, CHAR(#), BINARY[(#)], VARCHAR(#), DOUBLE, TIME`. See `gsn.beans.DataTypes`) and `Description` is a text describing the field.

The following examples should help you get started.

5.1.4.1 Wireless Sensor Network Example

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

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

5.1.4.2 Webcam Example

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

```
public DataField[] getOutputFormat() {
  DataField[] outputFormat = new DataField[1];
  outputFormat[0] = new DataField("Picture", "binary:jpeg",
      "Picture from the Camera at room BC143");
  return outputFormat;
}
```

5.1.5 run()

Implementation of the `run()` method: as described before, the wrapper acts as a bridge between the actual hardware device(s) and GSN, thus in order for the wrapper to produce data, it should keep track of the newly produced data items. This method is responsible for forwarding (and possibly filtering or aggregating) the newly received data from the hardware to the GSN engine.

You should not try to start the thread by yourself: GSN takes care of this. The method should be implemented like below :

```
try {
//The delay needed for the GSN container to initialize itself.
//Removing this line might cause hard to find random exceptions
Thread.sleep (2000);
} catch (InterruptedException e1) {
  e1.printStackTrace();
}
while(isActive()) {
  if(listeners.isEmpty())
   continue;
  if (isLatestReceivedDataProcessed == false) {
    //Application dependent processing ...
    StreamElement streamElement = new StreamElement ( ...);
    isLatestReceivedDataProcessed = true;
    publishData ( streamElement );
  }
}
```

5.1.5.1 Webcam example

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

```
while(isActive()) {
  byte[] received_image = null;
  if(listeners.isEmpty())
    continue;
  received_image= getPictureFromCamera();
  StreamElement streamElement = new StreamElement(
     new String[] { "PIC" },
     new Integer [] { Types.BINARY },
     new Serializable[] {received_image},
     System.currentTimeMillis ())
  );
  publishData(streamElement);
  Thread.sleep(5*1000); // Sleeping 5 seconds
}
```

5.1.5.2 Data driven systems

Compared to the previous example, we do sometimes deal with devices that are data driven. This means that we don't have control neither on when the data produced by them (e.g., when they do the capturing) nor what is the rate of the data received from them.

For example, having an alarm system, we don't know when nor when we are going to receive a packet, neither how frequent the alarm system will send data packets to GSN. These kind of systems are typically implemented using a callback interface. In the callback interface, one needs to set a flag indicating the data reception state of the wrapper and control that flag in the `run` method to process the received data.

5.1.6 sendToWrapper()

Most devices, in addition to producing data, can also be controlled. You can override the method

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

if you want to offer this possibility to the users of your wrapper.

You can consult the `gsn.wrappers.general.SerialWrapper` class for an example.

5.2 A detailed description of the AbstractWrapper class

In GSN, a wrapper is piece of Java code which acts as a bridge between the actual data producing/receiving device (e.g., sensor network, RFID reader, webcam...)

and the GSN platform. A GSN wrapper should extend the `gsn.wrapper.AbstractWrapper` class. This class provides the following methods and data fields:

```
public static final String TIME_FIELD = "TIMED";
public AddressBean getActiveAddressBean();
public int getListenersSize();
public ArrayList<DataListener> getListeners();
public CharSequence addListener(DataListener dataListener);
public void removeListener(DataListener dataListener);
public int getDBAlias();
public boolean sendToWrapper(String action,
  String[] paramNames, Object[] paramValues)
    throws OperationNotSupportedException;
// Abstract methods
public abstract boolean initialize();
pulic abstract void finalize();
public abstract String getWrapperName();
public abstract DataField[] getOutputFormat();
```

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

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

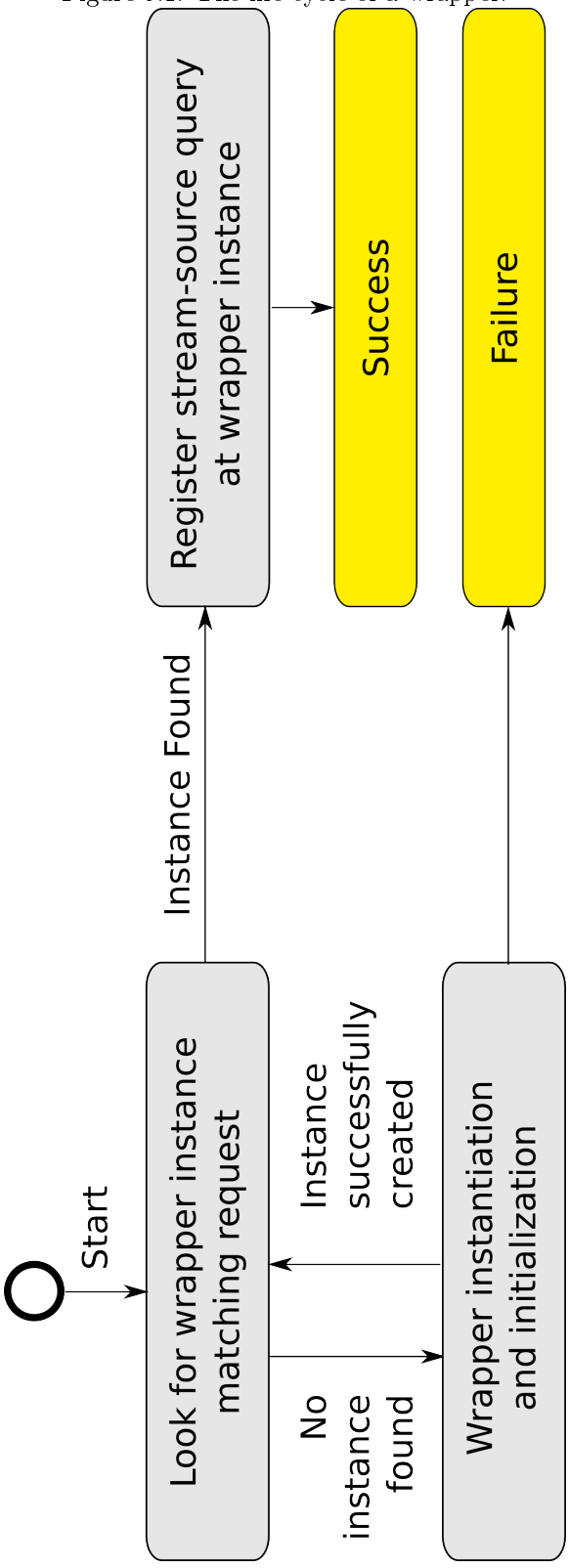
The default implementation of the afore-mentioned method throws `OperationNotSupportedException` exception because the wrapper doesn't support this operation. This design choice is justified by the observation that not all kind of devices (sensors) can accept data from a computer. For instance, a typical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

5.3 The life cycle of a wrapper

An instance of a wrapper is created whenever a *Wrapper Connection Request* (WCR) is received by the *Wrappers Repository* (WR). The WCRs are generated whenever GSN wants to activate a new virtual sensor. A WCR is generated for each stream source in the virtual sensor.

A Wrapper Connection Request is an object which contains a wrapper name and its initialization parameters as defined in the Virtual Sensor Configuration file (VSC). Therefore, two WCRs are identicals if their wrapper name and initialization parameters are the same. The Wrappers Repository in a GSN instance is a repository of the active wrapper instances indexed by their WCRs.

Figure 5.1: The life cycle of a wrapper.



Whenever a WCR is generated at the virtual sensor loader, it will be sent to the WR which does the following steps (as illustrated on Figure 5.1 on the previous page):

1. Look for a wrapper instance in the repository which has the identical WCR. If found, WR registers the stream-source query with the wrapper and returns `true`.
2. If there is no such WCR in the repository, the WR instantiates the appropriate wrapper object and calls its `initialize` method. If the `initialize` method returns `true`, WR will add the wrapper instance to the WR. Back to Step 1.
3. If there is no WCR in the repository and the WR can not initialize the new wrapper using the specified initialization parameters and GSN context (e.g., the `initialize` method returns `false`), WR returns `false` to the virtual sensor loader. When the virtual sensor loader receives `false`, it tries the next wrapper (if there is any) . The virtual sensor loader fails to load a virtual sensor if at least one of the stream sources required by an input stream fails.

The two main reasons behind using the wrappers repository are:

- Sharing the processing power by performing query merging.
- Reducing the storage when several stream sources use the same wrappers.

The Wrapper Disconnect Request (WDR) is generated at the `virtual-sensor-loader` whenever GSN wants to release resources used by a virtual sensor. Typically, when the user removes a virtual sensor configuration while GSN is running, the virtual-sensor-loader generates a WDR for each stream source that was previously used by this virtual sensor.

When WR receives a WDR request, it de-registers the stream-source query from the wrapper. If after removing the stream source query from the wrapper, there are no queries registered with this wrapper (e.g., no other stream source is using the considered wrapper), WR calls the `finalize` method of the wrapper instance so that all its allocated resources will be released.

5.4 Questions and Answers

5.4.1 When is the sendToWrapper method called ?

The sendToWrapper method can be only called from a virtual sensor which uses this wrapper. The code in the virtual sensor's class will be something like below:

```
virtualSensorConfiguration.getInputStream(INPUT_STREAM_NAME).
  getSource(STREAM_SOURCE_ALIAS_NAME).
  getActiveSourceProducer().
  sendToWrapper(mydata);
```

So a virtual sensor can send data to the wrapper and the wrapper will forward it (if the `sendToWrapper` method is implemented) to the actual data source.

5.4.2 How does a virtual sensor decide when to send data to the wrapper ?

A virtual sensor will typically decide using the following factors :

1. Based on its internal state, which depends on received data.
2. After an interaction initiated by a user/agent/gsn-instance with the web interface; HTTP (e.g., implementing the `dataFromWeb` method in the virtual sensor, or when another virtual sensor sends data to this sensor server).

Chapter 6

The Web Interface

This is a presentation of the web interface bundled with GSN. It can be used as a starting point for creating another one.

In GSN we separated the look and feel of the web interface from the underlying processing components. This design enables other users to provide their own web interface over the GSN. In this section we describe both the current server side protocol and the currently available web interface which visualizes the data from the server. While the server side component is fixed with GSN, the client side web interface can be changed based on the users and applications.

6.1 ServerSide protocol

In GSN, all the request are first received by the the /gsn (gsn.web.ControllerServlet).
In GSN we provide several request handlers each of them implements the gsn.web.RequestHandler interface. Each request from the clients has a REQUEST parameter to specify the request type, `http://server/gsn?REQUEST=`*type*, where *type* is an integer. If there is no *type* specified, ControllerServlet assumes type 0. Here are the list of the parameters and the Contant representing them.

- 0, or not present: REQUEST_LIST_VIRTUAL_SENSORS.
- 113, REQUEST_OUTPUT_FORMAT.
- 114, REQUEST_ONE_SHOT_QUERY.
- 115, REQUEST_ADDRESSING.
- 116, REQUEST_ONE_SHOT_QUERY_WITH_ADDRESSING.

The RequestHandler interface two methods:

- isValid (HttpServletRequest, HttpServletResponse) verifying the validity of the request (presence and validity of the parameters) or responding by and error message if not.
- handle (HttpServletRequest, HttpServletResponse) which treat the request and send the response in XML format

6.1.1 REQUEST_LIST_VIRTUAL_SENSORS (Default, Zero)

Has the following informations:

- GSN information from the `gsn.xml` file (name, author, email, description) .
- For each valid virtual sensor
  - Virtual Sensor name, description and the timestamp of the most recently produced stream element.
  - The list of the addressing predicates (category="predicate")
  - The list of the output fields,their types and their most recent value. The value of the field is only included in the output if it can be easily converted to String format. If the type of a field is binary (e.g., a JPEG image), as the value, the controller returns a URL pointing to the /field(gsn.web.FieldDownloadServlet) to retrive the actual binary data.

Example:

```
<gsn name="Container" author="EPFL" email="ali.salehi@epfl.ch" description="Not
Provided">
<virtual-sensor name="invmaccess" last-modified="1173280806000"
description="MyVS">
<field name="image" type="binary:image/jpeg">/field?vs=invmaccess&field=DATA&pk=2868</f
<field name="timed" type="long" description="The timestamp associated
with the stream element">1174999988339</field>
<field name="geographical" category="predicate">BC 143, EPFL, Lausanne
</field>
</virtual-sensor>
</gsn>
```

6.1.2 REQUEST_OUTPUT_FORMAT (113)

Provides the output structure of a given virtual sensors. The only parameter needed is name of the virtual sensor the user want the description of (e.g., `http://server/get?REQUEST=113\&name=`*VirtualSensorName*

6.1.3 REQUEST_ONE_SHOT_QUERY (114)

This request is handled by the class gsn.vsensor.http.OneShotQueryHandler. This servlet executes a one shot query over the GSN's internal table and presents the results in the form of the stream elements. This request needs the following parameters:

- name, the name of the virtual sensor [mandatory]
- condition: the WHERE part of the request (ex: LATITUDE $>$ 100 AND light $<$ 10) [optional]

- fields: an array of the requested fields in the stream elements. If not present, * is used [optional]
- window: the number last elements wanted (1 if not present) [optional]

Example, accessing the `http://localhost:22001/gsn?REQUEST=114&name=invmaccess`

```
<result>
<stream-element>
<field name="DATA">/field?vs=invmaccess&field=DATA&pk=5130</field>
<field name="timed">1175001199874</field> </stream-element>
</result>
```

6.1.4 REQUEST_ADDRESSING (115)

Given the name of the virtual sensor, it provides the addressing predicates associated with the virtual sensor. It is very similar to `REQUEST_OUTPUT_FORMAT`.

Example, accessing the `http://localhost:22001/gsn?REQUEST=115&name=invmaccess`

```
<virtual-sensor name="invmaccess" last-modified="1173280806000">
<predicate key="geographical">BC 143, EPFL, Lausanne</predicate>
</virtual-sensor>
```

6.1.5 REQUEST_ONE_SHOT_QUERY_WITH_ADDRESSING (116)

This request is actually the combination of the REQUEST_ONE_SHOT_QUERY and REQUEST_ADDRESSING in one request. The parameters are the union of the specified types. Example: `http://localhost:22001/gsn?REQUEST=116&name=invma`

```
<result>
<stream-element>
<field name="DATA">/field?vs=invmaccess&field=DATA&pk=15102</field>
<field name="timed">1175006917711</field>
<field name="geographical">BC 143, EPFL, Lausanne</field>
</stream-element>
</result>
```

You might ask why we need this request if we can get the same result by combining the 115 and 114 ? The answer is the performance, establishing the TCP connections are expensive.

6.2 Client Side JavaScript

6.2.1 Javascript files

For rapid development, the GSN web interface uses many pieces of existing javascript code which are in the js folder.

The following files are part of the webapp:

- jquery-latest.pack.js: The main jQuery library. The version in use comes from the svn as some needed fixes were not available in the latest version.
- jquery-dom.js: An extension to jQuery which allows easy DOM creation written like $.DIV("class":"test","content");

- jquery.history.js: An extension to jQuery to manage the hash part of the location url (index.html#hash). It is used to allow the page to be fully static html while at the same time still support the browser back button and bookmarking of a specific part of the page.
- dimensions.js: An extension to jQuery to know the page's width and height. Used only in fullmap.html in order to have the container with vsboxes of the right size.
- datepicker.js: Customized extension of jquery used for the datetime helper.
- jquery-tooltip.js: A plugin to have better tooltips than the browser default behaviour which is used for descriptions.
- gsn.js: Our core javascript code. Explanations in the next subsection.

6.2.2 gsn.js core function

Almost all of gsn JavaScript code is regrouped inside gsn.js. It is written using a JavaScript Object Notation and use a GSN object to reduce the risk of conflicts with any future addition of JavaScript code. *vsName*, the name of the virtual sensor (text) is considered as unique, which explains why it is so much used as a parameter. Most of the ajax calls go to the */gsn* servlet. The data part uses the */data* servlet for retrieving data. Binary and image are accessed through the */field* servlet and webinput upload goes to the */upload* servlet.

Here is a brief explanation of the core variables and functions:

- GSN.context: defines which page of gsn is being displayed (home, data, map or fullmap).
- GSN.load(): initializes a page load (begin, tab click & back button).
- GSN.msgcallback(): callback method when the webupload iframe finishes.
- GSN.nav(page): click on the top navigation bar.
- GSN.menu(vsName): click on the virtual sensor on the left bar.
- GSN.loaded: is true after the first load of XML info which calls GSN.init().
- GSN.init(): initializes the gsn title and the left side menu.
- GSN.updatenb: a counter to prevent multiple instances of GSN.updateall().
- GSN.updateall(): Ajax call to update all the sensor displays on the page and the map.
- GSN.addandupdate(vsName): adds a vsbox if it doesn't exist, bring it to front and update it.

The virtual sensor information (dynamic, static and structure) are regrouped in a div box called the vsbox. This div has the class .vsbox and .vsbox-the virtual sensor name. They will be put in the container specified by GSN.vsbox.container. Here are the function of the GSN.vsbox.* part:

- GSN.vsbox.container: the container to put the vsbox (default "#vs").
- GSN.vsbox.add(vsName): creates an empty vsbox.
- GSN.vsbox.bringToFront(vsName): brings a vsbox at the beginning of the container.
- GSN.vsbox.update(vs): updates and shows all the data of the vsbox.
- GSN.vsbox.remove(vsName): removes the vsbox from the container.
- GSN.vsbox.toggle(...): to manage the click on tab.

All the functionality used to display the virtual sensor on the google map is regrouped in the GSN.map.* part. Here is the core functionality:

- GSN.map.loaded: ss true after the initialisation of the google map.
- GSN.map.markers : array containing the displayed markers.
- GSN.map.highlighted : the index of the focused marker or null.
- GSN.map.highlightedmarker : the green marker.
- GSN.map.init(): initializes the google map.
- GSN.map.userchange(): callback after any map change (zoom and map move) and vs toggle to update #hash of URL.
- GSN.map.zoomend(..): callback after any zoom change, used for the tricked followed marker.
- GSN.map.trickhighlighted(): followed marker and top of the others.
- GSN.map.autozoomandcenter(): auto-zoom and center on the visible sensors.
- GSN.map.addMarker(vsName,lat,lon): adds a vs marker on the map.
- GSN.map.markerIsVisible(vsName): returns if a marker is visible.
- GSN.map.toggleAllMarkers(): toggles all markers (visible or not).
- GSN.map.toggleMarker(vsName): toggles marker (visible or not).
- GSN.map.updateMarker(vsName,lat,lon): updates a vs marker position on the map.
- GSN.map.followMarker(vsName): highlights a marker, stops it if called with null name.
- GSN.map.showAllMarkers(): zoom out to see all markers.

The functionality in the data webpage is regrouped in the GSN.data.* part and was only bugfixed and slightly improved. The remaining GSN.util.* functions are basic formatting piece of code that don't need any explanations at all.

6.2.3 Problems faced

6.2.3.1 Dropping tables

The first version of the gsn webapp "template" was using tables for layout which is considered as bad by most web designers. Therefore, it was transformed to a layout using only divs elements and css attributes for styling. The advantages are: cleaner html code, easier to understand, more flexible (as changing only the css could change the whole look & feel) and faster as the size is smaller and the css file can be cached by the client browser. It was quickly rewritten from scratch to match the current gsn website style.

6.2.3.2 IE showstopper

A big problem was identified after the first week of implementation. Some of the functionality used in jQuery library was not compatible with IE. Some rare XML parsing modes totally broke. Hopefully, with the help of the jQuery team, it was resolved. Therefore, a switch was made from jQuery 1.0 to the jQuery from the svn as it was the only fixed version.

Another IE only bug caused problems, because Ajax request in IE are locally cached like any normal http request. Therefore, the webapp wasn't refreshing even if the Javascript was working fine and making ajax calls. It was easily resolved by adding to the XML API servlet various http headers (Expires, Cache-Control,..) to avoid any caching at all.

6.2.3.3 Retrieving dynamic data with ajax

As the webapp is composed only of static html files, the dynamic data has to be pulled from the servlet through JavaScript. It is done easily using the *.ajax* function of jQuery. In the main *GSN.updateall* function which updates all virtual sensors data at every refresh, the webapp calls the servlet at `http://localhost:22001/gsn` using the following code:

```
$.ajax({ type: "GET", url: "/gsn", success: function(data) {
        // use the data variable which contain virtual sensors data
}
```

As shown above, using ajax in jQuery is very straightforward. The retrieved data is in xml and formatted as below:

```
<gsn name="gsn" author="cb" email="cb@epfl.ch" description="none">
  <virtual-sensor name="vs" last-modified="1171105166765">
    <field name="temp" type="long" description="none">-5</field>
  </virtual-sensor>
</gsn>
```

After retrieving the xml data, it is parsed with the *$()* function of jQuery which supports xml data similarly to inline html. The syntax looks as below with the data variable being the ajax reply from the servlet:

```
$("virtual-sensor",data).each(function(){
  var vsname = $(this).attr("name");
});
```

When the *GSN.updateall* function finishes to parse xml and updates the inline html with the new virtual sensors data, the standard Javascript function *setTimeout()* is used in order to call the *GSN.updateall* function again for the next refresh.

In addition, when clicking on the right side menu on a virtual sensor, an ajax call is made as well. The data of the clicked virtual sensor is retrieved using `http://localhost:22001/gsn?name=vsname`. The logic is the same than during a *GSN.updateall* but only the specified virtual sensor is in the xml data. Using the same JavaScript code, it will update only the specified virtual sensor box.

Not much can go wrong with these ajax calls. As specified in its format, the reply will always be fully understood by the JavaScript parsing. The worst that can happen is that the servlet stops responding. In this case, the ajax query will timeouts and never succeeds which will stop any further update. The ajax indicator (animated gif) will turn forever, showing that something went wrong during xml data fetching. The client browser should never freeze nor crash. The user will actually still be able to use the gsn webapp, but will not receive any new updates to the data.

6.2.3.4 XML formatting of virtual sensor

As explained in the design section, the gsn webapp use an xml api to give data. All the information concerning a virtual sensor is regrouped into a single *virtual-sensor* block. All the different types of data (dynamic, structure, predicate,..) are merged into a single entity as it is much more efficient than having to query multiple times the servlet. The overall size of a single block, as it is only text, costs far less than the latency of multiple queries. Moreover, the main gsn webapp page requires all types of data at the initialization, thus merging them makes sense.

Every type of data is mapped into the *field* block following the rules below. Field names are converted into lowercase and fields are sorted by type in the same order than below.

To begin with, the *virtual-sensor* block is opened with the following included attributes:

- name: virtual sensor name (unique) ;
- last-modified: last-modified datetime in unix format ;
- description: virtual sensor description.

Here is a sample of the xml output from the servlet:

```
<gsn name="gsn" author="cb" email="cb@epfl.ch" description="none">
  <virtual-sensor name="vs" last-modified="1171105166765">
    <field name="temp" type="long" description="none">-5</field>
  </virtual-sensor>
</gsn>
```

Then, the dynamic fields with their data are mapped from the output-structure block of the xml config of the virtual sensor into a unique field block. Below is a xml config sample:

```
<gsn name="gsn" author="cb" email="cb@epfl.ch" description="none">
  <virtual-sensor name="vs" last-modified="1171105166765">
    <field name="temp" type="long" description="none">-5</field>
  </virtual-sensor>
</gsn>
```

It would be mapped into the following line in the xml output from the servlet:

```
<gsn name="gsn" author="cb" email="cb@epfl.ch" description="none">
  <virtual-sensor name="vs" last-modified="1171105166765">
    <field name="temp" type="long" description="none">-5</field>
  </virtual-sensor>
</gsn>
```

An additional dynamic field is added with the timestamp in unix format as in the following xml sample:

```
<gsn name="gsn" author="cb" email="cb@epfl.ch" description="none">
  <virtual-sensor name="vs" last-modified="1171105166765">
    <field name="temp" type="long" description="none">-5</field>
  </virtual-sensor>
</gsn>
```

Afterwards, the static predicate fields are mapped from the adressing block of the xml config of the virtual sensor into a unique field block as well if they exist. Below is a predicate xml config sample:

```
<gsn name="gsn" author="cb" email="cb@epfl.ch" description="none">
  <virtual-sensor name="vs" last-modified="1171105166765">
    <field name="temp" type="long" description="none">-5</field>
  </virtual-sensor>
</gsn>
```

It would be mapped into the following line in the xml output from the servlet:

```
<field name="latitude" category="predicate">37.4</field>
```

Additionally, the webinputs block from the xml config of the virtual sensor are mapped into a unique field block if they exist. They are not kept hierarchically by command anymore. Below is a xml config sample:

```
<web-input password="test">
  <command name="cmd1">
    <field name="control" type="*binary:2mb">Binary upload</field>
  </command>
</web-input>
```

It would be mapped into the following line in the xml output from the servlet:

```
<field command="cmd1" name="control" category="input"
type="*binary:2mb" description="Binary upload"/>
```

Finally, the *virtual-sensor* block is closed.

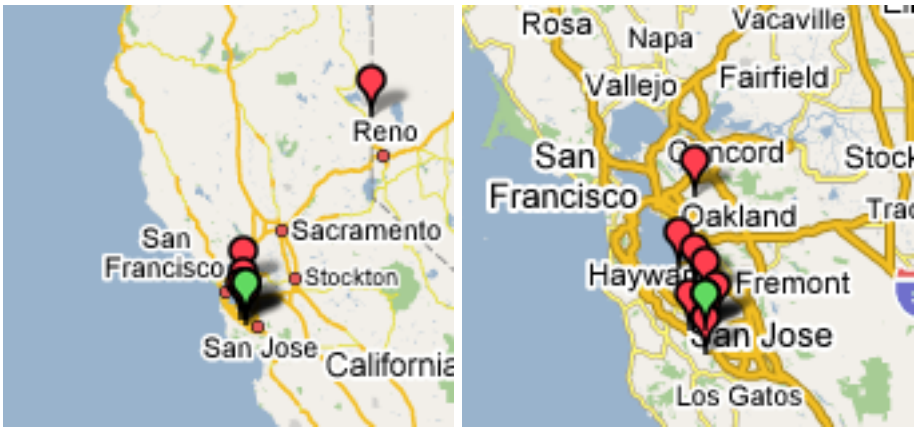
```
</virtual-sensor>
```

Mapping all these different types into the same field block syntax is easier to parse by jQuery and allow reuse of code for similar functionality, like between dynamic and static fields.

6.2.3.5 Highlighted marker on Google map

The code around GSN.map.trickhighlighted() can look quite suspicious. Why so much code and diverse variables just to highlight a marker on Google map? Changing from one type of marker to another does not require all this. It could have been made easier, but then when you zoom out you might not see the highlighted marker if it is surrounded by many other markers. It is due because Google map does not allow to specify the z-index of a marker and automatically order them by their latitude. Thus the marker with the lowest latitude will be shown on top of the others one.

This behaviour can be changed by using a trick which is to use a marker with an offsetted latitude and an offsetted marker image which will always be at a lower latitude than the other markers. Thus always be on top of them. It is offsetted depending of the zoom level and the normal marker height. With this trick, the highlighted marker is popping out of crowd as shown on the left side image.



The only drawback of this technique is that it might look weird sometime because of the shape of the marker. As the right side image shows, the spike of the highlighted market is on top of the one bellow. Anyway it isn't that frequent and the benefits when zoomed out in a dense area of marker are worth it.

6.2.3.6 Bookmarking and back button support for Ajax website

When a web application uses ajax techniques, one of the key features of the browsing experience is broken: the URL. With ajax calls made without a page load, the state of the current web page is not related to the URL anymore. It breaks the back button feature of any browser, and even worse, the current web page cannot be bookmarked properly. As those feature are essential to a website, it is necessary to find a way to mimic them. The solution is to use the hash part of the URL : `http://server.com/index.html#hash`. The hash part is easily modified through javascript without a page load and is considered as a page change by every browser which means that the back button feature works. In addition, an URL with a hash part can be bookmarked and, with some javascript initialization, can restore the webpage in the same state than when it was bookmarked. This was implemented using a jQuery extension as some tricky javascript code is required to make this functionality compatible with Internet Explorer.

For example, a URL in the gsn webapp when browsing the map part might look like the following:
`http://localhost:22001/#map,lt=38.17343267903539,lo=-121.9537353515625,z=8,vs=gpsvs:gps2:memorymonitorvs`

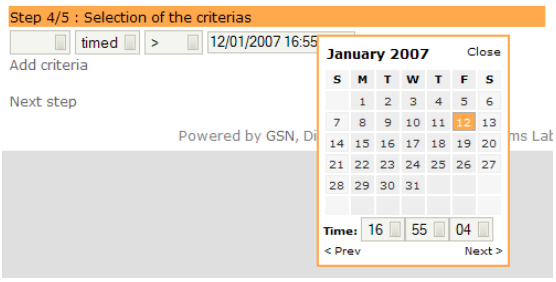
Such a URL is fully bookmarkable and will initialize the webapp at the same state which allows people to share URL in order to show specific sensors. It is decomposed into several parameters which represent:

- `map`: which context/part of the webapp (could be home, data or map)
- `lt=38.17343267903539`: latitude of the map when not in automatic mode
- `lo=-121.9537353515625`: longitude of the map when not in automatic mode
- `z=8`: zoom level of the map when not in automatic mode
- `vs=gpsvs:gps2:memorymonitorvs`: shown virtual sensors when some are hidden

Nevertheless, the URL can become quite long with all these parameters inside it. However, it is not a problem as Internet Explorer supports URL lengths up to 2048 characters. The alternative web browsers (FireFox, Opera & Safari) are even better and support up to 100000 characters.

6.2.3.7 Improved UI

A lot of work went into the design of the general user interface. The UI of the data part was highly customized. To simplify and to be more error prone, the datetime field was given a special helper shown bellow.

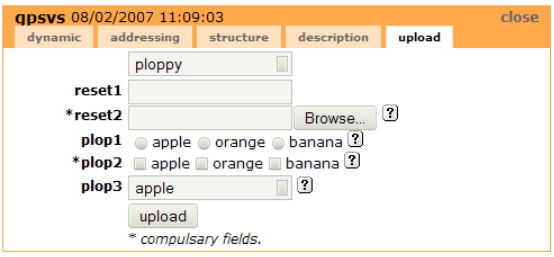


It allows the user to specify a date and time only with mouse click. It was proven to be much more quicker that way.

A waiting indicator was added during ajax calls when gsn refreshes sensors fields or when data is retrieved in the data part. Such indicators are necessary in ajax applications, otherwise the user believes that nothing is happening, gets frustrated and might click multiple times, which won't improve respond time.

6.2.3.8 Webinput upload to virtual sensor

The ability to upload input to a virtual sensor, as shown below, was one of the last additions to the gsn webapp.



For simplicity and modularity, the upload inputs are regrouped into subgroups which can be selected with the upper select box. Upload inputs are optional and specified in the sensor xml configuration, thus the tab only appears when it is specified. The different types are specified but are not enforced on the client side as this could be easily bypassed. The validation is done on the server side as client input should never be trusted. The following types have special input fields :

- *binary*: file upload ;
- *radio*: radio button with limited choice ;
- *checkbox*: checkbox with limited choice ;
- *select*: drop down box with limited choice ;

The radio, checkbox and select types use the syntax like "radio:ele1|ele2" to specify the multiple choice. The binary field is limited by a maximum size specified in the type part with a syntax like "binary:Xmb". When the type begins with a star, it is compulsary. Here is a sample xml config which shows all the capabilities:

```
<gsn name="gsn" author="cb" email="cb@epfl.ch" description="none">
  <virtual-sensor name="vs" last-modified="1171105166765">
    <field name="temp" type="long" description="none">-5</field>
  </virtual-sensor>
</gsn>
```

An ajax like behaviour was required to give feedback to the user after the webinput upload. However, the XMLHTTPrequest object used to send ajax calls doesn't support file upload for security reasons. To bypass this limitation, a normal FORM element is used but uses a hidden inline frame as a target. When the user clicks the submit button, the upload starts in the hidden IFrame to the destination */upload*. When the upload ends or breaks, the servlet replies back with html code like the following code line. This javascript code calls back the parent window GSN core object which allows the webapp to give feedback to the user.

```
<script>window.parent.GSN.msgcallback('msg',code);</script>
```

On the servlet side, the jakarta fileupload library was used to speed up development. A servlet translates the POST upload into XML-RPC format before sending it to the gsn system. The xml produced is similar to the following block lines. Normal fields are given in text format and binary fields are encoded in Base 64.

```
<input>
  <vsname>gpsvs</vsname>
  <command>test</command>
  <fields>
    <field>
      <name>reset1</name>
      <value>test comment</value>
    </field>
    <field>
      <name>reset2</name>
      <value>ABQOBJREFUevQ18FNW9/3+C/q7tza+tXB ... AARK5CYII=</value>
    </field>
    <field>
      <name>test1</name>
      <value>banana</value>
    </field>
    <field>
      <name>test2</name>
      <value>orange</value>
    </field>
    <field>
      <name>test2</name>
      <value>apple</value>
    </field>
    <field>
```

```
        <name>test3</name>
        <value>orange</value>
      </field>
    </fields>
  </input>
```

The servlet takes care of keeping only the input fields of the command activated by the drop down menu. The xml ouput can be read in the *sb.toString()* variable and has to be validated by the gsn system. The feedback is sent to the user as explained before and corresponds, in the servlet, to the *msg* and *code* variables. The *msg* could be any text string but should escape single quote and will be displayed at the top of the webinterface. A *code* higher than 200 is considered as an error and is shown in red to the user, otherwise it is in black. Nevertheless the code is not shown to the user in the current interface.

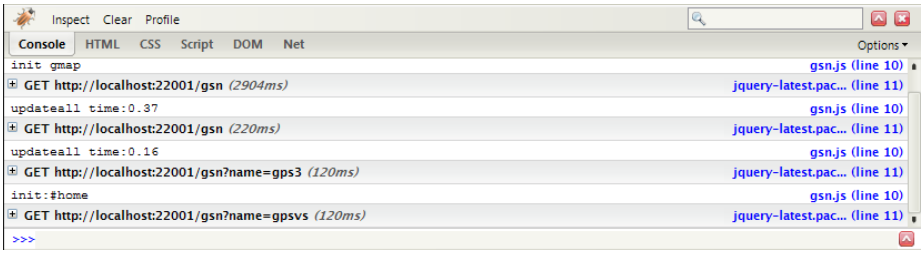
6.2.4 Testing & Debugging

GSN webapp was fully tested using the main browsers on the web: Internet Explorer, Firefox & Opera. There was not any formal testing routine or unit testing. However, the core functionality were tested in the three browsers to ensure that they were working similarly.

After a month of implementation, the interface was quite advanced but never tested with many sensors. When we added more than 40 sensors, it became ridiculously slow. It took more than 15 seconds to show them. JavaScript is slow and using some nice syntax features of jQuery can be very expensive, especially the $("") function ! By refactoring the code, avoiding to use the most expensive functions and extensive testing, the execution time was reduced to less than two seconds, thus an improvement of a factor ten. However, as the number of virtual sensors could be even much bigger, it was decided to show only the first ten sensors on the front page to bound the loading time. Later, the speed was again improved by updating jQuery to the new 1.1 version.

6.2.4.1 Javascript debugging

With ajax webapp, the amount of complex javascript code is significant, thus it needs to be debugged and analyzed. Hopefully, a relatively new extension for FireFox called FireBug[?] helps a lot.



As shown above, it allows to easily output debugging messages with the *console.debug(...)* object. Objects can be fully analysed and changed. The html and css code could be modified on the fly. This is very powerful and helps a lot.

Nevertheless, when testing gsn compatibility with IE, this is a pain to not have FireBug. Moreover that IE usually have some weird and hard to catch JavaScript bug.

6.2.5 GPSVS sensor

A GPS bluetooth sensor was received during the project development. A small virtual sensor was created to test it. Using the serial port wrapper, it was very fast and easy to do. This experience also brought some relevant feedback to the team. The sensor code was quickly put together using a serial port wrapper, as bluetooth supports serial port emulation (BlueTooth Serial Port Profile). GPS sensors actually have some specification for their output called NMEA[**?**]. Only a subset of it was implemented, the GPRMC part to be precise. This is some formatted text only output looking for example like *"$GPRMC,081836,A,3751.65,S,14507.36,E,000.0,360.0,130998,011.3,E*62"* which has to be converted to Google maps coordinates.

The GPS sensor was developed using a GPSlim device but should be successfully used by any device following the NMEA specification. It was actually tested with a Nokia GPS device without any modification. The only small problem was the linkage of the Nokia GPS device which requires a passcode. It can be found deep down in the documentation and is 0000 by default.

Chapter 7

Writing a Protocol Description

In many cases, your data sources will emit structured data that respect a predefined format. Such a format is called a protocol. It is of course possible to do the data extraction from the received data packets with your own code. However, when these protocols are complex, the task is not that easy and it can consume a lot of time.

This is why GSN ships with the jnetstream packet decoding library. To use it with your protocol, you only need to describe your packets structure in a C-like syntax. You will be able to update the protocol definition without recompiling the parser. For more information, please consult `http://jnetstream.sourceforge.net`.

It is also possible to build such packets using jnetstream. This feature is currently in development. This allows to communicate with your data source.

Index