Internal Report 2010–05 June 2010

Universiteit Leiden
Opleiding Informatica

Data mining of sensory data streams
in
complex systems

Gratian Schiopu

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1 Abstract

In data mining of data streams produced by complex systems, equipped with many sensors we will look at some of the main steps of how to deal with such massive flow of data. In trying to overcome some of the challenges when dealing with data streams, we will look at a real-time system, a bridge called De Hollandse Brug, which at the moment produces massive quantities of data, waiting to be analyzed.

Some of the steps presented in this thesis will cover the methods of how to filter the produced data, how to describe and to transport the data to a specific location, to analyze the specific features of the data and the relation between the different type of sensors. We will also propose a model based on the hidden Markov models to distinguish between the different type of states of the bridge, based on the emmited events detected.

By specifically looking at data streams, we try to gain insight in new data mining methods when dealing with massive and rapidly changing data.

2 Introduction

The number of complex systems equipped with many various types of sensors is growing at a rapid pace in todays modern society. Such systems can vary in complexity, the number and the type of sensors depending on their scope and domain of utilization.

As an example of such a system we can think of a complex medical monitoring systems where the sensors monitor the patients condition after a surgical procedure in an intensive care room. In such systems the sensors attached can measure blood pressure, the cardiac rhythm by ECG, body temperature, etc. Another example are geological systems that use their sensors scattered over a wide surface area to monitor earthquakes or the condition of an erupting volcano. Other systems could monitor the condition of civil engineer constructions like bridges exposed to daily traffic but also to climate and seasonal changes.

From a computer science perspective the conclusion is that all these systems produce large quantities of data per time unit that need to be processed and analyzed. The data can be seen as a continuous stream being produced very fast and in large quantities to be processed by data mining algorithms. The dynamic nature of this data stream production requires a different approach concerning processing, transportation, storage, visualization of the data in comparison to more static data types.

In this thesis we look at such a real world complex system, a bridge, equipped with many different sensors which produces daily considerable amounts of data. We are interested to find practical ways in processing, transporting and storing the data. From analyzing such a real world complex system we aim not only to find practical solutions and build tools for the civil engineers monitoring the bridge but also to gain a greater understanding when dealing with mining of data streams and the algorithms and techniques.

3 De Hollandse Brug

The bridge "De Hollandse Brug", with its 365 m length, is a large bridge in The Netherlands, a real-world complex system comprising 145 different sensors each of them producing large amounts of data around the clock, we use it as a relevant case study.

We first briefly describe the sensors on the bridge.
The monitoring network of the bridge consists of 145 different sensors, each identified by s_i for a number i:

- strain-gauges embedded in the concrete and attached to the outside, measuring the stress of the bridge in two different (X-Y) coordinates. There are 91 strain-gauge sensors in ranges: [s100 . . . s120], [s124 . . . s145], [s148 . . . s168], [s172 . . . s198]. Total number of stain-gauges is 21 + 22 + 21 + 27 = 91

  The sensors [s124-s125, s130-s131,s140-s141, s148-s149, s158-s159, s164-s165, s191, s193, s195, s197] are attached in the direction of the X-axis. The sensors [s100-s102, s109-s111,s116-s118, s126, s132, s135-s137, s142, s150, s153-s155, s160, s166, s172-s174, s185-s190, s192, s194, s196, s198] are embedded in the concrete in the direction of the X-axis. Total 16 + 34 = 50.

  The sensors [s104, s106, s108, s113, s115, s129, s145, s163, s176, s178, s180, s182, s184] are attached in the direction of the Y-axis. The sensors [s103, s105, s107, s112, s114, s119-s120, s127-s128, s133-s134, s138-s139, s143-s144, s151-s152, s156-s157, s161-s162, s167-s168, s175, s177, s179, s181, s183] are embedded in the concrete in the direction of the Y-axis. Total 13 + 28 = 41.

- 'geo-phones' (vibration sensors) that measure the vertical movement (Z-axis) of the bottom of the road-deck as well as the supporting columns. The sensors [s200 . . . s209], [s216 . . . s224], [s232 . . . s246] are the 34 geo-phones. Total 10 + 9 + 15 = 34.

- thermometers embedded in the concrete and attached to the outside. These are the 16 sensors numbered: [s300 . . . s315].

Furthermore, there is a weather station and a video-camera that provides a continuous video stream of the actual traffic on the bridge.

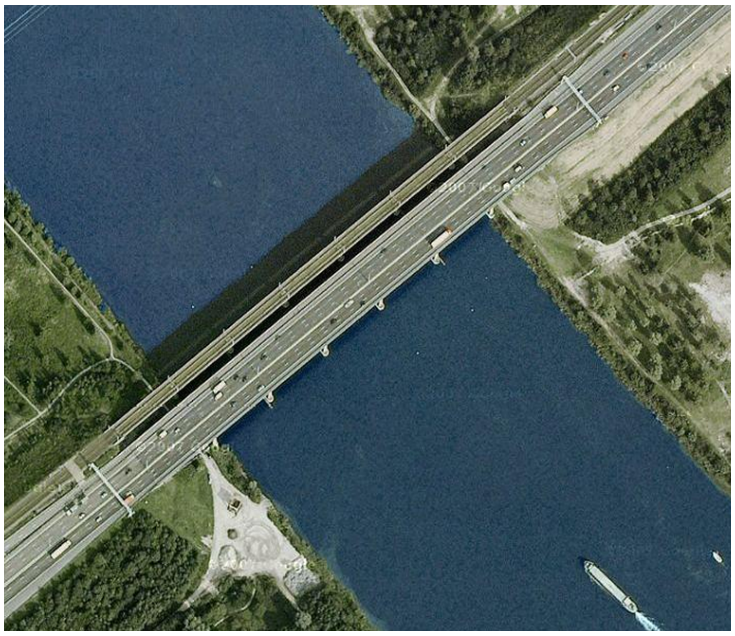


Figure 1: Picture of the bridge ”De Hollandse Brug”, which connects the province Flevoland to the province North-Holland in The Netherlands.

4 Definitions and notations

In this section we provide some definitions that are used throughout the thesis:

Time series: is defined as a series of discrete values $\langle v_1, v_2 \ldots, v_n \rangle$ produced on times $[t_1, t_2 \ldots, t_n]$, where $t_n \in \mathbf{N}$, $v_n \in \mathbf{R}$, with: $t_1 \mapsto v_1, t_2 \mapsto v_2, \cdots t_n \mapsto v_n$.

A shorter notation we will also use is as a range of discrete values $Y = \langle y_1, y_2, \ldots y_i \rangle$ produced on times $[t_1, t_2, \ldots, t_i]$ where $t_i \in \mathbf{N}$ and $y_i \in \mathbf{R}$. The used indices in the discrete values will implicitly denote that they are produced at the corresponding time.

Threshold (θ): given a time series $Y = \langle y_1, y_2, \ldots y_i \rangle$ we define a constant value called threshold that we will use to compare the given values with.
To start with, we will compute this threshold value as the average over a set of given discrete values: $Y = \langle y_1, y_2, \ldots y_i \rangle$

$$\theta = \frac{\sum_{i=1}^{n} y_i}{n} \tag{4.1}$$

For monitoring our system, the bridge, we are interested in events happening on the bridge. Under influence of the external events on the bridge, the sensors will produce data that will measure the intensity of these events accordingly. The informal idea of a measured event on the bridge is as data being produced by a sensor that will exceed a given threshold for a certain period of time.

An event can be captured by different type of sensors. By looking at the data measured by the **strain-gauges**, we will give some definitions with respect to the events outputted by these sensors.

Event (ev): given the a series $Y = \langle y_1, y_2, \ldots y_i, \ldots y_j, \ldots \rangle$, with: $t_1 \mapsto y_1, t_2 \mapsto y_2, \cdots, t_i \mapsto y_i, \cdots, t_j \mapsto y_j$. The subset $Y' \subset Y, Y' = \langle y_i, \ldots y_j \rangle$ will mark an event if all the values in the subset Y' are above the given threshold $\theta : y_i \geq \theta, \ldots, y_j \geq \theta$.

We define the *start* of the event at t_i where the first value $y_i \geq \theta$ and the *end* of the event at t_j where the last value $y_j \geq \theta$

- t_i = start-event(ev)
- t_j = end-event(ev)

Event length (event-length(ev)): given the time series $Y = \langle y_1, y_2, \ldots y_i, \ldots y_j, \ldots \rangle$. The event length is the distance in time units for the period that the event is occuring. If for a given event ev contained in the subset $ev = \langle y_i, \ldots y_j \rangle$ by the condition that all its values in the subset are above the given threshold $\theta : y_i \geq \theta, \ldots y_j \geq \theta$ with: $t_i \mapsto y_i, \cdots, t_j \mapsto y_j$.

If t_i marks the start of the event ev and t_j marks the end of ev, the length of the event is given by: event-length$(ev) = (t_j - t_i)$.

Event-gap (event-gap(ev_1, ev_2)): given the time series $Y = \langle y_1, y_2, \ldots y_i, \ldots y_j, \ldots, y_p, \ldots y_q, \ldots \rangle$ containing two events $ev_1 = \langle y_i, \ldots y_j \rangle$ and event $ev_2 = \langle y_p, \ldots y_q \rangle$:

$ev_1 = Y' \subset Y, Y' = \langle y_i, \ldots y_j \rangle$, with $t_i \mapsto y_i, \cdots, t_j \mapsto y_j$ and
$ev_2 = Y'' \subset Y, Y'' = \langle y_p, \ldots y_q \rangle$, with $t_p \mapsto y_p, \cdots, t_q \mapsto y_q$,
where $t_i < t_j < t_p < t_q$, we define an event-gap: $(t_j, \cdots, t_p)$ the set $Y''' \subset Y, Y''' = \langle y_j, \ldots y_p \rangle$, where the values $y_j < \theta, \ldots, y_p < \theta$.

- $t_i =$ start-event(ev_1)
- $t_j =$ end-event(ev_1)
- $t_p =$ start-event(ev_2)
- $t_q =$ end-event(ev_2)
- event-gap$(ev_1, ev_2) = (t_p - t_j)$, where the the values $y_j < \theta, \ldots, y_p < \theta$

An *event-gap* is the time distance between the two events: $(t_p - t_j)$ where the values $y_j < \theta, \ldots, y_p < \theta$.

Minimal gap: given the time series $Y = \langle y_1, y_2, \ldots y_i, \ldots y_j, \ldots, y_k, \ldots y_l, \ldots \rangle$ with two events $ev_1 = \langle y_i, \ldots y_j \rangle$ and $ev_2 = \langle y_k, \ldots y_l \rangle$, we define the minimal gap, the distance between the two events $(t_k - t_j) < \mu$, where μ is a given constant. In case of a minimal gap, where $(t_k - t_j) < \mu$ we consider $ev_1 \bigcup ev_2 = ev_\cup$ with:

- $t_i =$ start-event$(ev_\cup)$
- $t_l =$ end-end$(ev_\cup)$

If the gap is not minimal we consider the two events as being distinct.
Cumulative gaps: given the time series with three events:
$Y = \langle y_1, y_2, \ldots y_i, \ldots y_j, \ldots, y_k, \ldots y_l, \ldots y_p, \ldots y_q, \ldots \rangle$ and two minimal event gaps:

- $(t_k - t_j) < \mu$, between $ev_1 = \langle y_i, \ldots y_j \rangle$ and $ev_2 = \langle y_k, \ldots y_l \rangle$
- $(t_p - t_l) < \mu$, between $ev_2 = \langle y_k, \ldots y_l \rangle$ and $ev_3 = \langle y_p, \ldots y_q \rangle$

The *cumulative gap* is the distance $(t_k - t_j) + (t_p - t_l)$. If the distance $(t_k - t_j) + (t_p - t_l) < \mu$ we consider the three events as being one event: $ev_1 \bigcup ev_2 \bigcup ev_3 = ev_\cup$ with:

- $t_i =$ start-event$(ev_\cup)$
- $t_q =$ end-event$(ev_\cup)$

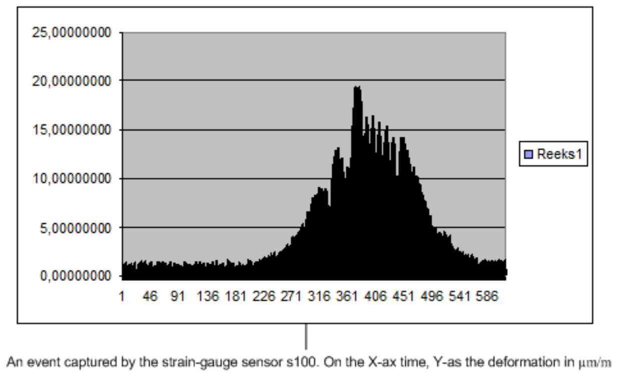


Figure 2: Sensor data produced by the strain-gauge sensor s100 represents a dynamic event on the bridge.

Dynamic event (DE): is a short period of time event (lasting from a few seconds to maybe ten seconds) with a very well defined global maximum above the threshold. In figure 2 we see a dynamic event produced by a heavy duty truck driving over the bridge. On the given time scale, for a threshold $\theta = 2.5$ the event starts approximately at $t_1 = 2.3$ sec and ends at $t_2 = 5.4$ sec.

Static event (SE): is a gradually growing event over a longer period of time (from a few minutes to maybe hours). In these sort of events, the forming of a plateau of values oscillating around a high global maximum can be seen. See such an event in figure 3.

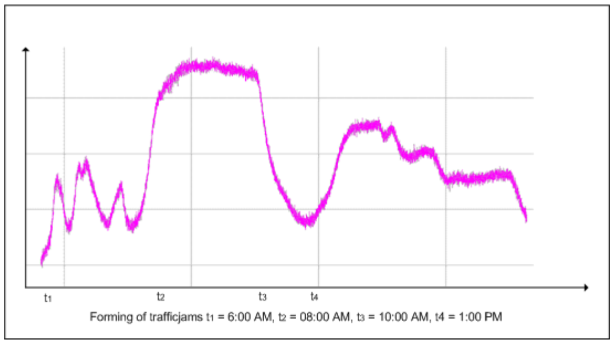


Figure 3: Sensor data produced by the strain-gauge sensor represents a static event.

Let us define when f, the function representing such an event either dynamic (DE) or static (SE). For an event the surface under the function f can be expressed by the integral I_{DE} and for a static event by the integral I_{SE} for a given θ :

$$\mathfrak{I}_{DE} = \int_{t_1}^{t_2} f_1(x) \cdot dx \tag{4.2}$$

$$\mathfrak{I}_{SE} = \int_{t_3}^{t_4} f_2(x) \cdot dx \tag{4.3}$$

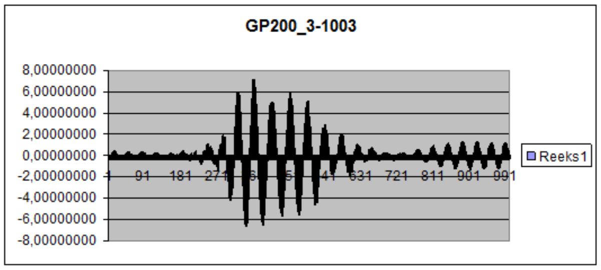
The difference between a dynamic event and a static event using the same θ is that:

$$\Im_{DE} << \Im_{SE} \tag{4.4}$$

not only because the $\Delta_1 = (t_2 - t_1) << (t_4 - t_3) = \Delta_2$ but also because $max_1 < max_2$ where $max_1 \in \mathbf{R}$ is the global maximum on the interval $[t_1, \ldots, t_2]$ and $max_2 \in \mathbf{R}$ is the global maximum on the interval $[t_3, \ldots, t_4]$.

The simplest way to approximate the previous integrals is by multiplying the found max_i with the Δ_i. By comparing these approximations among them we can categorize the events. The civil engineers are specially interested in dynamic events. In building a real-time bridge monitoring system we could use such *rules* to filter the interesting events from the banal events.

The data collected by the geo-phones representing a dynamic event can be seen in Figure 4



Geo-phone sensor s200 captures an event.

Figure 4: The sensor data produced by the geo-phone s200 represents the same dynamic event as seen in the figure 2.

Looking at the data produced by a strain-gauge sensor as seen in figure 2, we define with $f_{SG}(x)$ the function which represents such a dynamic event, captured by the strain-gauge sensor. If we compare this function with the function $f_{GEO}(x)$ produced by a geo-phone sensor in the close vicinity of the strain-gauge sensor and representing the same dynamic event seen in figure 4, we find a relation between the two functions given by Koopman [4]:

$$f'_{_{DE}}(x) = \frac{\partial}{\partial t} f_{_{SG}}(x) \tag{4.5}$$

$$f'_{_{DE}}(x) = c_1 \cdot f_{_{GEO}}(x) + c_2 \tag{4.6}$$

$$f_{GEO}(x) = c_3 \cdot \frac{\partial}{\partial t} f_{SG}(x) + c_4 \tag{4.7}$$

where $c_1, c_2, c_3, c_4 \in \mathbf{R}$.

The relation between the strain-gauge function and the first derivative which represents the data measurements of the geo-phone gives us the opportunity to find the local maxima and minima in the function $f(x)$. Each time that the $f'(x)$ will change sign, it means that on the interval where it occurs we have a local minimum or maximum.

5 Data description

5.1 Introduction

Each different type of sensor will produce a series of values over time, measuring the events happening on the bridge. The strain-gauges will measure the physical deformations of the bridge caused by the passing vehicles. The deformations will be measured in micro-strain units in the X and Y coordinates:

- X = longitudinal, in the driving direction by the traffic on the bridge (strain-gauges).
- Y = transversal, in the (XY)-plane but perpendicular on the driving direction X (strain-gauges).
- Z = vertical, perpendicular on the (XY)-plane (geo-phones). The geo-phones will measure the vibrations caused by the forces produced during the motion of the vehicles on the bridge.

The forces exerted on the bridge will cause physical deformations of the construction and vibrations. The majority of these forces are caused by the traffic on the bridge. There could be also other events, like hazardous factors causing unusual vibrations or deformations of the bridge: earth quakes, possible collisions by a vessel with the bridge, etc.

Based on the system requirements document defined by Struktons Civil Engineers in their document [1], the Bridge Monitoring System can be described as is shown in Figure 5. Next we give a global description of the bridge monitoring system. The three main components are:

- WIM = weigh-in-motion (traffic monitoring system on the bridge).
- SIM = structural integrity monitoring.
- Environmental Data Analysis.

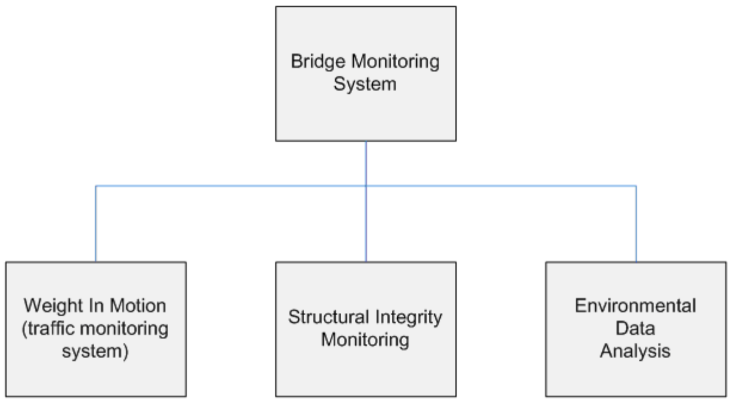

Bridge Monitoring System
Weight In Motion (traffic monitoring system)
Structural Integrity Monitoring
Environmental Data Analysis

Figure 5: The Bridge Monitoring System

The first subsystem, called WIM (weigh-in-motion) is a traffic monitoring system. This subsystem will keep track of the traffic using the bridge, which will generate events registered by the sensors placed in the deck of the bridge. After registering and storing such major events, this subsystem will carry an in-depth analysis about such generated events. Monitoring the frequency, load, speed and a generic typing of the vehicles generating these events, can be viewed as its most important tasks.

The second subsystem, SIM (structural integrity monitoring) will monitor the structural integrity of the construction as a whole. This subsystem will store and analyze information concerning the deformations of the construction caused by the daily and seasonal fluctuations in temperatures, speed response to major events by the construction analyzed over longer periods of time, etc. Following the expertise of civil engineers in this subsystem we could add new monitoring goals correlated to various environmental parameters.

The third subsystem, EDA (environmental data analysis) will store some of the most correlated environmental sensory data with the monitored construction. Air temperature, humidity, precipitations, wind speed and direction, solar energy, etc. are some of the parameters which will continuously be stored at certain hours of the day.

In this thesis, because of the complexity of the monitoring system to be build following the specifications of the civil engineers, we will mainly focus on WIM and SIM subsystems of the Bridge monitoring system.

5.2 Weigh in motion subsystem (WIM)

The actual WIM system (version 1.0) is functional and collecting the data from its sensors. The data produced by the sensors is stored in flat files which will be described in the next section. The assumption is that the data is available and stored in files on the local server. As a fact that the data transmission is unavailable at this moment, the present WIM system cannot yet be used as a real-time monitoring tool which was defined as one of the primary intentions of its builders.

As seen in the Figure 6, the first step in building a real-time WIM system (version 2.0) consists of the data reduction and data transmission step. Here the bridge monitoring system will determine the start threshold. Based on this threshold the main events are detected and filtered from the rest of the data. Using a Client/Server architecture, the detected events are being transmitted between the server where the data is collected from the sensors and a portal where the data is gathered for further transmission to a GRID for data analysis.

As stated by Witten in [2], data preparation accounts for 60% of the effort when we consider data mining applications. Filtering the data can be regarded as a very important step in WIM system. The decision of drawing a line between which data is important from what can be considered as unimportant is not only decisive for the amount of data we will transmit but also for the quality of the data. On one side by using a high threshold will discard some useful on the other side by using a low threshold we will add too much noise to the data.

In appendices A through G we present the source code of a filter program that can be used on the local server. The present limitations of the program are that it looks at only the data produced by two sensors in the stored flat files and not all the sensors. Also it does not run autonomously, unsupervised the filtering of all the produced files on the local server. Currently, it is run manually by deciding which files to use as an input.

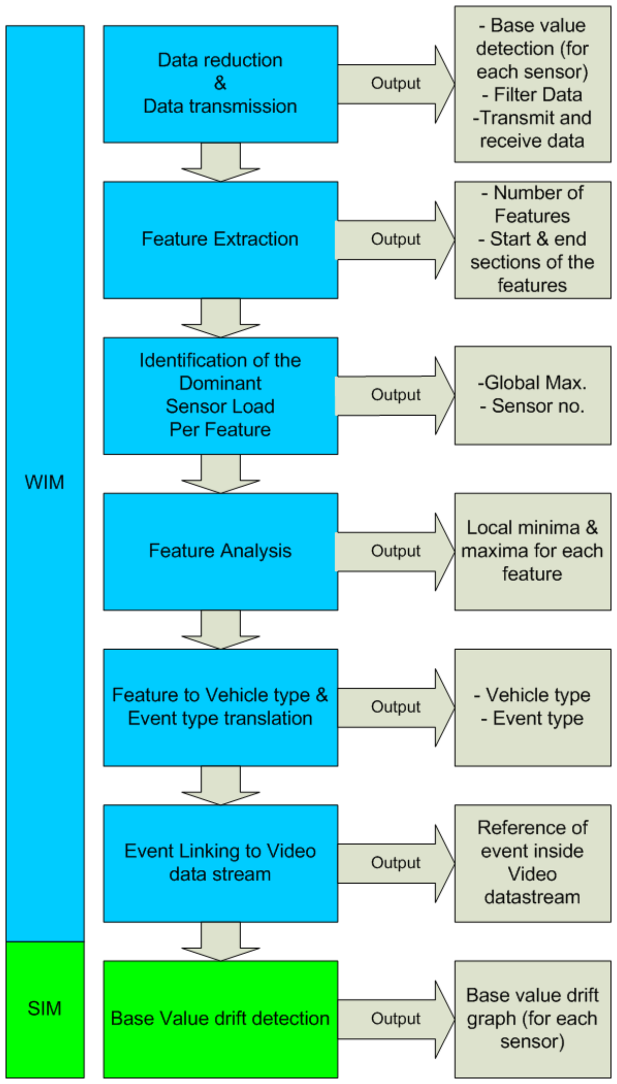

WIM
SIM
Data reduction & Data transmission
Output
- Base value detection (for each sensor)
- Filter Data
-Transmit and receive data
Feature Extraction
Output
- Number of Features
- Start & end sections of the features
Identification of the Dominant Sensor Load Per Feature
Output
-Global Max.
- Sensor no.
Feature Analysis
Output
Local minima & maxima for each feature
Feature to Vehicle type & Event type translation
Output
- Vehicle type
- Event type
Event Linking to Video data stream
Output
Reference of event inside Video datastream
Base Value drift detection
Output
Base value drift graph (for each sensor)

Figure 6: The distinct modules used by WIM and SIM in data analysis produced by the sensory network.

5.3 The flat file description

In the next table we can see how the data that is being produced by the sensors is stored in a flat file:

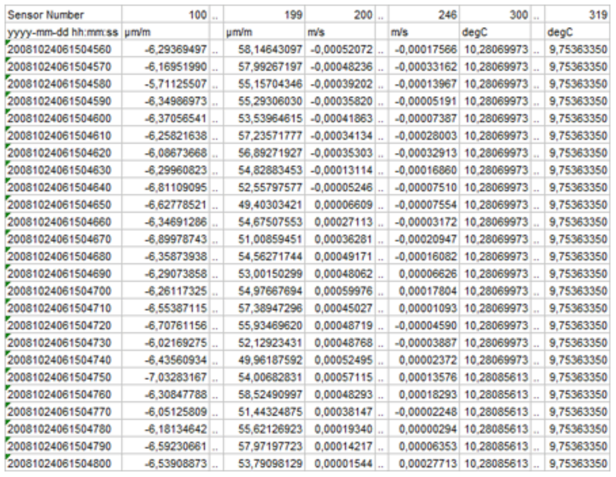
SensorNo	100	..	198	200	..	246	300	..	319
yyyymmddhh:mm:ss	μm/m	..	μm/m	ms	..	ms	degC	..	degC
20081024061504560	-6,29369497	..	8,14643097	-0,00052072	..	-0,00017566	10,28	..	9,75
20081024061504570	-6,16951990	..	7,99267197	-0,00048236	..	-0,00033162	10,28	..	9,75
20081024061504580	-5,71125507	..	5,15704346	-0,00039202	..	-0,00013967	10,28	..	9,75
.									
.									
20081024061504800	-6,53908873	..	5,79098129	0,00001544	..	0,00027713	10,28	..	9,75

The sensors numbered from 100 to 199 are strain-gauges, the sensors 200 to 246 are geophones and the sensors numbered 300 to 319 are temperature sensors.

Each 5 minutes, a 50Mb flat data file, filled with sensory data will be produced on the local server. On a 24hour basis we will have 288 of such data files. The files will have a unique name based on the $< yyyymmdd.xxx >$ notation. In this notation the $yyyy$ is the year, mm is the month, dd is the day when the file was created and xxx is the counter of the file: $count \in [000, \dots, 287]$, based on the 5minutes time scale; this represents the unique production in time of a flat data file.

We can deduct the time sequence inside a data file without opening the file, from the extension xxx counter.
The hour h in the samples is given by: $h = xxx\ DIV\ 12$.
The minute range t_{range} is given by: $t_{range} = xxx\ MOD\ 12$:

$$t_{range} = \begin{cases} 0 & \text{minutes} \in [\ 0, \dots,\ \ 4], \\ 1 & \text{minutes} \in [\ 5, \dots,\ \ 9], \\ 2 & \text{minutes} \in [10, \dots, 14], \\ 3 & \text{minutes} \in [15, \dots, 19], \\ \dots & \\ 11 & \text{minutes} \in [55, \dots, 59]. \end{cases} \tag{5.1}$$



Sensor Number	100	..	199	200	..	246	300	..	319
yyyy-mm-dd hh:mm:ss	µm/m		µm/m	m/s		m/s	degC		degC
20081024061504560	-6,29369497	..	58,14643097	-0,00052072	..	-0,00017566	10,28069973	..	9,75363350
20081024061504570	-6,16951990	..	57,99267197	-0,00048236	..	-0,00033162	10,28069973	..	9,75363350
20081024061504580	-5,71125507	..	55,15704346	-0,00039202	..	-0,00013967	10,28069973	..	9,75363350
20081024061504590	-6,34986973	..	55,29306030	-0,00035820	..	-0,00005191	10,28069973	..	9,75363350
20081024061504600	-6,37056541	..	53,53964615	-0,00041863	..	-0,00007387	10,28069973	..	9,75363350
20081024061504610	-6,25821638	..	57,23571777	-0,00034134	..	-0,00028003	10,28069973	..	9,75363350
20081024061504620	-6,08673668	..	56,89271927	-0,00035303	..	-0,00032913	10,28069973	..	9,75363350
20081024061504630	-6,29960823	..	54,82883453	-0,00013114	..	-0,00016860	10,28069973	..	9,75363350
20081024061504640	-6,81109095	..	52,55797577	-0,00005246	..	-0,00007510	10,28069973	..	9,75363350
20081024061504650	-6,62778521	..	49,40303421	0,00006609	..	-0,00007554	10,28069973	..	9,75363350
20081024061504660	-6,34691286	..	54,67507553	0,00027113	..	-0,00003172	10,28069973	..	9,75363350
20081024061504670	-6,89978743	..	51,00859451	0,00036281	..	-0,00020947	10,28069973	..	9,75363350
20081024061504680	-6,35873938	..	54,56271744	0,00049171	..	-0,00016082	10,28069973	..	9,75363350
20081024061504690	-6,29073858	..	53,00150299	0,00048062	..	0,00006626	10,28069973	..	9,75363350
20081024061504700	-6,26117325	..	54,97667694	0,00059976	..	0,00017804	10,28069973	..	9,75363350
20081024061504710	-6,55387115	..	57,38947296	0,00045027	..	0,00001093	10,28069973	..	9,75363350
20081024061504720	-6,70761156	..	55,93469620	0,00048719	..	-0,00004590	10,28069973	..	9,75363350
20081024061504730	-6,02169275	..	52,12923431	0,00048768	..	-0,00003887	10,28069973	..	9,75363350
20081024061504740	-6,43560934	..	49,96187592	0,00052495	..	0,00002372	10,28069973	..	9,75363350
20081024061504750	-7,03283167	..	54,00682831	0,00057115	..	0,00013576	10,28085613	..	9,75363350
20081024061504760	-6,30847788	..	58,52490997	0,00048293	..	0,00018293	10,28085613	..	9,75363350
20081024061504770	-6,05125809	..	51,44324875	0,00038147	..	-0,00002248	10,28085613	..	9,75363350
20081024061504780	-6,18134642	..	55,62126923	0,00019340	..	0,00000294	10,28085613	..	9,75363350
20081024061504790	-6,59230661	..	57,97197723	0,00014217	..	0,00006353	10,28085613	..	9,75363350
20081024061504800	-6,53908873	..	53,79098129	0,00001544	..	0,00027713	10,28085613	..	9,75363350

Figure 7: The sensor data file: 20081024075 was produced October, the 24th 2008 at 6:15:00 AM and contains the sensor data time series between 6:15:00,000 - 6:19:59,590 AM.

5.4 Data filtering

In Figure 8 we present the block diagram of a filter program (version 2.0). The application can be used to filter the events from a given sensor data file.

The filter application is given as source code in appendices A through G. The first module calculates the threshold from a given sensor file. The value is stored in a threshold file in case that we want to save the thresholds for each calculated sensor file. In this version the application takes only into consideration the data produced by two sensors: the sensor s100 and s101. These sensors are strain-gauges and they are placed on the bridge in each other's vicinity, correlating highly with respect to their local minima and maxima. By using a low threshold $\theta < -10$ the program will not filter anything and produce one single event for the whole input file. In such case all the input sensory data will be used as output in the chosen file format.

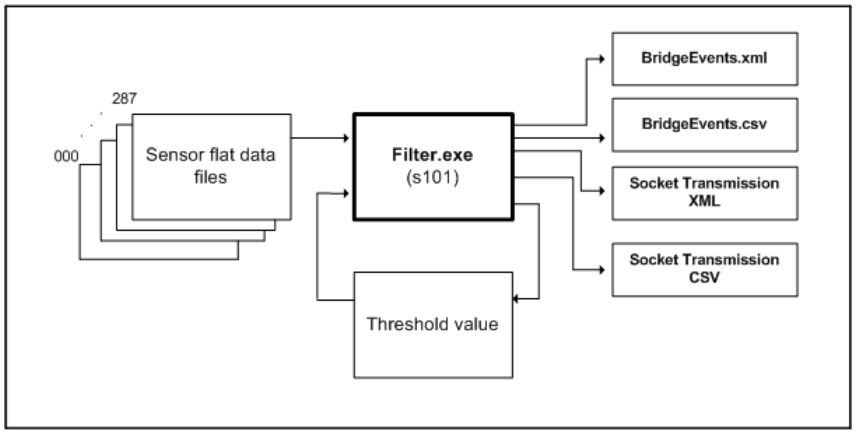


Figure 8: Data filtering program.

The first step of the filter program is to compute a start threshold. The threshold will be calculated from a real-time flat data file that will be given as input. A flat data file provided by Strukton civil engineers which contains a few dynamic events was used as input for the program to detect these events. To start with, the events have been visualized and identified outside the filter program in an Excel spreadsheet. Because the actual version of the program looks only at a limited number of sensors (the strain-gauge sensors S_{100} and S_{101}), the filter program calculates from the given data the threshold $\theta_{S_{100}}$ for the sensor S_{100} . We assume that this sensor is the "leading" sensor. All the given data for this sensor was averaged over the total number of samples.

Another more accurate method would be, to use the first threshold $\theta'_{S_{100}}$ and then calculate a second threshold value, $\theta''_{S_{100}}$ derived from the first one, after leaving out the events. This method is not yet implemented in the filter program.

In the next version of the filter program we intend to build the feature which will compute all the thresholds $\theta'_{S_{100}}, \ldots, \theta'_{S_{198}}$ for the strain-gauge sensors.

In the second phase the filtering process takes place. The filtering algorithm can be described in short:

```
var F_theta: theta_file;  // stored threshold from previous computations
var I : flatdatafile;         // flat data file used as input
var O : outputfile;         // output data file (XML or CSV)
var s : string;   // used for a line from the datafile
var c :  string;  // use for the substring representing sensor s100
var v : float;    // s100 float value
var prev :int;  // counter to keep track the end of an event
var eventId: integer;  // counter for the events we detect
var i: integer // counts the processed rows from the input file
var theta : float;   // threshold
var gap: integer;   // used to measure the gap between events
var StartSeq : boolean;  // signals the start of the first event in the whole sequence of data
var StartEvent : boolean;  // signals the start of an event


StartSeq = StartEvent = true;   // init. StartSeq and StartEvent
eventId = 0;                           // init. eventID
gap = 50;      // init. gap to 1/2 second

read (F_theta, theta);           // read theta from file

If I exists then                      // if input file exists
 OpenInputFile (I);                 // open input file
 white not eof (I) do               // while not the end of input file
   getline (I, s);                    // read one line from input file
   copyStr (s, c);       // copy string from file to a string variable
   Str2float (c, v);      // conver string sensor float var.

   if StartEvent then             // event marker is true
     eventId++ ;                   // increase the event counter
     StartEvent = false;         // set marker to false
   fi

   if (v >= theta) then   // if the float var. is above theta

     OpenOutputFile (O);    // open output file
     if O.open then          // if output file is open
       fout (v);                 // write sensor data to the output file
     fi
     prev++;                  // increase prev. by one
     if StartSeq then       // is this the first event?
       prev = i;               // yes:  set the counters: prev and i right
       StartSeq = false;  // set the StartSeq false
     else                          / No: not the first event
       if ((i - prev) > gap ) then  // is the cumulated gap  too big ?
         StartEvent = true;        // yes: new event marker set to true
         prev = i;                        // begin counter of the next event is set to i
       fi // (i - prev) > gap
     fi  //  StartSeq
   fi    // (v >= theta)
   i++ ; increase row counter
 od

 Close I;   // close Input file
 Close O; // close Output file
fi
```

The next module of the program uses sampling over a constant sequence of sensory data values. We will use a stack as ADS to average the values on the stack. Each time a read is performed from the input data file, we will first push those values on a stack. After the stack is full we will pop the values from the stack and compute their average. We can look at it as a repeated buffering, performing an average over the buffered values each time when the buffer is full. In this case we will not compare each time series values with the threshold but with only one computed average over a set of values. The stack is implemented as a single linked list and it can hold 10 elements. Because the initial sensory data is being sampled with the frequency of $100Hz$, by using a stack with 10 elements and averaging over those values to one value, we will reduce the initial sampling rate to $10Hz$. This means that each $\frac{1}{10}$ of a second we will output one sensor value.

```
var F_theta: theta_file;  // stored threshold from previous computations
var I : flatdatafile;         // flat data file used as input
var O : outputfile;         // output data file (XML or CSV)
var s : string;   // used for a line from the datafile
var c :  string;  // use for the substring representing sensor s100
var v : float;    // s100 float value
var prev :int;  // counter to keep track the end of an event
var eventId: integer;  // counter for the events we detect
var i: integer // counts the processed rows from the input file
var theta : float;   // threshold
var gap: integer;   // used to measure the gap between events
var StartSeq : boolean;  // signals the start of the first event in the whole sequence of data
var StartEvent : boolean;  // signals the start of an event
var avegeStack: float;    // keeps the avg value of the stack

var S: Stack;  // a stack ADS of 10 elements


StartSeq = StartEvent = true;   // init. StartSeq and StartEvent
eventId = 0;                           // init. eventID
gap = 50;      // init. gap to 1/2 second

read (F_theta, theta);          // read theta from file

If I exists then                     // if input file exists
 OpenInputFile (I);                // open input file
 white not eof (I) do              // while not the end of input file
   getline (I, s);                   // read one line from input file
   copyStr (s, c);       // copy string from file to a string variable
   Str2float (c, v);      // convert string sensor float var.

   if StartEvent then             // event marker is true
     eventId++ ;                   // increase the event counter
     StartEvent = false;         // set marker to false
   fi

   if (!S.stackIsFull ( ) ) then // stack is not full
     S.push (v);                      // push the newly read value on the stack
   else                                  // stack is full
     averageStack = S.averageElem ( ) ;         // pop all elements and return their average
     if (averageStack >= theta) then   // if the avg is above theta

       OpenOutputFile (O);    // open output file
       if O.open then          // if output file is open
         fout ( averageStack);   // write avg over a group of sensors to the output file
       fi
       prev = prev + 10;   // increase prev. with the stackSize
       if StartSeq then       // is this the first event?
         prev = i;               // yes:  set the counters: prev and i right
         StartSeq = false;  // set the StartSeq false
       else                      // No: not the first event
         if ((i - prev) > gap ) then  // is the cumulated gap  too big ?
           StartEvent = true;        // yes: new event marker set to true
           prev = i;                        // begin counter of the next event is set to i
         fi // (i - prev) > gap
       fi  //  StartSeq
     fi    // (average >= theta)
   fi    //
   i++ ; // increase row counter
 od

 Close I;   // close Input file
 Close O;  // close Output file
fi
```

5.5 Data description

In this section we present the file formats used as output by the filter program. The first data file format is a XML file in which the detected events are specified with their date-time occurrence and the values detected above the threshold.
Currently only the data from the sensors s100 and s101 is collected and processed as output in the XML file. The processing of the data from more sensors can be added to the XML output file.

```
<bridge>
 bridgeID = "1"
 bridgeName = "Hollandse Brug"
 sensor threshold = -3
 Leading sensor = 101

 <event>
   eventID = "1"
  <sensor>
       sensorID = "101"
       sensorType = "LOAD"
       dateTime = "2008-10-24 06:15:10,840"
       value = -2.98235178
       rowCounter = 630
  </sensor>

  <sensor>
    sensorID = "102"
    sensorType = "LOAD"
    dateTime = "2008-10-24 06:15:10,840"
    value = 1.20642102
    rowCounter = 630
  </sensor>
  . . .
  <sensor>
    sensorID = "319"
    sensorType = "TEMP"
    dateTime = "2008-10-24 06:15:10,840"
    value = 10.26341127
    rowCounter = 630
  </sensor>

  <sensor>
    sensorID = "101"
    sensorType = "LOAD"
    dateTime = "2008-10-24 06:15:10,850"
    value = -2.62756395
    rowCounter = 631
  </sensor>
  <sensor>
    sensorID = "102"
    sensorType = "LOAD"
    dateTime = "2008-10-24 06:15:10,850"
    value = -1.25633427
    rowCounter = 631
  </sensor>
 . . .
  <sensor>
    sensorID = "319"
    sensorType = "TEMP"
    dateTime = "2008-10-24 06:15:10,850"
    value = 10.26341127
    rowCounter = 631
  </sensor>
 . . .
 </event>
</bridge>
```

The next representation of the output file is in the CSV format. This format can be used to import the detected events in different databases. The values of sensors s100 and s101 are both used as output. New data from other sensors can be added to the file.
The data represented as a csv file with nine found events:

```
EventId;msec;Date;Time;msec;s101;s102
1;630;2008-10-24;06:15:10;840;-2.98235;1.20642
...
1;709;2008-10-24;06:15:11;630;1.74817;4.62424
...
1;2736;2008-10-24;06:15:31;900;-2.66009;1.50504
2;2738;2008-10-24;06:15:31;920;-2.51817;1.71495
...
2;2891;2008-10-24;06:15:33;450;38.9225;32.5855
...
2;3059;2008-10-24;06:15:35;130;-2.62756;1.49025
2;19067;2008-10-24;06:18:17;730;-2.67783;0.958068
```

```
3;19068;2008-10-24;06:18:17;740;-2.98235;1.32173
3;19183;2008-10-24;06:18:18;890;5.04182;6.82692
...
3;20323;2008-10-24;06:18:30;290;-2.82565;0.955111
4;20324;2008-10-24;06:18:30;300;-2.31712;1.70313
...
4;20396;2008-10-24;06:18:31;020;4.20806;6.95109
...
4;20874;2008-10-24;06:18:35;800;-2.94392;1.27442
5;20880;2008-10-24;06:18:35;860;-2.58913;1.25373
...
5;20914;2008-10-24;06:18:36;200;-1.83816;1.91896
...
5;27458;2008-10-24;06:19:41;830;-2.74583;1.21529
6;27459;2008-10-24;06:19:41;840;-2.82565;1.1059
...
6;27526;2008-10-24;06:19:42;510;8.04869;9.0089
...
6;27550;2008-10-24;06:19:42;750;8.92681;10.5611
6;27551;2008-10-24;06:19:42;760;8.81446;10.4783
...
6;27552;2008-10-24;06:19:42;770;9.06873;10.4783
6;27553;2008-10-24;06:19:42;780;8.79376;10.023
6;27945;2008-10-24;06:19:46;700;-2.88479;0.996504
...
7;27947;2008-10-24;06:19:46;720;-2.73104;1.57008
...
7;27988;2008-10-24;06:19:47;130;-0.634835;3.1016
...
7;28630;2008-10-24;06:19:53;551;-2.9587;1.40451
8;28632;2008-10-24;06:19:53;571;-2.45313;1.24781
...
8;28700;2008-10-24;06:19:54;251;-0.587529;2.87985
...
8;29390;2008-10-24;06:20:01;151;-2.97644;1.13251
9;29392;2008-10-24;06:20:01;171;-2.61869;1.12364
...
9;29472;2008-10-24;06:20:01;971;2.87168;5.7448
...
9;29555;2008-10-24;06:20:02;801;-2.93505;1.47251
```

A short, compressed representation of the produced events can be considered in the next data representation. In this data representation the events are encoded in the representation. The start time of the event is given with its length in time and its global maximum value:

```
< starttime; timelength; maxvalue >
<060500;   30;-6.12>        // NE= 30sec
<060530;   10;20.12>        // DE= 10sec
<060540;  600;-6.51>        // NE= 10min
<061540;  600; 4.50>        // DE= 4sec
<061544; 3600;-5.67>        // NE= 1800sec
<064544; 7200;30.20>        // SE= 7200sec
<084544; ....; ....>        // xE=  ....
...
```

We can consider this data representation as a sketch of the filtered event. By decoding the event we don't have the exact detailed information about the event. This representation can be used as compressed information when we want to perform a search in a video file where the events are recorded.

5.6 Data transmission

In figure 9 we propose the configuration of the network. The micro controllers (MC1 through MC5) are limited in computational power and data storage. These micro controllers collect the sensory data from their sensors and store it on a *local server*. This local server is part of the infrastructure of the bridge and forms the first collecting and storage point of the accumulated data. At the moment the hardware runs under a MS-Windows server operating system. The data is collected on an external hard disk which is exchanged manually once a week.
The first step is to connectect the bridge local server to the World Wide Web through an ADSL high speed data link. The files produced by the local server can then be transferred by to a portal server. The portal which is connected to a computer GRID will facilitate the distributed parallel processing and the data storage of the transferred files. As soon as the files arrive, the portal server will distribute them to the computer Grid. Through use of parallelism, the calculations can be performed at the same time on different computers attached to the GRID. The results are returned to the portal and stored there. At any time the portal can be accessed by work stations at different locations for queries about the results.

In the source code given in appendix G, H (clients) and appendix I (server) we use a direct TCP/IP socket communication to transmit the filtered data to a desired server.
The filter application has a build-in TCP/IP socket client data transmission module as is given by Tanenbaum p.494 [5].

Here follows a short description of the client and server programs:

- From inside the filter program which is running on the computer where all the flat data files are located, the data transmission is done by TCP/IP sockets. The filter program acts like a client. The data filtered by a certain computed θ will be transmitted to a given IP-address and a given port number: IP XXX.XXX.XXX.XXX:yyyyy (i.e. IP 192.168.100.1:55555).

- A server program is listening at the IP-address of the client with the same port. The data is buffered by the server and can be written in a file. The file represents the filtered XML or CSV of the original flat file.

- The prototype of the filter program is looking only at the sensors s100 and s101. By adding more sensors to the output, more data can be transmitted. In case that we add more sensors to the data transmission, we need to re-adjust the buffer of our server.

- Both client and server are implemented on a MS-Windows platform. Because the Windows sockets library ver. 2.0 is fully compatible with UNIX, the porting of the server software to a Linux can be easily realised. In this case the client will run on a Windows computer and the server will listen and receive the data on a UNIX server.

- By both the client and the server being placed in the same LAN the data transmission tests completed without any problem. For the data transmission to different LANS, through firewalls, the IP-address rules with the defined ports should be added to the firewalls' existent rules.

Another communication method which we will consider is the Java Commodity Kit. This is a Java based tookit containing packages which are plugged into the computer Grid fabric and facilitate calls to the lower-layers of the Grid.

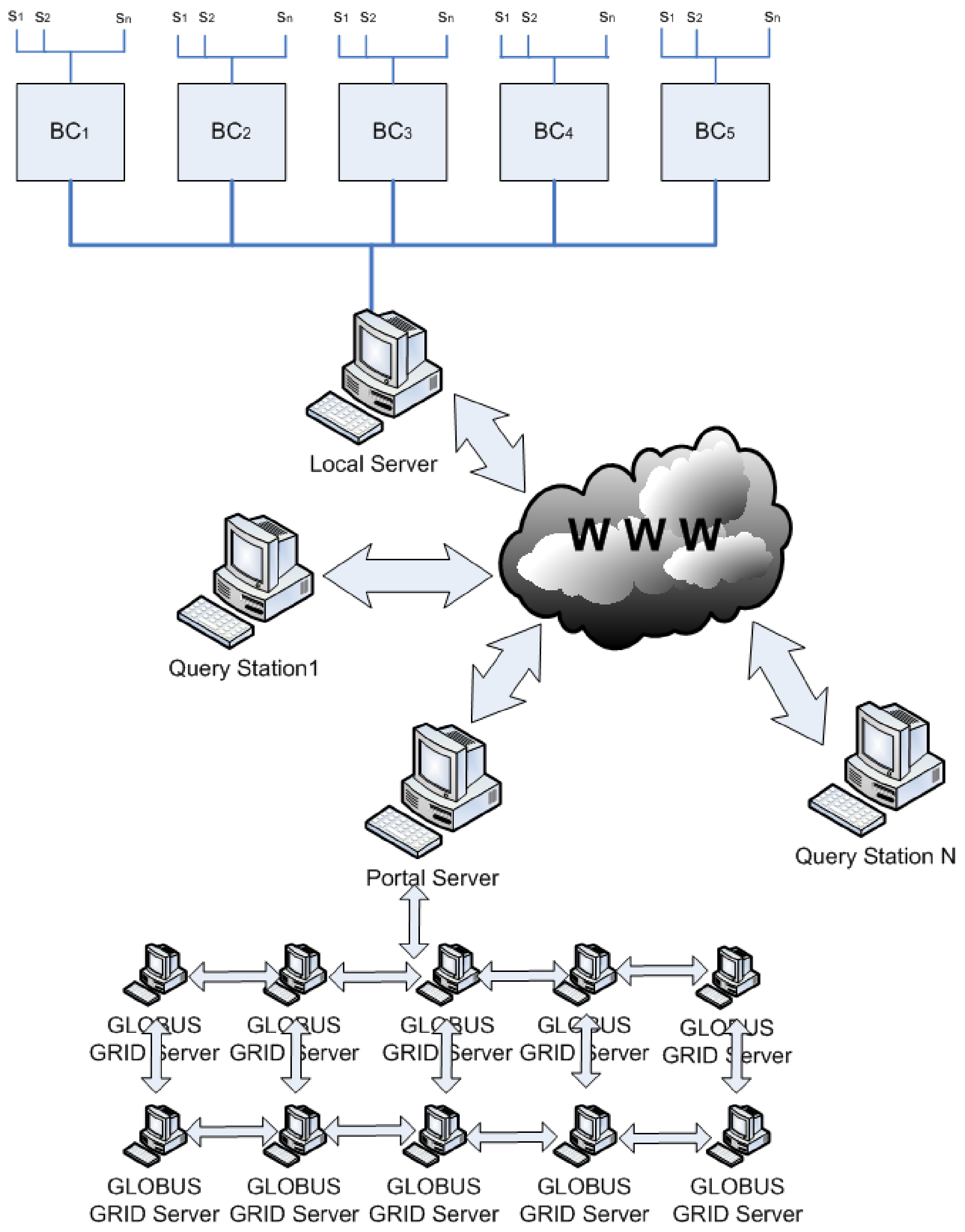


Figure 9: Data transmission scheme.

One possible communication sequence between a query station, a server running the filter program and the computer Grid can be seen in Figure 10. When a query station asks the server to compute a certain option of the filter program, the server transmits the results either back to the query station (in case that the option is a simple query) or the server will transmit the query to a Grid for advanced computations and after it receives the results from the Grid, it returns those results back to the query station.

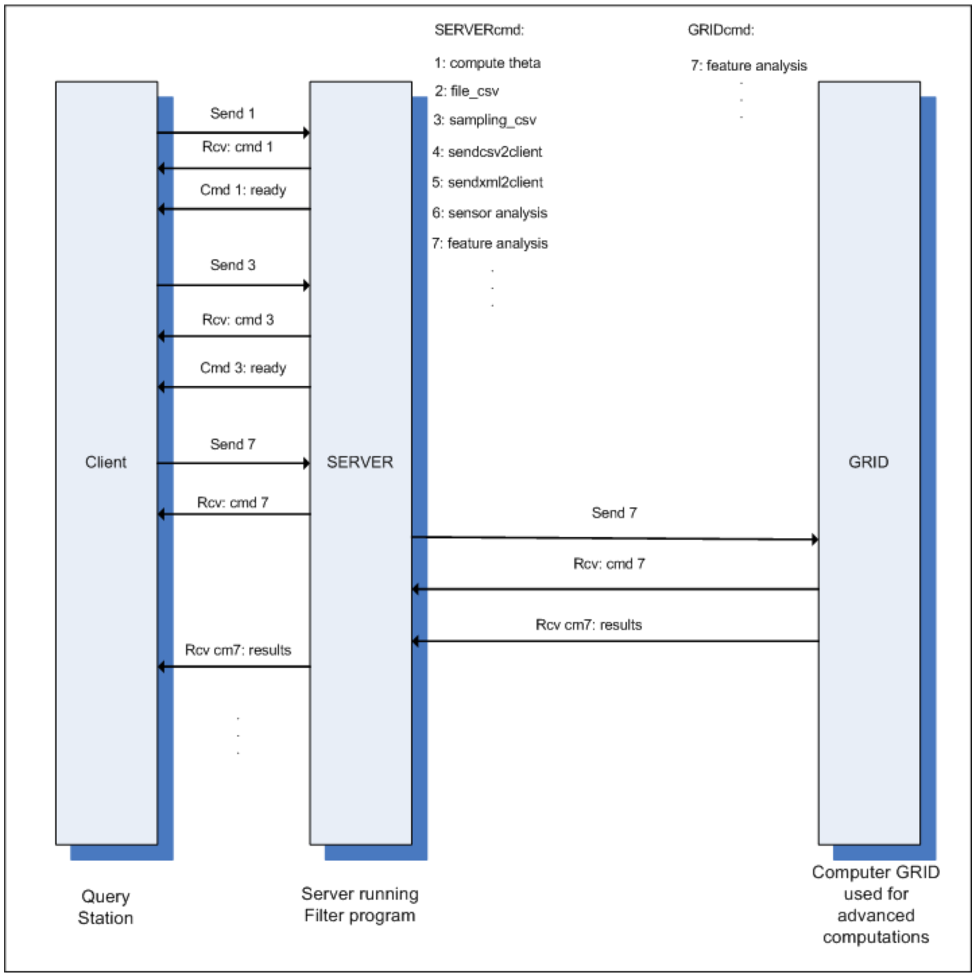
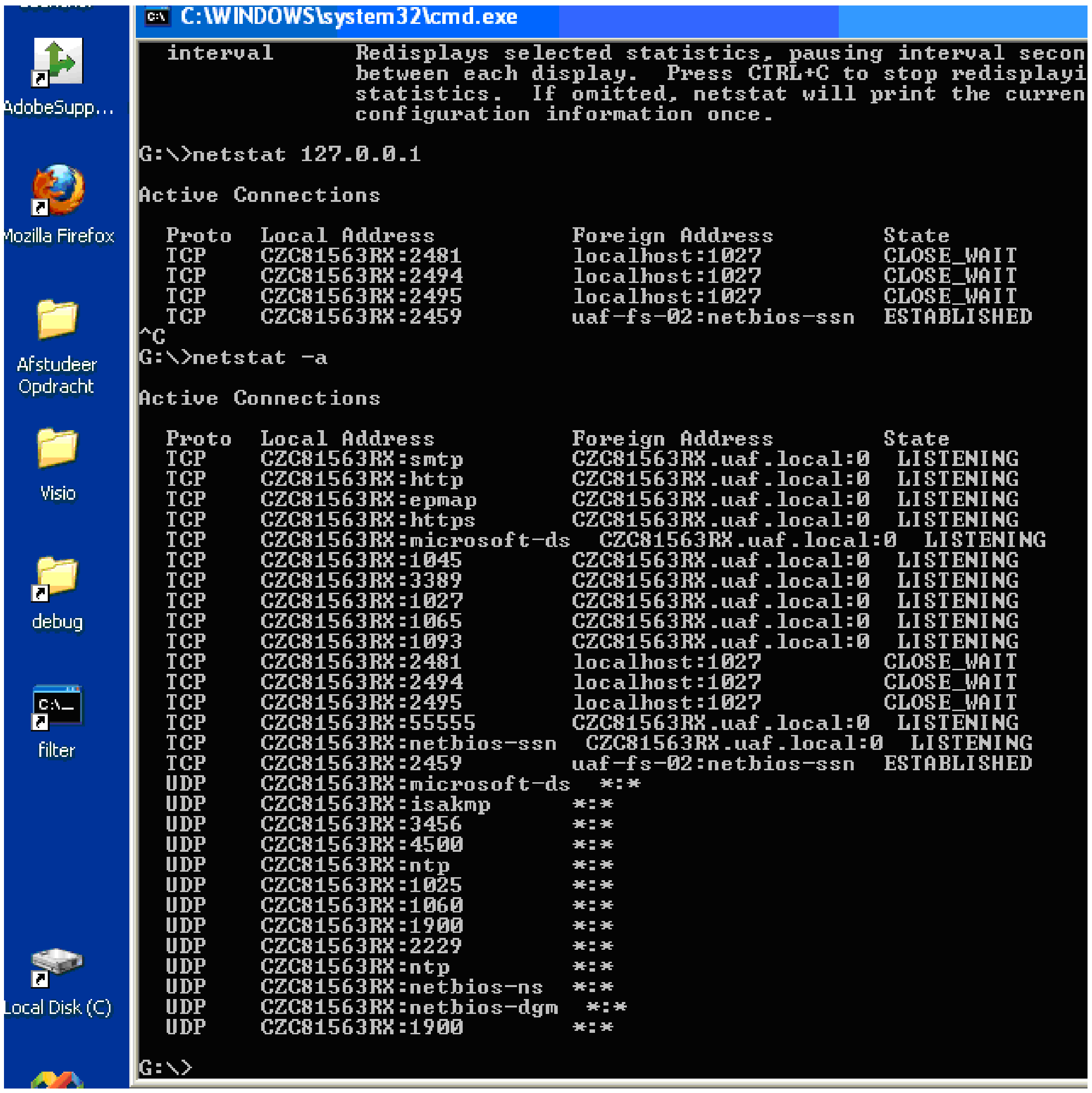


Figure 10: A data communication sequence between a query station, a server running the filter program and the Grid.

5.7 Tests

In Figure 11 we see, after running **netstat -a**, the server program **SocketServer.exe** listening at the local host (IP 127.0.0.1:55555):

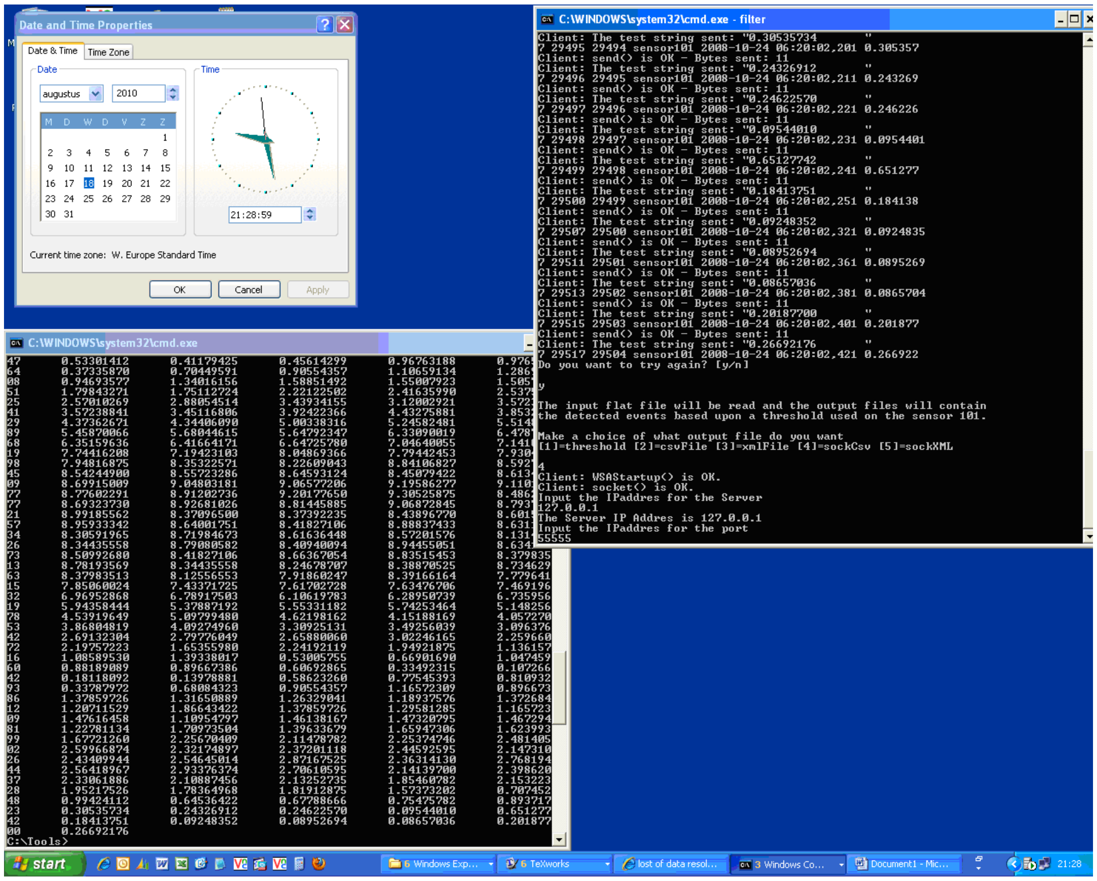
- Protocol: TCP
- Local Address: CZC81563RX:55555
- Domain: uaf.local
- Foreign Address: CZC81563RX.uaf.local:0
- Status: LISTENING



```
  interval       Redisplays selected statistics, pausing interval secon
                 between each display.  Press CTRL+C to stop redisplayi
                 statistics.  If omitted, netstat will print the curren
                 configuration information once.

G:\>netstat 127.0.0.1

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    CZC81563RX:2481        localhost:1027         CLOSE_WAIT
  TCP    CZC81563RX:2494        localhost:1027         CLOSE_WAIT
  TCP    CZC81563RX:2495        localhost:1027         CLOSE_WAIT
  TCP    CZC81563RX:2459        uaf-fs-02:netbios-ssn  ESTABLISHED
^C
G:\>netstat -a

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    CZC81563RX:smtp        CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:http        CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:epmap       CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:https       CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:microsoft-ds  CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:1045        CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:3389        CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:1027        CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:1065        CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:1093        CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:2481        localhost:1027         CLOSE_WAIT
  TCP    CZC81563RX:2494        localhost:1027         CLOSE_WAIT
  TCP    CZC81563RX:2495        localhost:1027         CLOSE_WAIT
  TCP    CZC81563RX:55555       CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:netbios-ssn   CZC81563RX.uaf.local:0  LISTENING
  TCP    CZC81563RX:2459        uaf-fs-02:netbios-ssn  ESTABLISHED
  UDP    CZC81563RX:microsoft-ds  *:*
  UDP    CZC81563RX:isakmp      *:*
  UDP    CZC81563RX:3456        *:*
  UDP    CZC81563RX:4500        *:*
  UDP    CZC81563RX:ntp         *:*
  UDP    CZC81563RX:1025        *:*
  UDP    CZC81563RX:1060        *:*
  UDP    CZC81563RX:1900        *:*
  UDP    CZC81563RX:2229        *:*
  UDP    CZC81563RX:ntp         *:*
  UDP    CZC81563RX:netbios-ns  *:*
  UDP    CZC81563RX:netbios-dgm   *:*
  UDP    CZC81563RX:1900        *:*

G:\>
```

Figure 11: Server listens to local host IP-address 127.0.0.1 port:55555.

In Figure 12 the client **filter.exe** (on the right) transmits its data to the server program **SocketServer.exe** (on the left):


Date and Time Properties
Date & Time
Time Zone
Date
augustus
2010
Time
21:28:59
Current time zone: W. Europe Standard Time
OK
Cancel
Apply
C:\WINDOWS\system32\cmd.exe
C:\WINDOWS\system32\cmd.exe - filter
start

```
47     0.53301412   0.41179425   0.45614299   0.96763188   0.976
64     0.37335870   0.70449591   0.90554357   1.10659134   1.286
08     0.94693577   1.34016156   1.58851492   1.55007923   1.505
51     1.79843271   1.75112724   2.22122502   2.41635990   2.537
25     2.57010269   2.88054514   3.43934155   3.12002921   3.572
41     3.57238841   3.45116806   3.92422366   4.43275881   3.853
29     4.37362671   4.34406090   5.00338316   5.24582481   5.514
89     5.45870066   5.68044615   5.64792347   6.33090019   6.478
68     6.35159636   6.41664171   6.64725780   7.04640055   7.141
19     7.74416208   7.19423103   8.04869366   7.79442453   7.930
98     7.94816875   8.35322571   8.22609043   8.84106827   8.592
45     8.54244900   8.55723286   8.64593124   8.45079422   8.613
09     8.69915009   9.04803181   9.06577206   9.19586277   9.110
77     8.77602291   8.91202736   9.20177650   9.30525875   8.486
77     8.69323730   8.92681026   8.81445885   9.06872845   8.793
21     8.99185562   8.37096500   8.37392235   8.43896770   8.601
57     8.95933342   8.64001751   8.41827106   8.88837433   8.631
34     8.30591965   8.71984673   8.61636448   8.57201576   8.131
26     8.34435558   8.79080582   8.40940094   8.94455051   8.634
73     8.50992680   8.41827106   8.66367054   8.83515453   8.379835
13     8.78193569   8.34435558   8.24678707   8.38870525   8.734629
63     8.37983513   8.12556553   7.91860247   8.39166164   7.779641
15     7.85060024   7.43371725   7.61702728   7.63476706   7.469196
32     6.96952868   6.78917503   6.10619783   6.28950739   6.735956
19     5.94358444   5.37887192   5.55331182   5.74253464   5.148256
78     4.53919649   5.09799480   4.62198162   4.15188169   4.057270
53     3.86804819   4.09274960   3.30925131   3.49256039   3.096376
42     2.69132304   2.79776049   2.65880060   3.02246165   2.259660
72     2.19757223   1.65355980   2.24192119   1.94921875   1.136157
16     1.08589530   1.39338017   0.53005755   0.66901690   1.047459
60     0.88189089   0.89667386   0.60692865   0.33492315   0.107266
42     0.18118092   0.13978881   0.58623260   0.77545393   0.810932
93     0.33787972   0.68084323   0.90554357   1.16572309   0.896673
86     1.37859726   1.31650889   1.26329041   1.18937576   1.372684
12     1.20711529   1.86643422   1.37859726   1.29581285   1.165723
09     1.47616458   1.10954797   1.46138167   1.47320795   1.467294
81     1.22781134   1.70973504   1.39633679   1.65947306   1.623993
99     1.67721260   2.25670409   2.11478782   2.25374746   2.481405
02     2.59966874   2.32174897   2.37201118   2.44592595   2.147310
26     2.43409944   2.54645014   2.87167525   2.36314130   2.768194
44     2.56418967   2.93376374   2.70610595   2.14139700   2.398620
37     2.33061886   2.10887456   2.13252735   1.85460782   2.153223
28     1.95217526   1.78364968   1.81912875   1.57373202   0.707452
48     0.99424112   0.64536422   0.67788666   0.75475782   0.893717
23     0.30535734   0.24326912   0.24622570   0.09544010   0.651277
42     0.18413751   0.09248352   0.08952694   0.08657036   0.201877
00     0.26692176
C:\Tools>
```

```
Client: The test string sent: "0.30535734       "
7 29495 29494 sensor101 2008-10-24 06:20:02,201 0.305357
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.24326912       "
7 29496 29495 sensor101 2008-10-24 06:20:02,211 0.243269
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.24622570       "
7 29497 29496 sensor101 2008-10-24 06:20:02,221 0.246226
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.09544010       "
7 29498 29497 sensor101 2008-10-24 06:20:02,231 0.0954401
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.65127742       "
7 29499 29498 sensor101 2008-10-24 06:20:02,241 0.651277
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.18413751       "
7 29500 29499 sensor101 2008-10-24 06:20:02,251 0.184138
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.09248352       "
7 29507 29500 sensor101 2008-10-24 06:20:02,321 0.0924835
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.08952694       "
7 29511 29501 sensor101 2008-10-24 06:20:02,361 0.0895269
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.08657036       "
7 29513 29502 sensor101 2008-10-24 06:20:02,381 0.0865704
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.20187700       "
7 29515 29503 sensor101 2008-10-24 06:20:02,401 0.201877
Client: send() is OK - Bytes sent: 11
Client: The test string sent: "0.26692176       "
7 29517 29504 sensor101 2008-10-24 06:20:02,421 0.266922
Do you want to try again? [y/n]

y

The input flat file will be read and the output files will contain
the detected events based upon a threshold used on the sensor 101.

Make a choice of what output file do you want
[1]=threshold [2]=csvFile [3]=xmlFile [4]=sockCsv [5]=sockXML

4
Client: WSAStartup() is OK.
Client: socket() is OK.
Input the IPaddres for the Server
127.0.0.1
The Server IP Addres is 127.0.0.1
Input the IPaddres for the port
55555
```

Figure 12: The client program communicates with the server by TCP/IP sockets.

In the Figure 13 we see a graph with the test results after running the filter program with different thresholds.

The average threshold computed for sensor s100, for the given flat data file, was: -6.11446. The initial input flat file size was 51.020 kb.

In the graph

- the blue line represents the threshold used as input
- the red line represents the TCP/IP socket transmission time in seconds. The server and client program, both run on the local host, using the IP-address 127.0.0.1:55555
- the yellow line shows the found events by the filtering process.

We remark that the longest transmission time is needed when the threshold is under the computed threshold average. In this case all the values computed by the client are transmitted to the server. The highest number of events (168) were detected around the average threshold. By using values above the average threshold, more data is filtered and as a result less data needs to be transmitted. The more data we filter the less data our filter program will transmit and the transmission time will decrease.

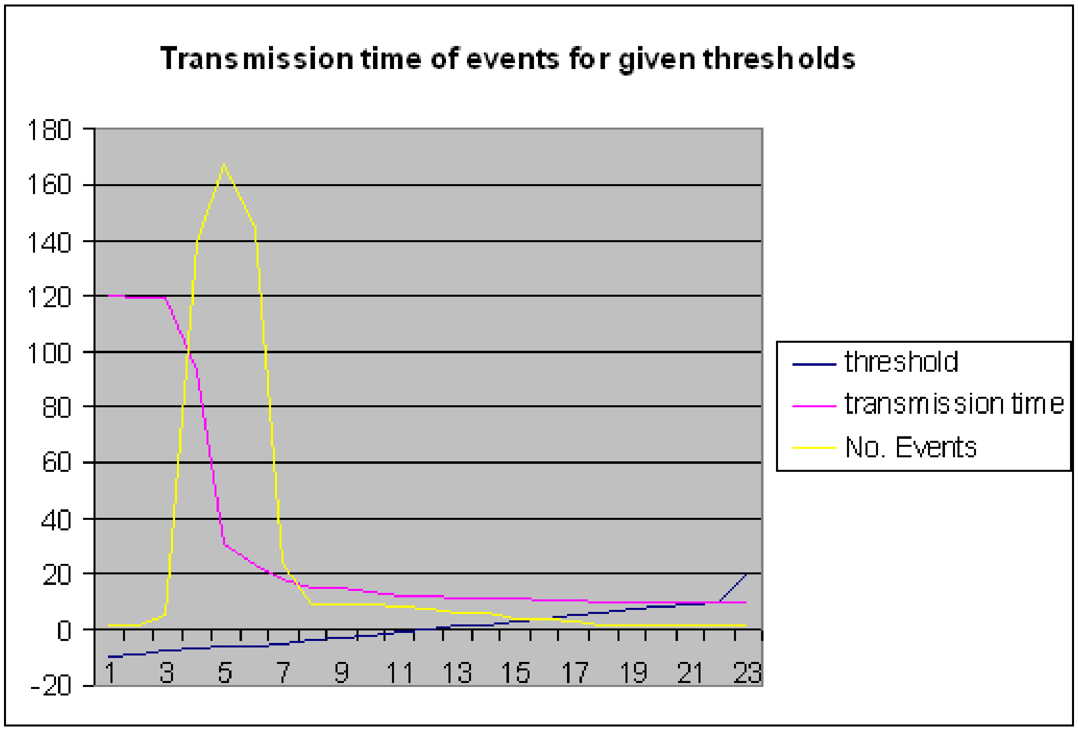


Figure 13: Transmission data time for the filter program, when it is run with different thresholds.

In the next table we see the tests for the data being transmitted by the filter program to the SocketServer for the sensor s100, depending of the value of the used threshold. The size of the data is also given:

Trial number	Threshold	Transmission time (sec)	Number of events	Data size (bytes)
1	-10	120.01	1	262,193
2	-9	119.56	2	261,846
3	-8	119.22	5	259.961
4	-7	93.20	139	193.963
5	**-6.11446**	**31.10**	**168**	**50.249**
6	-6	23.24	144	41.836
7	-5	18.22	24	18.401
8	-4	15.01	9	14.217
9	-3	14.49	9	11.980
10	-2	13.56	9	9.687
11	-1	12.01	8	7.552
12	0	11.90	7	6.242
13	1	11.50	6	5.215
14	2	11.20	6	4.501
15	3	11.10	4	3.834
16	4	10.50	4	3.245
17	5	10.40	3	2.701
18	6	10.20	2	2.498
19	7	10.11	2	2.277
20	8	10.10	2	2.106
21	9	10.08	2	1.661
22	10	10.01	1	1.574

In the next table we see the tests for the data being produced by the filter program using sampling with a fixed window size, depending of the value of the used threshold. Every 10^{th} data values are averaged by using a stack. The average is then compared with the threshold to detect the events. The detected number of events and the size of the data output is also given:

Trial number	Threshold	Time (sec)	Number of events	Data size (bytes)
1	-10	9.62	1	122,840
2	-9	9.56	1	122,636
3	-8	9.22	3	122.103
4	-7	9.32	81	99.192
5	**-6.11446**	**9.96**	**55**	**20.845**
6	-6	9.89	43	17.695
7	-5	9.77	12	8.563
8	-4	9.80	9	6.841
9	-3	9.81	9	5.634
10	-2	9.76	8	4.722
11	-1	9.65	8	3.719
12	0	9.62	6	3.126
13	1	9.60	6	5.215
14	2	9.55	5	2.386
15	3	9.52	4	1.980
16	4	9.48	3	1.612
17	5	9.43	3	1.327
18	6	9.41	2	1.286
19	7	9.38	2	1.125
20	8	9.37	2	1.084
21	9	9.35	1	838
22	10	9.33	1	798

We remark from the test results that by using sampling we detect the same number of events while the size of the data output can be reduced considerably. This is a positive factor when considering the further implementation of sampling in the data transmission modules of the filter program.

6 Globus GRID

In this section we introduce the Globus computer Grid. We will present its main components and their functions presented to the grid users. The computer grids are used for distributed parallel computations. In many application areas like the bridge monitoring system, the extra computational power is needed to perform complex calculation with respect to the detection of features in the events, the correlation between the sensors, context shift detection, visualization of data, correlation between the events and video data, etc.

6.1 Introduction

Looking at a computer Grid there are two different perspectives:

- **The end-user perspective:**
  From the perspective of the end-user the Grid appears as a virtual self managing, super-computer on which the user can run their applications.
  For certain applications the need for extra computational capacity and data storage is high. A computer Grid offers for those users who run complex, computational demanding applications a simple solution: plug-in into an available computer Grid and gain computer power instantly to run such complex algorithms for solving difficult engineering problems.

  To fully take advantage of the parallel processing power of a computer Grid, the end-user needs to understand parallelism and divide their algorithms into smaller, independent code that can run in parallel on the computer Grid. A scalable application that can be divided in 100 different, independent smaller subtasks will run 100 times faster if it can use 100 CPUs at the same time.

  Regarding parallelism, some of the important questions someone should ask, are:

  – can the used algorithm be split under different CPUs to facilitate parallelism?
  – in how many distinct parts should it be parted?
  – how independent are those parts among each other?

  A computer grid can also provide the end-users with additional storage, other resources, special equipment, software packages, licenses or services.

- **The system-perspective:**
  From the systems forming the Grid infrastructure, the perspective can be seen as a collection of heterogeneous computers, high-performance servers, super-computers and other hardware devices which can share their various resources to facilitate a higher degree of their usage.

  The check list of Foster [6] gives a comprehensive, general idea about what a computer grid is:

  1. *A set of heterogeneous, distributed resources which are coordinated by some Grid software.*

2. *The Grid software should use open, general-purpose, standard protocols for authentication, authorization, resource discovery and resource access.*
3. *The Grid software should be capable to deliver different sort of quality of services for different users.*

The next topics are of importance, when we consider any sort of computer grid architecture:

- **Security**
  Some sort of basic security must be provided, supporting the mechanisms for authentication, authorization and the communication between the grid computers. In Globus Toolkit the *Grid Security Infrastructure* (GSI) provides these security components.
  - *authentication*
    is the process of verifying some claimed identity. This can be extrapolated beyond users, to computers, services, applications or any entity that may be required to authenticate.
  - *authorization*
    can be expressed by the assurance that a user or computer may use a service which is allocated by certain given access rights, the usage being based only on these access rights.
  - *data integrity*
    gives the assurance that no data can be altered or destroyed if there isn't any sort of authorization given for such actions.
  - *data privacy*
    is defined by the assurance that the data cannot be revealed to anyone who hasn't the authorization to access the data.
  - *key management*
    deals with the secure generation, distribution and storage of keys used in cryptography for the authentication purposes.
    Here are some general concepts about the data encryption:
    - symmetric encryption: one key for both encryption and decryption;
    - asymmetric encryption: two keys, one for encryption and one for decryption;
    - SSL/TLS - secure socket layer/transport layer security are basically the same protocols, named differently;
    - PKI - public key infrastructure: components and protocols to facilitate the PKI environment;
    - mutual authentication: two parties use their PK stored in their certificates to authenticate with one another.

On top of the Grid Security Infrastructure (GSI) which provides the security functions, there are three main blocks:

- **Resource management**
  In Globus Toolkit the functionality of this block is performed by GRAM (grid resource allocation manager) and GASS (global access to secondary storage).
  - *Resource allocation*
    allocates resources based on their availability, load balancing, etc.
  - *Submitting jobs*
    manages the running of executable files and receiving results.
  - *Managing job status*
    gives information about the status of the running files and their progress in execution.
- **Information services**
  Based on the LDAP (lightweight directory access protocol) query language these services collect information from the grid concerning the available resources and facilitate queries about the collected information.
  In Globus Toolkit the MDS (monitoring and discovery service) is responsible for this function due to its two main components:
  - *GRIS* is the grid resource information service. Some information about the resources is static: like computers in the grid and some is dynamic: like CPUs or disk storage. Through an interface with GRIS, the users can add their own resources to the grid. GRIS reports this information to a hierarchy of GIIS servers kept on the grid.
  - *GIIS* is the grid index information service. Using LDAP query language the GIIS servers on the grid can be queried about the available resources on the grid.
- **Data management**
  The data management provides support for
  - the transfer of files between the different computers available, attached to the grid.
  - the management of these data file transfers.

  The data management key component is the GridFTP which is responsible for the secure, high-performance data transfer.

  The data management provided by Globus toolkit consists of a client, server and SDK packages.

  In figure 14 we describe the standard file transfer and in figure 15 a third-party file transfer in a Grid.

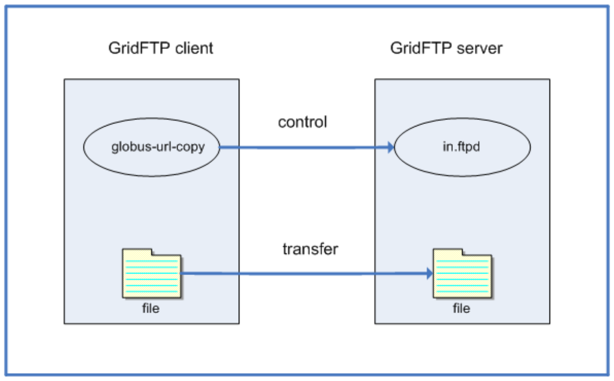


Figure 14: Standard file transfer in a Grid.

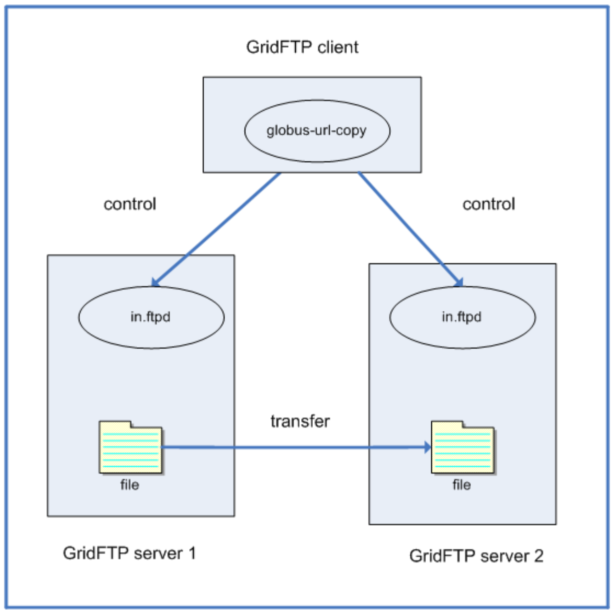


Figure 15: Third-party file transfer in a Grid.

- **Data GRID**
  Through collaboration, data grids can also include new concepts such as the concept of a federated database: the GRID makes a heterogeneous group of databases available which behave like a single database. This database will have:
  - a single query point,
  - a local data source used to store query results,
  - federated DBMS, catalog, heterogeneous data sources;
  - it will use "wrappers" to access heterogeneous data sources,
  - respect local and global DB policies,
  - maximize the utilization of existing storage,
  - reduce the complexity of data management.

In Figure 16 we see the architecture of a federated DBMS in a Grid.

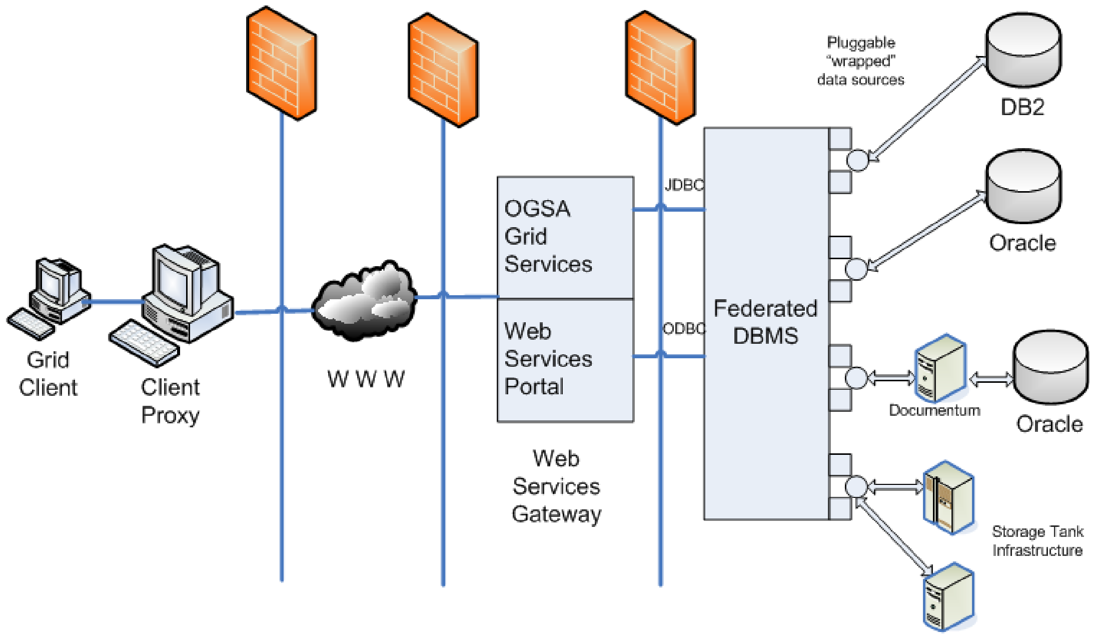


Figure 16: Federated DBMS in a Grid.

6.2 Globus: Java Commodity Grid kit (CoG)

The computer science community has witnessed the development of two parallel worlds:

- on one side there is the "commodity" world referred by a broad spectrum of distributed programming technologies represented by Java, Jini, DCOM, CORBA mainly used on desktop computing level, focused on component composition.
- on the other side the high-performance distributed "Grid" computing, used for scientific computational demanding problems.

The *Java Commodity Grid (CoG)* in trying to bring these two worlds closer to each other has defined two main goals:

–a) to give the Grid developers the opportunity to exploit some of the commodity technologies.

–b) to export Grid technologies to commodity computing.

With these goals being formulated, the birth of a new set of tools has been marked. In the paper of Laszewski [8] we find the definition of a commodity grid toolkit:

Definition: A Commodity Grid Toolkit (CoG Kit) defines and implements a set of general components that map Grid functionality into a commodity environment/framework.

Here are some of the advantages of why to use the Java language for building a Java Commodity Grid Toolkit:

1. **Java programming language** offers some of the elements:
   - object-oriented: objects, class, inheritance, encapsulation, garbage collection,
   - packages: facilitating to build large-scale software projects,
   - interfaces: a contract between a class and the outside world.
2. **Class library** there exists a wide variety of class libraries needed for the grid access:
   - specific functions,
   - socket communication,
   - access SSL, etc.
3. **Components** through JavaBeans, Java offers a component based architecture and software program development.
4. **Deployment**: easy deployment of newly developed software, functionality.
5. **Portability**: to many different platforms; the concept of *"write-once-run-anywhere"*.
6. **Maintenance**: integrated documentation facility. JavaBeans components can easily be integrated in different IDE's (interface development environments) .
7. **Performance**: some of present Java applications come close in performance to that of C or Fortran similar applications.

8. **Gadgets**: java has been successfully ported to many different gadgets as: PDA's, smart cards and other smart devices.

9. **Industry**: the need for longevity of a technology is required for scientific projects. Java is a suitable candidate when considering grid applications.

10. **Community**: Java is well established in the academic world, forming a wide community of users.

The distinct layers of a computer grid are visible in the figure 17. On top there is the application layer. Supporting the applications there are the application toolkits. Under the toolkits there is the grid services layer which is supported by the grid fabric layer.

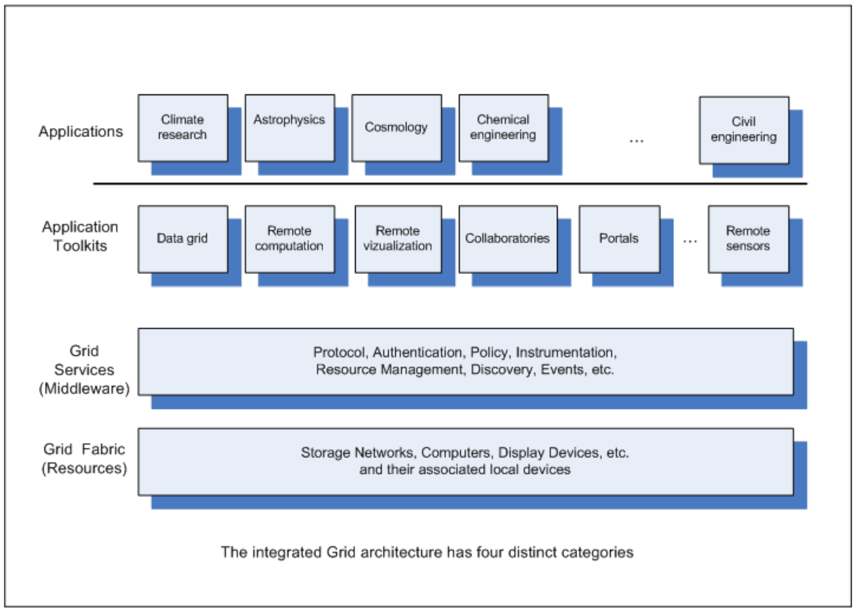

Applications
Climate research
Astrophysics
Cosmology
Chemical engineering
...
Civil engineering
Application Toolkits
Data grid
Remote computation
Remote vizualization
Collaboratories
Portals
...
Remote sensors
Grid Services (Middleware)
Protocol, Authentication, Policy, Instrumentation, Resource Management, Discovery, Events, etc.
Grid Fabric (Resources)
Storage Networks, Computers, Display Devices, etc. and their associated local devices
The integrated Grid architecture has four distinct categories

Figure 17: The computer grid distinct layers

- **The Grid Fabric** (the "resources")
  This layer provides the implementations of basic Grid operations, i.e.: scheduling system, storage system, QoS for a network router. The implementation of the basic Grid services are resource-specific.

- **The Grid Services** (the "middleware")
  The Grid fabric layer facilitates the implementation of resource-independent, application-independent Grid services. As an example for such a grid service we can mention the authorisation and authentication service. The information service, providing uniform access to the grid resources and structure is another example.

- **The Grid Toolkits** (the "toolkits")
  The previous two layers enable the creation of more application-specific services as toolkits: such a toolkit facilitates the use of distributed data management for data the specific applications or it can enable some sort of flow management capabilities for creating collaborative work environments.

- **The Grid applications** (the "applications")
  On top of the grid services and toolkits, facilitating their implementation, are the applications.

In the next figure, figure 18 the distinct layered components of The Java Commodity Kit are visible:

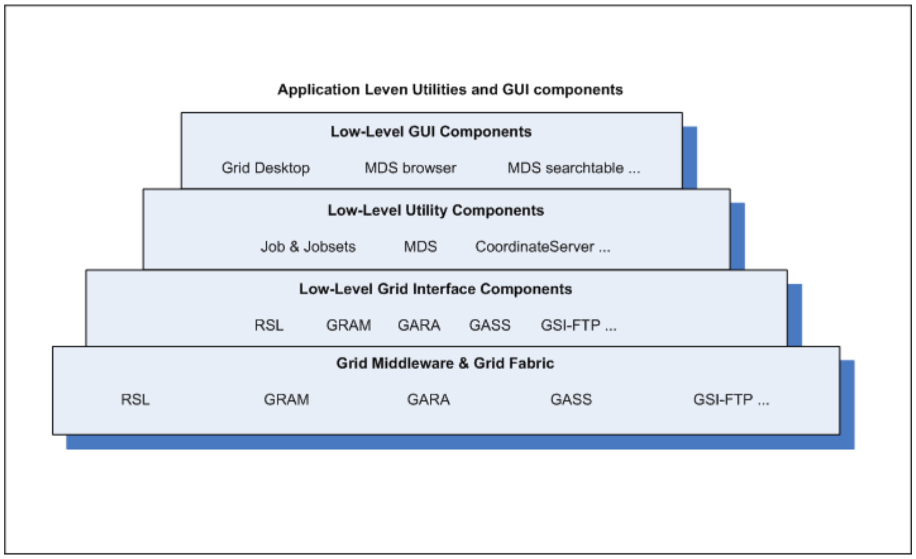


Figure 18: The Java Commodity Kit components

- **Low-level Grid Interface components**
  These components map into commonly used Grid services as: services using GSI (grid security infrastructure); MDS (globus meta-computing directory services) which provides services through LDAP to access information about the structure and state of Grid resources and services; GRAM service facilitates resource management services, by supporting the allocation and management of computational and other resources of the grid.; GASS + GSI services enables FTP; no graphical components are available at this level.

  - **RSL** Java package *org.globus.rsl*
    This package provides methods for creating, manipulating and validation of RSL-expressions in Globus.

- **Low-Level Utility Components**
  gather some of the utility functions designed to increase the functionality beyond those components implemented in the C programming language of the Globus toolkit. I.e.: MDS components for the user to find all compute resources by using XML language; RSL (GLOBUS job submission language) that locate he geographical coordinates of a compute resource or test if that resource is alive. There are no graphical components available at this level.

- **Common Low-Level GUI Components**
  some of the GUI components that can be reused by application developers. I.e.: LDAP attribute editors, RSL editors, LDAP browsers, search components, etc.

- **Application-specific GUI Components**
  are used to simplify the interface between the applications and the CoG Kit components. I.e.: stock-market monitor components, graphical climate data display component, specialized search engine for the climate data, etc.

We will look in detail at some of the mentioned components to get a view about how they are implemented and how they work by using a Java skeleton program. In the following Java code we see how some of the components are initialized, what parameters they take and how the structure of a Java CoG kit is being built for connecting and transferring data to and from the Grid, concerning a specific application:

```
//------------------------- Java CoG code ----
// Step 0. Initialization
String mdsServer = "mds.globus.org";
MDS mds=new MDS(mdsServer,"389");

// JOB SUBMISSION
//Step 1. Search for an available machine
result = mds.search ("o=Grid","&((objectclass=GridComputeResource)(freenodes=64))","contact");

// Step 1.a) Select a machine
machineContact = <select the machine with minimal execution time from
the contacts that are returned in result>

// Step 2. Prepare the data for the experiment
// Step 2.a) Search for the climate data and return
// the attributes: server,port,directory,file
dn = mds.search("(objectclass=ClimateData)(year=2000)(region=midwest)","dn",MDS.SubtreeScope);

result = mds.lookup (dn, "server port directory file");

// Step 2.b) download the data to the machine

filename = result.get("filename");

sourceURL = result.get("server")+":"
+ result.get("port")+"/"
+ result.get("directory")+"/"
+ filename;

destinationURL = "gsi-ftp://"
+ "machineContact" // destination
+ "~/" // directory
+ filename; // filename

UrlCopy.copy (sourceURL, destinationURL);

// Step 3. Prepare a description for running the model
RSL rsl = new RSL("(executable=climateModel)(processors=64)"
+ "(arguments=-grads)(arguments=-
out map.grads)"
+ "(arguments=-in " + filename +")");

// Step 4. Submit the program
GramJob job = new GramJob(rsl.toString());
job.addJobListener(new GramJobListener() {
public void stateChanged(GramJob job) {

// react to job state changes
}
});
try{
job.request(machineContact);
} catch (GramException e) {
// problem submitting the job
}
//------------------------- Java CoG code ----
```

7 Hidden Markov Models

In this section we will look at algorithms and modeling tools that we can use once the data is on the GRID. With the help of hidden Markov models we can try to differentiate between the states in which the bridge will emit certain events with certain probabilities. Based on the assumption that the system is in a state where the probability is very high of emitting a certain event type we could use that information in building a real-time system which tries to minimize the computations of what the system will predict based on the previous observations.

7.1 Introduction

According to Huang [3], a hidden Markov model (in short HMM) M consists of the triplet $M = (Q, \Sigma, \Theta)$, where:

- Q = is a set of finite states.
- Σ = is a finite alphabet of symbols, the symbols being emitted from the states.
- Θ = is a set of probabilities:
  - given two states $u, v \in Q$ we define the *state transition probability* $a_{u,v}$ as the probability of the model to make the transition from the state u to the state v:

$$a_{u,v} = P(s_t = v | s_{t-1} = u) \tag{7.1}$$

  - given a state $u \in Q$ and a symbol $k \in \Sigma$ we define the *emission probability* $e_u(k)$ as the probability of the symbol k to be emitted in the state u. Let X be a sequence of observed symbols $\mathbf{X} = (x_1, x_2, \ldots, x_t, \ldots)$ produced by an unobserved (hidden) sequence of states $\mathbf{S} = (s_1, s_2, \ldots, s_t, \ldots)$ in the HMM. The emission probability $e_u(k)$ can be written as:

$$e_u(k) = P(x_t = k | s_t = u) \tag{7.2}$$

The probability for the whole sequence of symbols $\mathbf{X}$ as being generated by the sequence of states $\mathbf{S}$ can be written:

$$P(X, S) = a_{s_0,s_1} \cdot \prod_{i}^{L} e_{s_i}(x_i) \cdot a_{s_i,s_{i+1}} \tag{7.3}$$

where the state $s_0 = \textit{Begin}$ state and the state $s_{L+1} = \textit{End}$ state.

In the next figure, Figure 19 we see an example of a HMM with two states and three emission symbols in each state.
Assuming that i represents our *Good* state and j the *Bad* state the state transition matrix $a_{i,j}$ and the emission matrices $e_i(k)$ and $e_j(k)$ follow:

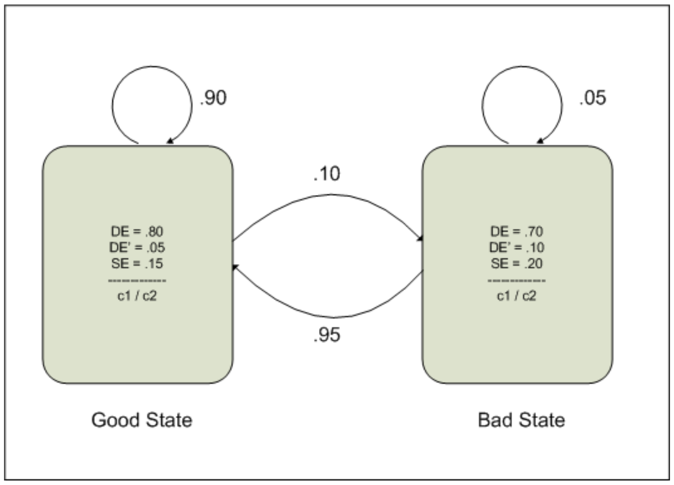


Figure 19: HMM representing the Good and Bad states with their emission of DE= dynamic, DE' = dynamic above maximum and SE= static events.

$$a_{i,j} = \begin{bmatrix} .90 & .10 \\ .95 & .05 \end{bmatrix} \tag{7.4}$$

$$e_{Good}(x) = \begin{bmatrix} .80 \\ .05 \\ .15 \end{bmatrix} \tag{7.5}$$

$$e_{Bad}(x) = \begin{bmatrix} .70 \\ .10 \\ .20 \end{bmatrix} \tag{7.6}$$

By adding a new state called *Very Bad state* we get the following hidden Markov model:

$$a_{i,j} = \begin{bmatrix} .985 & .10 & .005 \\ .950 & .04 & .010 \\ .970 & .02 & .010 \end{bmatrix} \tag{7.7}$$

The emission probabilities for the state Good and Bad remain the same but we have a new emission probability for the third state, called *VeryBad state*:

$$e_{Good}(x) = \begin{bmatrix} .80 \\ .05 \\ .15 \end{bmatrix} \tag{7.8}$$

$$e_{Bad}(x) = \begin{bmatrix} .70 \\ .10 \\ .20 \end{bmatrix} \tag{7.9}$$

$$e_{VeryBad}(x) = \begin{bmatrix} .55 \\ .20 \\ .25 \end{bmatrix} \tag{7.10}$$

The parameters for a HMM can be set in two ways:

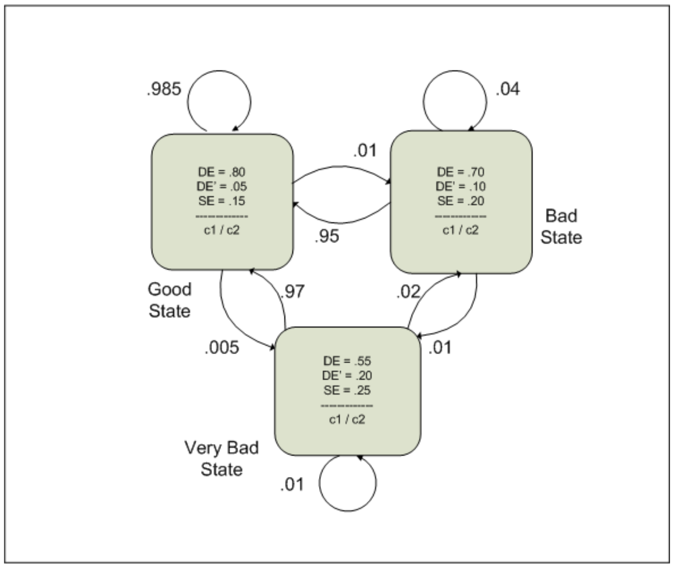


Figure 20: A HMM with three hidden states.

- a HMM can be trained from initially unlabeled sequences. Training can be done by:
  - Baum-Welch training algorithm
  - Gibbs sampling
  - simulated annealing
  - genetic algorithm training methods.
- a HMM can be build from pre-labeled sequences (the state paths are assumed to be known). The parameters for the HMM will be found by converting observed counts of emission and state transitions into probabilities.

7.2 Viterbi algorithm

Given a hidden Markov model $M = (Q, \Sigma, \Theta)$ and a sequence of observed symbols $\mathbf{X} = (x_1, x_2, \ldots, x_{t-1}, x_t, \ldots)$, generated by a sequence of states $\mathbf{S} = (s_1, s_2, \ldots, s_{t-1}, s_t, \ldots)$ in the given HMM we want to know the most probable path of states s_t that emitted the symbol sequence x_t.
We define with $v_k(i)$ the probability of the most probable path of states $(s_1, s_2, \ldots, s_k)$ generating the sequence of symbols $(x_1, x_2, \ldots, x_i)$:

$$v_k(i) = \max_{\{S|S_i=k\}} P(x_1, \ldots, x_i, S) \tag{7.11}$$

Viterbi algorithm finds the most likely path of states that have generated the sequence of observed symbols:

1. **Initialize:**

$$v_{begin}(0) = 1 \tag{7.12}$$

$$\forall i > 0 \;\; v_{s_i}(0) = 0 \tag{7.13}$$

2. **Recursion:**
For each emitted symbol i = [0, ...,L-1] and for each state $l \in Q$ recursively calculate:

$$v_l(i+1) = e_l(x_{i+1}) \cdot \max_{k \in Q}\{v_k(i) \cdot a_{kl}\} \tag{7.14}$$

3. **End step:**
The value of $P(X, \Pi^*)$ is:

$$P(X, \Pi^*) = \max_{k \in Q}\{v_k(L) \cdot a_{k,end}\} \tag{7.15}$$

We can reconstruct the path of most likely hidden states Π^* which have emitted the sequence of symbols X by keeping back pointers during the recursive calls and by tracing them back after the recursion.

Because we are dealing with probabilities and as a consequence with numbers on the interval [0,...,1] , the repeated multiplications will result in underflow. We will use logarithms to avoid such side effect:

Viterbi algorithm using logarithms:

1. **Initialize (Viterbi logarithmic):** because of the logarithms we initialize differently: in the begin state (with probability 1, we initialize to 0), all other states (with probability 0, we initialize to minInt).

$$v_{begin}(0) = 0 \tag{7.16}$$

$$\forall i > 0 \;\; v_{s_i}(0) = -\infty \tag{7.17}$$

2. **Recursion (Viterbi logarithmic):**
For each i = [0, ...,L-1] and for each $l \in Q$ recursively calculate:

$$v_l(i+1) = log\ [e_l(x_{i+1})] \cdot \max_{k \in Q}\{v_k(i) \cdot log\ (a_{kl})\} \tag{7.18}$$

3. **End step (Viterbi logarithmic):**
The value of $P(X, \Pi^*)$ is:

$$P(X, \Pi^*) = \max_{k \in Q}\{v_k(L) \cdot log\ (a_{k,end})\} \tag{7.19}$$

The first-order Markovian assumption states that the probability of being in a state q_t depends only of the probability of being in the previous state q_{t-1}:

$$P(q_t | q_{t-1}) \tag{7.20}$$

The first-order Markovian assumption restricts the accurately modeling of time-series data with highly varied dynamics. In this case we can increase the accuracy by using the probability not only depending of the previous state but to a certain n-length of states:

$$P(q_t \mid q_{t-1}, \cdots, q_{t-n}) \tag{7.21}$$

The emitted probability density function would be in that case also adapted to the n-length state sequence:

$$P(e_{q_t}(x) \mid q_{t-1}, \cdots, q_{t-n}) \tag{7.22}$$

For fully connected HMMs with a constant number of state transitions per state the complexity of the Viterbi algorithm is: $O(NM)$ both in time and space, where N is the length of the observed sequence of symbols and M is the number of states.

8 Future work and conclusion

8.1 Future work

Here we list some of the possibilities for future work for sensors:

- **smoothing of the sensor curve**
  Work with a previous and current pointer to the value of the sensor data. Use some sort of average between these two values in the filtering process based on some constant weights:

  $$v_t := \frac{w_1 \cdot v_{t-1} + w_2 \cdot v_t}{2} \tag{8.1}$$

  where $t \in \mathbf{N}, w_1, w_2 \in \mathbf{R}$ and are constant. For example we can chose as weight for the previous sensor value $w_1 = .25$ and for the weight of the current value $w_1 = .75$

- **sampling of the sensory data**
  We have used a stack of 10 elements to buffer the sensory data. For future work we can use different stack sizes and other ADS for sampling. In place of averaging over all the elements on the stack, we can use:

  - an average over a subset or
  - we can randomly pick-up a subset of values from the buffer and perform an average only over that subset.

- **wavelet transform of the sensory data**
  The sensory data can be transformed using a wavelet function to a new domain. The produced output can then be analyzed for new, specific patterns.

- **find the most correlated sensors when an event occurs**
  When an event takes place on the bridge, some of the sensors close to the event will react in some sort of cluster. To find their specific relation with respect to the event and try to find the most correlated sensors in that case, will mean that we have to perform a sensor space search and identify those most correlated sensors.

- **identifications of event**
  INPUT: Given a certain correlation function between a set of sensors
  OUPUT: Identify the type of event; in case of a dynamic event identify its characteristics (speed, direction, amplitude)

- **features of events**
  INPUT: Given a dynamic event with its features
  OUPUT: Identify and classify from the given event the features: the number of relevant features; specify in detail its main features: the local minima, maxima with their corresponding time durations (analyse in detail each feature of a given event).

- **transient, dynamic clustering of events**
  The dynamic nature of events combined with the speed of their change will affect a group of sensors greater than other groups. The forming of highly related, affected sensor groups has a dynamic nature. Identify the groups and specify their forming and change.
  INPUT: Given a dynamic event at time t_i
  OUPUT: Identify for the given event the group of sensors affected. Follow the event during its transition (from its birth till its death) by specifying the dynamic change of groups of sensors affected by it.
  Use some sort of classifier to specify groups: entropy, Mahalanobis distance, Euclidian distance, etc.

Next we list some points for future work concerning the Grid:
How can we split-up the application to facilitate the Grid parallel processing?

- general benefit of parallel computing for an application
- which parts of the algorithms can be used in parallel
- specific concerns dealing with parallel, distributed programming

8.2 Conclusion

By analyzing a real-time system, the bridge called "De Hollandse Brug" , we have tried for the real-time system to:

- find a suited threshold for its sensory data,
- reduce the data by finding events,
- reduce the data by using sampling,
- give a new description to the data (XML),
- transport the data (by events) to a specific location,
- describe the GRID architecture for the data storage and further computation,
- introduce a model for the real-time monitoring of the bridge.

These steps represent the first attempt to deal with complex data streams in real-time systems.

9 Acknowledgements

We thank Joost Kok from Liacs for the support, advice, ideas and corrections during the project and the thesis.

We thank Bas Obladen, Carlos Bosma, Hessel Galenkamp from Strukton for their help for the system requirements and for explaining the implemented system through the available data and documentation.

We thank Arne Koopman, Arno Knobbe from Liacs for their advice during the project and for their related documents.

The InfraWatch project is funded by the Dutch funding agency STW, under project number 10970.

References

[1] C.F. Bosma, B. Obladen, *Onderzoek beperking Verkeershinder bij onderhoud door beton overlagingen*, 2009, Strukton Civiel, internal document, not available.

[2] Ian H. Witten, Eibe Frank, 2nd edition, *DATA MINING Practical Machine Learning Tools and Techniques*, 2005, ISBN: 978-0-12-088407-0, 525pages.

[3] Xuedong Huang, Alex Acero, Hsiao-Wuen Hon, *Spoken Language Processing, A Guide to Theory, Algorithm, and System Development*, 2001, ISBN: 0-13-022616-5, Hardcover, 980pages.

[4] Arne Koopman, Arno Knobbe, Marvin Meeng, *Pattern Selection Problems in Multivariate Time-Series using Equation Discovery*, 2010.

[5] Andrew S. Tanenbaum, fourth edition *Computernetwerken*, vertaling van Andrew S. Tanenbaum, *Computer networks*, 2003, Prentice-Hall, ISBN 90-430-0698-X.

[6] Ian Foster, *What is the Grid? A Three Point Checklist*, July 20, 2002.

[7] Luis Ferreira, Viktors Berstis, Jonathan Armstrong, Mike Kendzierski, Andreas Neukoetter, Masanobu Takagi, Richard Bing-Wo, Adeeb Amir, Ryo Murakawa, Olegario Hernandez, James Magowan, Norbert Bieberstein *Introduction to Grid Computing with Globus*, September 2003, IBM Redbooks.

[8] Gregor von Laszewski, Ian Foster, Jarek Gawor, Peter Lane, *A Java Commodity Grid Kit*, November, 2000.

10 Appendix A: threshold.h

The source code of the filter program implemented in C++:

```
// ======================================================
// name: threshold.h
// This program will compute a threshold based on sensor s100 from a flat data file
// datum 27-06-2010
// ======================================================


#include <string>
#include <fstream>
#include <iostream>


#include <Windows.h>
#include <stdio.h>

using namespace std;


int thresholdCalc() {

  SYSTEMTIME st;

  string fileName, fileNameOut;
  string line, c_jaar, c_mnd, c_dag, c_uur, c_min, c_sec, c_msec, cs101, cs102;
  string subline;

  ifstream fin;
  ofstream fout;

  float threshold101, s101, threshold102, s102, max101, max102, min101, min102, total=0.0;

  int jaar, mnd, dag, uur, min, sec, msec;
  int tmp, i, event101, eventId, sem_Event, sem_FirstEvent, prev, event_duration;
  int StartEventCounter, EndEventCounter;


  threshold101  = -3;
  event101 = 0;   // event affecting sensor 101
  eventId  = 0;

  sem_FirstEvent   =  1;   // keeps track of the first event in the whole sequence
  sem_Event  =  1;           // keeps track of the start of one event
  prev  =  0;            // pointer to keep track of the end of an event

  /* accum. timeDistance between events (think of the accumulated error)
  *  accumulated error in case that the event oscillates a greater amount of "gap"-times
  */ under the threshold.
  event_duration   = 50;

  StartEventCounter    = 0;
  EndEventCounter      = 0;

  GetSystemTime (&st);
  cout<<st.wYear<<st.wMonth<<st.wDay<<'\n';

  max101 = threshold101;

  cout<<"Give the name of the Sensor data input file"<<'\n';
  cin>>fileName;
  fileNameOut = "threshold.txt";
  cout<<"Wait ... calculating the threshold for sensor101 from the file. "<<'\n';

  fin.open(fileName.c_str());

  // open file
  while(fin.good() == false)  {
    cout << "Unable to open file: "<<fileName<<'\n';
    cout << "Enter a new name: ";
    cin >> fileName;
    fin.open(fileName.c_str());
  }

  fout.open(fileNameOut.c_str());

  if (fout.good() == false)  {
    cout << "Unable to open file: "<<fileNameOut<<'\n';
  }

  // the file can be opened
  if (fin.is_open()) {
    i = 0;

    while (!fin.eof() ) {
      getline (fin,line);
      if (i >= 2) {  // skip first 2 header lines from the file: i=0 first line, i=1 sec line
        if (!line.empty() ) {  // if the line from input file is not empty read sensor data
          cs101 = line.substr(24,11);  // read sensor101 string from a specific location
          cs102 = line.substr(35,11);  // read sensor102 string from a specific location
        } else {    // if line is empty make substr data empty
          cs101 = ""; //
          cs102 = ""; //
        }

        // if string value of sensors is not empty -> transform its value to float
        if (!cs101.empty()) {  // if the substr sensor data for sensor101 is not empty
          s101 = atof (cs101.c_str());   // convert substr to float value
          s102 = atof (cs102.c_str());   // convert substr to float value

          // check for new maxima
          if (s101 > max101)   // assign new values to the max for this sensor
              max101 = s101;
          if (s102 > max102)   // assign new values to the max for this sensor
              max102 = s102;

          // check for new minima
          if (s101 < min101)   // assign new values to the min for this sensor
              min101 = s101;
          if (s102 > min102)   // assign new values to the min for this sensor
              min102 = s102;

        } else {  // else: string value is empty -> assign low values to sensors
           s101 = -10;
           s102 = -10;
        }

        // the start of the new event
        if (sem_Event == 1) { // if we have detected a new event
            eventId++;
            StartEventCounter = i;  // mark beginning of event
            sem_Event = 0;
        }

        if (threshold101 < s101) {  // detect an event above the threshold
           prev++;

           if (!line.empty() ) {  // line from the file cannot be empty
              c_jaar = line.substr(0,4);  // read year string
              c_mnd  = line.substr(5,2); // read month string
              c_dag  = line.substr(8,2);  // read day string
              c_uur  = line.substr(11,2); // read hour string
              c_min  = line.substr(14,2); // read minutes string
              c_sec  = line.substr(17,2); // read sec. string
              c_msec = line.substr(20,3); // read msec. string
           } else {  // line empty, initialize the strings to empty
               c_jaar = ""; c_mnd = ""; c_dag = "";
               c_uur = ""; c_min  = ""; c_sec = ""; c_msec="";
           } // if line not empty


           if (!c_jaar.empty()) jaar = atoi (c_jaar.c_str()); else jaar = 0;
           if (!c_mnd.empty())  mnd  = atoi (c_mnd.c_str());  else mnd  = 0;
           if (!c_dag.empty())  dag  = atoi (c_dag.c_str());  else dag  = 0;

           if (!c_uur.empty())  uur  = atoi (c_uur.c_str());  else uur  = 0;
           if (!c_min.empty())  min  = atoi (c_min.c_str());  else min  = 0;
           if (!c_sec.empty())  sec  = atoi (c_sec.c_str());  else sec  = 0;
           if (!c_msec.empty()) msec = atoi (c_msec.c_str()); else msec  = 0;

           if (sem_FirstEvent == 1) {   // signal for the first event in the sequence
              prev = i;
              sem_FirstEvent = 0;
           } else { // prev != 0
                if (event_duration < (i-prev)) {  // check if event duration is long enough
                    sem_Event = 1;
                     prev = i;
                } // if (i-prev) > event_duration
           }
        } // sensor 101

      } // if i>2
      i++;
      total = total + s101;
    } // while !eof()

    if (i!=0) {
      threshold101 = total / i;  // calc. average for the threshold
    }
    cout<<'\n';
    cout <<"Rows = "<< i << " Sum = " <<total << " Threshold = " <<threshold101;
    cout << " Max = " <<max101<< "  min = " <<min101<< " "<<endl;
    cout<<'\n';

    if (fout.is_open() ) {
      fout << threshold101 << endl;
    }

    fin.close();
    fout.close();
    cout<<'\n';
    cout<<"The threshold " <<threshold101 <<" was written to file threshold.txt. "<<'\n';
    cout<<'\n';
  } // if file open
  else
    cout << "Unable to open file";

  return 0;
}
```

11 Appendix B: fileio.h

```
// ========================================================
// name: fileio.h
// date:  11-03-2010
// version 1.00
// This module will read the sensor flat data file and produce a CSV file based on a threshold.
// input the file you want to read, use first the option 1 to produce a threshold in the threshold file or
// input the threshold manually.
// ========================================================

#include <string>
#include <fstream>
#include <iostream>

#include <Windows.h>
#include <stdio.h>

using namespace std;



void makeCsvFile() {

  SYSTEMTIME st;


  char kol   = ';';
  char quote = '"';

  string fileName, fileNameOut, thresholdFileName;
  string line, threshold_line, c_jaar, c_mnd, c_dag, c_uur, c_min, c_sec, c_msec, cs101, cs102;
  string subline;

  ifstream fin, thresholdFile;
  ofstream fout;

  float threshold101, s101, threshold102, s102, max101, max102, min101, min102;;
  int jaar, mnd, dag, uur, min, sec, msec, tmp, i, event101, eventId, StartEvent, StartSeq, prev, gap;

  threshold101  = -10;
  event101 = 0;   // event affecting sensor 101
  eventId  = 0;

  StartSeq   = 1;   // keeps track of the first event in the whole sequence
  StartEvent = 1;   // keeps track of the start of one event
  prev = 0;         // pointer to keep track of the end of an event
  gap       = 50;    // max. timeDistance between events (think of the accumulated error)
  // accumulated error in case that the event oscillates an greater amount of "gap"-times
  // under the threshold.

  GetSystemTime (&st);
  cout<<st.wYear<<st.wMonth<<st.wDay<<'\n';

  cout<<"Input the threshold (use a value around zero) "<<'\n';
  cin>>threshold101;

  cout<<"Input the name of the Sensor data input file: "<<'\n';
  cin>>fileName;

  fileNameOut = fileName + ".csv";
  thresholdFileName = fileName + ".trh";

  fin.open(fileName.c_str());
  thresholdFile.open(thresholdFileName.c_str());

  // open input sensor datafile
  while(fin.good() == false)  {
    cout << "Unable to open file: "<<fileName<<'\n';
    cout << "Enter a new name: ";
    cin >> fileName;
    fin.open(fileName.c_str());
  }

  fout.open(fileNameOut.c_str());

  if (fout.good() == false)  {
    cout << "Unable to open file: "<< fileNameOut <<'\n';
  }

  if (thresholdFile.good() == false)  {  //open thresholdfile
    cout << "Unable to open file: "<< thresholdFileName <<'\n';
  } else {
    if (thresholdFile.is_open()) {
      getline (thresholdFile,threshold_line);
      if (! threshold_line.empty() ) {
        threshold101 = atof (threshold_line.c_str());
      }
    }
  }

  // the file can be opened
  if (fin.is_open()) {
    i = 0;
    while (! fin.eof() ) {
      getline (fin,line);
      if (i >= 2) { // skip first 2 header lines from the file
        if (!line.empty() ) {
          cs101 = line.substr(24,11);  // sensor 101
          cs102 = line.substr(35,11);  // sensor 102
        } else {
          cs101 = "";
          cs102 = "";
        }

        if (!cs101.empty()) {
          s101 = atof (cs101.c_str());  //
          s102 = atof (cs102.c_str());  //

        } else {
          s101 = -10;
          s102 = -10;
        }

        // signal the start of the new event StartEvent = TRUE
        if (StartEvent == 1) {
          eventId++;
          StartEvent = 0;
          max101 = threshold101;  // init. max for this new event
          min101 = threshold101;  // init. min for this new event
        }

        if (s101 > threshold101) {  // threshold for this sensor is value = 0
          prev++;

          if (!line.empty() ) {  // line from file cannot be empty
            c_jaar = line.substr(0,4);  // read strings
            c_mnd  = line.substr(5,2);
            c_dag  = line.substr(8,2);
            c_uur  = line.substr(11,2);
            c_min  = line.substr(14,2);
            c_sec  = line.substr(17,2);
            c_msec = line.substr(20,3);

          } else {  // line is empty -> nothing to read
             c_jaar = ""; c_mnd = ""; c_dag = ""; c_uur = ""; c_min  = ""; c_sec = ""; c_msec="";
          }

          if (!c_jaar.empty()) jaar = atoi (c_jaar.c_str()); else jaar = 0;
          if (!c_mnd.empty())  mnd  = atoi (c_mnd.c_str());  else mnd  = 0;
          if (!c_dag.empty())  dag  = atoi (c_dag.c_str());  else dag  = 0;

          if (!c_uur.empty())  uur  = atoi (c_uur.c_str());  else uur  = 0;
          if (!c_min.empty())  min  = atoi (c_min.c_str());  else min  = 0;
          if (!c_sec.empty())  sec  = atoi (c_sec.c_str());  else sec  = 0;
          if (!c_msec.empty()) msec = atoi (c_msec.c_str()); else msec  = 0;

          if (fout.is_open() ) {

            fout << eventId << kol;
            fout <<     i   << kol;
            fout << c_jaar  << "-" <<c_mnd << "-" <<c_dag << kol;
            fout << c_uur << ":" << c_min <<":" << c_sec  << kol;
            fout << c_msec << kol;
            fout << s101   << kol;
            fout << s102   << endl;

          } // if (s101 > threshold101)

          if (StartSeq == 1) {
            prev = i;
            StartSeq = 0;
          } else { // prev != 0
            if ((i-prev) > gap) {
              StartEvent = 1;
              prev = i;
            } // if (i-prev) > gap
          }

          cout << eventId << " " << i << " "<< prev <<" " << "sensor101 "<<c_jaar << "-" <<c_mnd << "-" <<c_dag << " " <<c_uur << ":" << c_min <<":" << c_sec <<"," << c_ms

        } // s101 > threshold101

      } // if i>=2
      i++;

    } // while !eof()

    fin.close();
    fout.close();
  } // if file open

  else cout << "Unable to open file";

}
```

12 Appendix C: listImplStack.h

```
//==========================================
//author: Gratian Schiopu
//Single linked list implementation of a stack
// this ADS (abstract data structure) will be used for sampling with a fixed window size
//==========================================

#ifndef _LISTIMPLSTACK_
#define _LISTIMPLSTACK_
#include <iostream>
using namespace std;

const int MAX_STACK = 10;

typedef float stackItemType;

class StackClass {
public:
  StackClass();
  // stackClass (const stackClass & S);  // copy constructor
  ~StackClass();

  bool stackIsEmpty();  // check if the stack is empty
  bool stackIsFull();  // check if the stack is full
  void push (stackItemType newItem);  // push one elem. on the stack
  void pop();                         // pop one elem from the stack
  void getStackTop(stackItemType & topItem);  //get the top element
  void displayStackElem ();
  float averageElem (); // perform the average of all the elem. on the stack
  void emptyStack ();   // delete all elem. from the stack

private:
// typedef node *nodePrt;

struct node {
  stackItemType item;
  node *next;
} *bottom, *top;

  int totalElements;
};


// stack constructor
// PRE: the stack doesn't exist.
// POST: the stack is created as an empty stack
StackClass::StackClass() {bottom = top = NULL; totalElements = 0;}

// stack destructor
// PRE: the stack exists as a simply linked list
// POST: each node from the stack is poped from the stack and the memory deallocated
StackClass::~StackClass() {
}


// PRE: the stack exists
// POST: if the stack is empty the method returns a TRUE, else a FALSE.
bool StackClass::stackIsEmpty() {return bool (top == NULL);}

// PRE: the stack exists
// POST: if the stack is empty the method returns a TRUE, else a FALSE.
bool StackClass::stackIsFull() {return bool (totalElements == (MAX_STACK-1));}


// PRE: the stack exists and the Maxsize of the stack isn't reached
// POST: the new element is pused onto the stack
// and the top of stack points to the new added element.
void StackClass::push(stackItemType newItem) {
 if (!stackIsFull()){
  if (stackIsEmpty()) {
     bottom = top = new node;
     top->item = newItem;
     top->next = NULL;
  } else {
    top->next = new node;
    top = top->next;
    top->item = newItem;
    top->next = NULL;
  }
  totalElements++;
 }
}


// PRE: the stack exists
// POST: the top element is deleted from the stack and
// top of the stack points to the previous element added to the stack.
void StackClass::pop(){
node *p;

 if (bottom == top) { // only one item on the stack
  bottom = NULL;
  delete top;
  top = NULL;
  totalElements = 0;
 } else { // more than one items on the stack
  p = bottom;
  while ((p->next) != top) {p = p->next; }
    p->next = NULL;
    delete top;
    top = p;
    totalElements--;
  } // fi
}


// PRE: the stack isn't empty
// POST: the element pointed to by the top is returned.
void StackClass::getStackTop(stackItemType& topItem){
 if (!stackIsEmpty()){
   topItem = top->item;
 }
}

// display the elemnets of the stack: from bottom to the top
void StackClass::displayStackElem() {
node *p;
stackItemType item;

 if (!stackIsEmpty()) {
   cout << "bottom of stack stack: ";
   cout << bottom->item;
   cout << "\n";

   cout << "top of the stack: ";
   cout << top->item;
   cout << "\n";

   p = bottom;

   while (p != top) {
    item = p->item;
    cout << item;
    p = p->next;
   }
   item = p->item;
   cout << item;
   cout << "\n";
   cout << "Total: " << totalElements << "\n";
 } else { cout << "EmptyStack! \n";}
}


// average the value of the elements on the stack, return the avg value
// the stack will be empty after the average operation.
float StackClass::averageElem() {
 node *p;
 stackItemType item;
 int counter = 0;
 float total = 0;

 while (!stackIsEmpty()) {
   getStackTop (item);
   total = total + item;
   pop();
   counter++;
 }

 if (counter != 0)
   return (total / counter);
 else
   return 0;
}

// delete all elem. from the stack
void StackClass::emptyStack() {
 while (!stackIsEmpty()) { pop(); }
}
#endif
```

13 Appendix D: sampling.h

```
//==================================================
// datum 19-08-2010
// version 1.00
// sampling using a stack
// elements are pushed on the stack
// if the stack is full we average the elements on the stack.
// We do this by using each time a pop until the stack is empty.
// After each pop we sum up the elements and in the end we compute the average
//==================================================
#include <string>
#include <fstream>
#include <iostream>

#include <Windows.h>
#include <stdio.h>

#include "listImplStack.h"

using namespace std;

void Sampling2Csv() {

 stackItemType stItem;
 StackClass S;
 int stCounter;

 SYSTEMTIME st;
 char kol   = ';';
 char quote = '"';

 string fileName, fileNameOut, thresholdFileName;
 string line, threshold_line, c_jaar, c_mnd, c_dag, c_uur, c_min, c_sec, c_msec, cs101, cs102;
 string subline;

 ifstream fin, thresholdFile;
 ofstream fout;

 float threshold101, s101, threshold102, s102, max101, max102, min101, min102;;
 int jaar, mnd, dag, uur, min, sec, msec, tmp, i, event101, eventId, StartEvent, StartSeq, prev, gap;

 threshold101  = -10;
 event101 = 0;   // event affecting sensor 101
 eventId  = 0;

 StartSeq   = 1;   // keeps track of the first event in the whole sequence
 StartEvent = 1;   // keeps track of the start of one event
 prev = 0;         // pointer to keep track of the end of an event
 gap       = 50;    // max. timeDistance between events (think of the accumulated error)
 // accumulated error in case that the event oscillates an greater amount of "gap"-times
 // under the threshold.

 GetSystemTime (&st);
 cout<<st.wYear<<st.wMonth<<st.wDay<<'\n';

 cout<<"Input the threshold (use a value around zero) "<<'\n';
 cin>>threshold101;

 cout<<"Input the name of the Sensor data input file: "<<'\n';
 cin>>fileName;

 fileNameOut = fileName + ".csv";
 thresholdFileName = fileName + ".trh";

 fin.open(fileName.c_str());
 thresholdFile.open(thresholdFileName.c_str());

 // open input sensor datafile
 while(fin.good() == false)  {
  cout << "Unable to open file: "<<fileName<<'\n';
  cout << "Enter a new name: ";
  cin >> fileName;
  fin.open(fileName.c_str());
 }

 fout.open(fileNameOut.c_str());

 if (fout.good() == false)  {
  cout << "Unable to open file: "<< fileNameOut <<'\n';
 }

 if (thresholdFile.good() == false)  {
  cout << "Unable to open file: "<< thresholdFileName <<'\n';
 } else {
  if (thresholdFile.is_open()) {
    getline (thresholdFile,threshold_line);
    if (! threshold_line.empty() ) {
     threshold101 = atof (threshold_line.c_str());
    }
  }
 }

 // the file can be opened
 if (fin.is_open()) {
  i = 0;
  while (! fin.eof() ) {
    getline (fin,line);
    if (i >= 2) { // skip first 2 header lines from the file
      if (!line.empty() ) {
        cs101 = line.substr(24,11);  // sensor 101
        cs102 = line.substr(35,11);  // sensor 102
      } else {
        cs101 = "";
        cs102 = "";
      }

      if (!cs101.empty()) {
        s101 = atof (cs101.c_str());  //
        s102 = atof (cs102.c_str());  //
      } else {
        s101 = -100;
        s102 = -100;
      }

      // signal the start of the new event StartEvent = TRUE
      if (StartEvent == 1) {
        eventId++;
        StartEvent = 0;
        max101 = threshold101;  // init. max for this new event
        min101 = threshold101;  // init. min for this new event
      }

      if (!S.stackIsFull()) { // if stack not full
        S.push(s101);  // push value on the stack
      } else { // stack is full
        stItem = S.averageElem(); // avg elem. from the stack; empty the stack

        if (stItem >= threshold101) {  // threshold for this sensor is value = 0
         if (!line.empty() ) {  // line from file cannot be empty
           c_jaar = line.substr(0,4);  // read strings
           c_mnd  = line.substr(5,2);
           c_dag  = line.substr(8,2);
           c_uur  = line.substr(11,2);
           c_min  = line.substr(14,2);
           c_sec  = line.substr(17,2);
           c_msec = line.substr(20,3);
         } else {  // line is empty -> nothing to read
           c_jaar = ""; c_mnd = ""; c_dag = ""; c_uur = ""; c_min  = ""; c_sec = ""; c_msec="";
         }

         if (!c_jaar.empty()) jaar = atoi (c_jaar.c_str()); else jaar = 0;
         if (!c_mnd.empty())  mnd  = atoi (c_mnd.c_str());  else mnd  = 0;
         if (!c_dag.empty())  dag  = atoi (c_dag.c_str());  else dag  = 0;

         if (!c_uur.empty())  uur  = atoi (c_uur.c_str());  else uur  = 0;
         if (!c_min.empty())  min  = atoi (c_min.c_str());  else min  = 0;
         if (!c_sec.empty())  sec  = atoi (c_sec.c_str());  else sec  = 0;
         if (!c_msec.empty()) msec = atoi (c_msec.c_str()); else msec  = 0;

         if (fout.is_open() ) {
           fout << eventId << kol;
           fout <<     i   << kol;
           fout << c_jaar  << "-" <<c_mnd << "-" <<c_dag << kol;
           fout << c_uur << ":" << c_min <<":" << c_sec  << kol;
           fout << c_msec << kol;
           fout << stItem << endl;
//          fout << stItem   << kol;
//          fout << s102     << endl;
         } // if output file is open

         prev = prev + 10;  // advance prev. with the stacksize

         if (StartSeq == 1) {
           prev = i;
           StartSeq = 0;
         } else { // prev != 0
           if ((i-prev) > gap) {
            StartEvent = 1;
            prev = i;
           } // if (i-prev) > gap
         }
        } // stItem >= threshold101
      } // stackIsFull()
    } // if i>2
    i++;

  } // while ! eof()

  fin.close();
  fout.close();
 } // if file open
 else cout << "Unable to open file";
}
```

14 Appendix E: fileioxml.h

```
//============================================
// name: fileioxml.h
// This program filters events from sensor a data file given a specific threshold
// datum 27-06-2010
// output file format is XML
//============================================
#include <string>
#include <fstream>
#include <iostream>


#include <Windows.h>
#include <stdio.h>

using namespace std;


int makeXmlFile() {

  SYSTEMTIME st;

  string fileName, fileNameOut;
  string line, c_jaar, c_mnd, c_dag, c_uur, c_min, c_sec, c_msec, cs101, cs102;
  string subline;

  ifstream fin;
  ofstream fout;

  float threshold101, s101, threshold102, s102, max101, max102, min101, min102;
  int jaar, mnd, dag, uur, min, sec, msec, tmp, i, event101, eventId, sem_Event, sem_FirstEvent, prev, event_duration;
  int StartEventCounter, EndEventCounter;

  threshold101  = -3;
  event101 = 0;   // event affecting sensor 101
  eventId  = 0;

  sem_FirstEvent   =  1;   // keeps track of the first event in the whole sequence
  sem_Event  =  1;   // keeps track of the start of one event
  prev             =  0;   // pointer to keep track of the end of an event
  event_duration   = 50;   // ax. timeDistance between events (think of the accumulated error)
  // accumulated error in case that the event oscillates an greater amount of "gap"-times
  // under the threshold.
  StartEventCounter    = 0;
  EndEventCounter      = 0;


  GetSystemTime (&st);
  cout<<st.wYear<<st.wMonth<<st.wDay<<'\n';

  cout<<"Input the threshold (use a value around zero) "<<'\n';
  cin>>threshold101;

  max101 = threshold101;
  //max102 = threshold102;

  cout<<"Give the name of the Sensor data input file"<<'\n';
  cin>>fileName;
  fileNameOut = fileName + ".xml";

  fin.open(fileName.c_str());

  // open file
  while(fin.good() == false)  {
    cout << "Unable to open file: "<<fileName<<'\n';
    cout << "Enter a new name: ";
    cin >> fileName;
    fin.open(fileName.c_str());
  }

  fout.open(fileNameOut.c_str());

  if (fout.good() == false)  {
    cout << "Unable to open file: "<<fileNameOut<<'\n';
  }

  // the file can be opened
  if (fin.is_open()) {
    i = 0;

    while (!fin.eof() ) {
      getline (fin,line);
      if (i < 2) {
        if (i < 1) {
          if (fout.is_open() ) {
            fout << "<bridge>" << endl;
            fout << " " <<"bridgeID   = " << '"' << "001" << '"' << endl;
            fout << " "<<"bridgeName = "  << '"' << "Hollandsebrug" << '"' << endl;
            fout << " "<<"Sensor threshold  = "  << '"' << threshold101 << '"' << endl;
            fout << " "<<"Leading sensor = "  << '"' << "101" << '"' << endl<< endl;

          }
        }
      } else { // skip first 2 header lines from the file
        if (!line.empty() ) {  // if the line from input file is not empty read sensor data
```

```
          cs101 = line.substr(24,11);  // read sensor101 string from a specific location
          cs102 = line.substr(35,11);  // read sensor102 string from a specific location
        } else {    // if line is empty make substr data empty
            cs101 = ""; //
            cs102 = ""; //
        }

        // if string value of sensors is not empty -> transform its value to float
        if (!cs101.empty()) {  // if the substr sensor data for sensor101 is not empty
          s101 = atof (cs101.c_str());   // convert substr to float value
          s102 = atof (cs102.c_str());   // convert substr to float value

          // check for new maxima
          if (s101 > max101)   // assign new values to the max for this sensor
            max101 = s101;
          if (s102 > max102)   // assign new values to the max for this sensor
            max102 = s102;

          // check for new minima
          if (s101 < min101)   // assign new values to the min for this sensor
            min101 = s101;
          if (s102 > min102)   // assign new values to the min for this sensor
            min102 = s102;

          } else {  // else assign low values to sensors
            s101 = -10;
            s102 = -10;
          }

          // the start of the new event
          if (sem_Event == 1) { // if we have detected a new event
            eventId++;
            StartEventCounter = i;  // mark beginning of event
            max101 = threshold101;  // init. max for this new event
            min101 = threshold101;  // init. min for this new event
            sem_Event = 0;

            if (fout.is_open() ) {
              fout << " <event>" << endl;
              fout << "  " <<"eventID   = " << eventId << endl;
            }
          }

          if (threshold101 < s101) {  // detect an event above a certain threshold
            prev++;

            if (!line.empty() ) {  // line from the file cannot be empty
              c_jaar = line.substr(0,4);  // read strings
              c_mnd  = line.substr(5,2);
              c_dag  = line.substr(8,2);
              c_uur  = line.substr(11,2);
              c_min  = line.substr(14,2);
              c_sec  = line.substr(17,2);
              c_msec = line.substr(20,3);

            } else {  // line is empty -> nothing to read
              c_jaar = ""; c_mnd = ""; c_dag = ""; c_uur = ""; c_min  = ""; c_sec = ""; c_msec="";
            }

            if (!c_jaar.empty()) jaar = atoi (c_jaar.c_str()); else jaar = 0;
            if (!c_mnd.empty())  mnd  = atoi (c_mnd.c_str());  else mnd  = 0;
            if (!c_dag.empty())  dag  = atoi (c_dag.c_str());  else dag  = 0;

            if (!c_uur.empty())  uur  = atoi (c_uur.c_str());  else uur  = 0;
            if (!c_min.empty())  min  = atoi (c_min.c_str());  else min  = 0;
            if (!c_sec.empty())  sec  = atoi (c_sec.c_str());  else sec  = 0;
            if (!c_msec.empty()) msec = atoi (c_msec.c_str()); else msec  = 0;

            if (fout.is_open() ) {
              fout << "   <sensor>" << endl;
              fout << "     sensorID   = " << '"' << "101" << '"' << endl;
              fout << "     sensorType = " << '"' << "Load" << '"' << endl;
              fout << "     dateTime   = " << '"' << c_jaar << "-" << c_mnd << "-" << c_dag << " " << c_uur << ":" << c_min << ":" << c_sec << "," << c_msec << '"' << endl
              fout << "     value      = " << cs101 << endl;
              fout << "     rowCounter = " << i << endl;
              fout << "     prevCounter= " << prev << endl;
              fout << "   </sensor>" << endl<< endl;
//-----------------------------------------------------------------------------------------
//              fout << "   <sensor>" << endl;
//              fout << "     sensorID   = " << '"' << "102" << '"' << endl;
//              fout << "     sensorType = " << '"' << "Load" << '"' << endl;
//              fout << "     dateTime   = " << '"' << c_jaar << "-" << c_mnd << "-" << c_dag << " " << c_uur << ":" << c_min << ":" << c_sec << "," << c_msec << '"' << en
//              fout << "     value      = " << cs102 << endl;
//              fout << "     rowCounter = " << i << endl;
//              fout << "     prevCounter= " << prev << endl;
//              fout << "   </sensor>" << endl<< endl;
//-----------------------------------------------------------------------------------------

          } // if (s101 > threshold101)
          if (sem_FirstEvent == 1) {   // signal for the first event in the sequence
            prev = i;
            sem_FirstEvent = 0;
          } else { // prev != 0
            if (event_duration < (i-prev)) {  // chech if event duration is long enough
              fout << " MaxValue on Sensor101 for event " <<eventId <<" = " << max101 << endl;
              fout << " MinValue on Sensor101 for event " <<eventId <<" = " << min101 << endl;
              fout << " </event>" << endl << endl;
              cout << "EventId = "<<eventId << " " << " min = "<< min101 << "   Max = "<< max101 << endl;

              sem_Event = 1;
              prev = i;
            } // if (i-prev) > event_duration
          }

          fout << i << " "<< prev <<" " << "sensor101 "<<c_jaar << "-" <<c_mnd << "-" <<c_dag << " " <<c_uur << ":" << c_min <<":" << c_sec <<"," << c_msec << " " << s101

        } // sensor 101

      } // if i>2
      i++;
//cout << i << " " <<jaar << " " << mnd << " " << dag << " " << uur << " " << min <<" " << sec <<" " << s101 <<endl;
//cout << i << " " <<c_jaar << " " <<c_mnd << " " <<c_dag << " " <<c_uur << " " << c_min <<" " << c_sec <<"," << c_msec << " " << s101 <<endl;

    } // while !eof()

    if (fout.is_open() ) {
      fout << " </event>" << endl;
      fout << "</bridge>" << eventId <<endl;
    }

    fin.close();
    fout.close();
  } // if file open
  else
    cout << "Unable to open file";

  return 0;
}
```

15 Appendix F: socketxml.h

```
//=========================================
// name: socketxml.h
// author Gratian Schiopu
// date: 27-06-2010
// for this module we need the server.exe running at the given IP addres
// this module reads a sensory flat file it will transmit the data to a server in XML
//=========================================




#include <string>
#include <fstream>
#include <iostream>


#include <Windows.h>
#include <stdio.h>
// #include <winsock2.h>

using namespace std;
// using namespace System;



int sockXml() {

  SYSTEMTIME st;
  WSADATA wsaData;    // Initialize Winsock
  SOCKET m_socket;    // socket ADtype


  string fileName, fileNameOut;
  string line, c_jaar, c_mnd, c_dag, c_uur, c_min, c_sec, c_msec, cs101, cs102;
  string subline;
  // Be careful with the array bound, provide some checking mechanism...
  char cIPaddr[15];  // IPaddr for the server the client will communicate with
  char sendbuf[200] = "This is a test string from client";   // send buffer
  char recvbuf[200] = "";                                     // receive buffer


  ifstream fin;
  ofstream fout;

  float threshold101, s101, threshold102, s102, max101, max102, min101, min102;
  int jaar, mnd, dag, uur, min, sec, msec, tmp, i, event101, eventId, sem_Event, sem_FirstEvent, prev, event_duration;
  int StartEventCounter, EndEventCounter;
  int portNo;   // socket port number;
  int bytesSent;
  int bytesRecv = SOCKET_ERROR;

  threshold101  = -3;
  event101 = 0;   // event affecting sensor 101
  eventId  = 0;

  sem_FirstEvent   =  1;   // keeps track of the first event in the whole sequence
  sem_Event  =  1;   // keeps track of the start of one event
  prev             =  0;   // pointer to keep track of the end of an event
  event_duration   = 50;   // ax. timeDistance between events (think of the accumulated error)
  // accumulated error in case that the event oscillates a greater amount of "gap"-times
  // under the threshold.

  StartEventCounter    = 0;
  EndEventCounter      = 0;

  int iResult = WSAStartup(MAKEWORD(2,2), &wsaData);  // check result of the WSAtartup
  if (iResult != NO_ERROR)
    printf("Client: Error at WSAStartup().\n");
  else
    printf("Client: WSAStartup() is OK.\n");

  // Create a SOCKET for connecting to a server
  m_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (m_socket == INVALID_SOCKET)   {
    printf("Client: Error at socket(): %ld\n", WSAGetLastError());
    WSACleanup();
    return 0;
  }
  else
    printf("Client: socket() is OK.\n");

 cout<<"Input the IPaddres for the Server "<<'\n';
 cin>>cIPaddr;
  cout<<"The Server IP Addres is "<<cIPaddr<<'\n';

  cout<<"Input the IPaddres for the port "<<'\n';
  cin>>portNo;
  cout<<"The Port for the Socket communication is: "<<portNo<<'\n';

  // The sockaddr_in structure specifies the address family,
  // IP address, and port of the server to be connected to.
  sockaddr_in clientService;
  clientService.sin_family = AF_INET;
  // clientService.sin_addr.s_addr = inet_addr("127.0.0.1");
  clientService.sin_addr.s_addr = inet_addr(cIPaddr);
  clientService.sin_port = htons(portNo);

  // Connect to server...
  if (connect(m_socket, (SOCKADDR*)&clientService, sizeof(clientService)) == SOCKET_ERROR) {
      printf("Client: Failed to connect.\n");
      WSACleanup();
      return 0;
    } else {
        printf("Client: connect() is OK.\n");
        printf("Client: Can start sending and receiving data....\n");
    }

   // will use this later on when we read more files based on date.counter
   // generated files: 2010629.000..2010629.287
   GetSystemTime (&st);
   cout<<st.wYear<<st.wMonth<<st.wDay<<'\n';

   cout<<"Input the threshold (use a value around zero) "<<'\n';
   cin>>threshold101;

   max101 = threshold101;
// max102 = threshold102;

   cout<<"Give the name of the Sensor data input file"<<'\n';
   cin>>fileName;
   fileNameOut = fileName + ".xml";

  fin.open(fileName.c_str());

  // open file
  while(fin.good() == false)  {
    cout << "Unable to open file: "<<fileName<<'\n';
    cout << "Enter a new name: ";
    cin >> fileName;
    fin.open(fileName.c_str());
  }

  fout.open(fileNameOut.c_str());

  if (fout.good() == false)  {
    cout << "Unable to open file: "<<fileNameOut<<'\n';
  }

  // the file can be opened
  if (fin.is_open()) {
    i = 0;

    while (!fin.eof() ) {
      getline (fin,line);
      if (i < 2) {
        if (i < 1) {
          if (fout.is_open() ) {
            fout << "<bridge>" << endl;
            fout << " " <<"bridgeID   = " << '"' << "001" << '"' << endl;
            fout << " "<<"bridgeName = "  << '"' << "Hollandsebrug" << '"' << endl;
            fout << " "<<"Sensor threshold  = "  << '"' << threshold101 << '"' << endl;
            fout << " "<<"Leading sensor = "  << '"' << "101" << '"' << endl<< endl;

          }
        }
      } else { // skip first 2 header lines from the file
         if (!line.empty() ) {  // if the line from input file is not empty read sensor data
            cs101 = line.substr(24,11);  // read sensor101 string from a specific location
            cs102 = line.substr(35,11);  // read sensor102 string from a specific location
         } else {    // if line is empty make substr data empty
            cs101 = ""; //
            cs102 = ""; //
         }

         // if string value of sensors is not empty -> transform its value to float
         if (!cs101.empty()) {  // if the substr sensor data for sensor101 is not empty
           s101 = atof (cs101.c_str());   // convert substr to float value
           s102 = atof (cs102.c_str());   // convert substr to float value

           // check for new maxima
           if (s101 > max101)   // assign new values to the max for this sensor
             max101 = s101;
           if (s102 > max102)   // assign new values to the max for this sensor
             max102 = s102;

           // check for new minima
           if (s101 < min101)   // assign new values to the min for this sensor
             min101 = s101;
           if (s102 > min102)   // assign new values to the min for this sensor
             min102 = s102;
         } else {  // else assign low values to sensors
           s101 = -10;
           s102 = -10;
         }

         // the start of the new event
         if (sem_Event == 1) { // if we have detected a new event
           eventId++;
           StartEventCounter = i;  // mark beginning of event
           max101 = threshold101;  // init. max for this new event
           min101 = threshold101;  // init. min for this new event
           sem_Event = 0;

           if (fout.is_open() ) {
             fout << " <event>" << endl;
             fout << "  " <<"eventID   = " << eventId << endl;
           }
         }

         if (threshold101 < s101) {  // detect an event above a certain threshold
         prev++;

         if (!line.empty() ) {  // line from the file cannot be empty
           c_jaar = line.substr(0,4);  // read strings
           c_mnd  = line.substr(5,2);
           c_dag  = line.substr(8,2);
           c_uur  = line.substr(11,2);
           c_min  = line.substr(14,2);
           c_sec  = line.substr(17,2);
           c_msec = line.substr(20,3);

         } else {  // line is empty -> nothing to read
           c_jaar = ""; c_mnd = ""; c_dag = ""; c_uur = ""; c_min  = ""; c_sec = ""; c_msec="";
         }

         if (!c_jaar.empty()) jaar = atoi (c_jaar.c_str()); else jaar = 0;
         if (!c_mnd.empty())  mnd  = atoi (c_mnd.c_str());  else mnd  = 0;
         if (!c_dag.empty())  dag  = atoi (c_dag.c_str());  else dag  = 0;

         if (!c_uur.empty())  uur  = atoi (c_uur.c_str());  else uur  = 0;
         if (!c_min.empty())  min  = atoi (c_min.c_str());  else min  = 0;
         if (!c_sec.empty())  sec  = atoi (c_sec.c_str());  else sec  = 0;
         if (!c_msec.empty()) msec = atoi (c_msec.c_str()); else msec  = 0;

         if (fout.is_open() ) {
           fout << "   <sensor>" << endl;
           fout << "     sensorID   = " << '"' << "101" << '"' << endl;
           fout << "     sensorType = " << '"' << "Load" << '"' << endl;
           fout << "     dateTime   = " << '"' << c_jaar << "-" << c_mnd << "-" << c_dag << " " << c_uur << ":" << c_min << ":" << c_sec << "," << c_msec << '"' << endl;
           fout << "     value      = " << cs101 << endl;
           fout << "     rowCounter = " << i << endl;
           fout << "     prevCounter= " << prev << endl;
           fout << "   </sensor>" << endl<< endl;
//---------------------------------------------------------------------------------------
//           fout << "   <sensor>" << endl;
//           fout << "     sensorID   = " << '"' << "102" << '"' << endl;
//           fout << "     sensorType = " << '"' << "Load" << '"' << endl;
//           fout << "     dateTime   = " << '"' << c_jaar << "-" << c_mnd << "-" << c_dag << " " << c_uur << ":" << c_min << ":" << c_sec << "," << c_msec << '"' << endl;
//           fout << "     value      = " << cs102 << endl;
//           fout << "     rowCounter = " << i << endl;
//           fout << "     prevCounter= " << prev << endl;
//fout << "   </sensor>" << endl<< endl;
//---------------------------------------------------------------------------------------


           // Sends data to server...
           bytesSent = send (m_socket, sendbuf, strlen(sendbuf), 0);
           if(bytesSent == SOCKET_ERROR)
               printf("Client: send() error %ld. \n", WSAGetLastError());
           else {
                  printf("Client: send() is OK - Bytes sent: %ld \n", bytesSent);
                  printf("Client: The test string sent: \"%s\" \n", sendbuf);
           }

         } // if (s101 > threshold101)
         if (sem_FirstEvent == 1) {   // signal for the first event in the sequence
           prev = i;
           sem_FirstEvent = 0;
         } else { // prev != 0
          if (event_duration < (i-prev)) {  // chech if event duration is long enough
            fout << " MaxValue on Sensor101 for event " <<eventId <<" = " << max101 << endl;
            fout << " MinValue on Sensor101 for event " <<eventId <<" = " << min101 << endl;
            fout << " </event>" << endl << endl;

            cout << "EventId = "<<eventId << " " << " min = "<< min101 << "   Max = "<< max101 << endl;

            sem_Event = 1;
            prev = i;
          } // if (i-prev) > event_duration
         }

         cout << i << " "<< prev <<" " << "sensor101 "<<c_jaar << "-" <<c_mnd << "-" <<c_dag << " " <<c_uur << ":" << c_min <<":" << c_sec <<"," << c_msec << " " << s101 <

       } // sensor 101

      } // if i>2
      i++;

//   cout << i << " " <<jaar << " " << mnd << " " << dag << " " << uur << " " << min <<" " << sec <<" " << s101 <<endl;
//   cout << i << " " <<c_jaar << " " <<c_mnd << " " <<c_dag << " " <<c_uur << " " << c_min <<" " << c_sec <<"," << c_msec << " " << s101 <<endl;

    } // while !eof()

    if (fout.is_open() ) {
      fout << " </event>" << endl;
      fout << "</bridge>" << eventId <<endl;
    }
    fin.close();
    fout.close();
  } // if file open
  else
    cout << "Unable to open file";

  WSACleanup();
   return 0;

}
```

16 Appendix G: socketcsv.h

```
//==========================================
// name: socketcsv.h
// author Gratian Schiopu
// date: 27-06-2010
// for this module we need the server.exe running at the given IP addres
// this module will transmit the data filtered by a threshold to a server (in CSV format)
//==========================================

#include <string>
#include <fstream>
#include <iostream>

#include <Windows.h>
#include <stdio.h>

using namespace std;

const int MAX_COM = 12;  // max. number of

int sockCsv() {

  SYSTEMTIME st;
  WSADATA wsaData;    // Initialize Winsock
  SOCKET m_socket;    // socket ADtype

  char kol   = ';';
  char quote = '"';

  string fileName, fileNameOut, thresholdFileName;
  string line, threshold_line, c_jaar, c_mnd, c_dag, c_uur, c_min, c_sec, c_msec, cs101, cs102;
  string subline;
  // Be careful with the array bound, provide some checking mechanism...
  char cIPaddr[15];  // IPaddr for the server the client will communicate with
  char sendbuf[12] = "From client";   // send buffer
  char recvbuf[12] = "";              // receive buffer

  ifstream fin, thresholdFile;
  ofstream fout;

  float threshold101, s101, threshold102, s102, max101, max102, min101, min102;;
  int jaar, mnd, dag, uur, min, sec, msec, tmp, i, event101, eventId, StartEvent, StartSeq, prev, gap;
  int portNo;    // socket port number;
  int bytesSent; // bytes sent couter
  int bytesRecv = SOCKET_ERROR;


  threshold101  = -10;
  event101 = 0;   // event affecting sensor 101
  eventId  = 0;

  StartSeq   = 1;   // keeps track of the first event in the whole sequence
  StartEvent = 1;   // keeps track of the start of one event
  prev = 0;         // pointer to keep track of the end of an event
  gap       = 50;    // max. timeDistance between events (think of the accumulated error)
  // accumulated error in case that the event oscillates an greater amount of "gap"-times
  // under the threshold.

  int iResult = WSAStartup(MAKEWORD(2,2), &wsaData);  // check result of the WSAtartup
  if (iResult != NO_ERROR)
     printf("Client: Error at WSAStartup().\n");
  else
     printf("Client: WSAStartup() is OK.\n");

   // Create a SOCKET for connecting to a server
   m_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   if (m_socket == INVALID_SOCKET)   {
      printf("Client: Error at socket(): %ld\n", WSAGetLastError());
      WSACleanup();
      return 0;
   }
    else
      printf("Client: socket() is OK.\n");

  cout<<"Input the IPaddres for the Server "<<'\n';
  cin>>cIPaddr;
  cout<<"The Server IP Addres is "<<cIPaddr<<'\n';

  cout<<"Input the IPaddres for the port "<<'\n';
  cin>>portNo;
  cout<<"The Port for the Socket communication is: "<<portNo<<'\n';

  // The sockaddr_in structure specifies the address family,
  // IP address, and port of the server to be connected to.
  sockaddr_in clientService;
  clientService.sin_family = AF_INET;

  // clientService.sin_addr.s_addr = inet_addr("127.0.0.1");
  clientService.sin_addr.s_addr = inet_addr(cIPaddr);
  clientService.sin_port = htons(portNo);

  // Connect to server...
  if (connect(m_socket, (SOCKADDR*)&clientService, sizeof(clientService)) == SOCKET_ERROR) {
      printf("Client: Failed to connect.\n");
      WSACleanup();
      return 0;
  } else {
      printf("Client: connect() is OK.\n");
  printf("Client: Can start sending and receiving data....\n");
  }

  // will use this later on when we read more files based on date.counter
  // generated files: 2010629.000..2010629.287
  GetSystemTime (&st);
  cout<<st.wYear<<st.wMonth<<st.wDay<<'\n';

  cout<<"Input the threshold (use a value around zero) "<<'\n';
  cin>>threshold101;

  cout<<"Input the name of the Sensor data input file: "<<'\n';
  cin>>fileName;

  fileNameOut = fileName + ".csv";
  thresholdFileName = fileName + ".trh";

  fin.open(fileName.c_str());
  thresholdFile.open(thresholdFileName.c_str());

  // open input sensor datafile
  while(fin.good() == false)  {
    cout << "Unable to open file: "<<fileName<<'\n';
    cout << "Enter a new name: ";
    cin >> fileName;
    fin.open(fileName.c_str());
  }

  fout.open(fileNameOut.c_str());

  if (fout.good() == false)  {
    cout << "Unable to open file: "<< fileNameOut <<'\n';
  }
  if (thresholdFile.good() == false)  {
   cout << "Unable to open file: "<< thresholdFileName <<'\n';
  } else {
    if (thresholdFile.is_open()) {
      getline (thresholdFile,threshold_line);
      if (! threshold_line.empty() ) {
        threshold101 = atof (threshold_line.c_str());
      }
    }
  }

  // the file can be opened
  if (fin.is_open()) {
    i = 0;

    while (! fin.eof() ) {
      getline (fin,line);
      if (i >= 2) { // skip first 2 header lines from the file
        if (!line.empty() ) {
          cs101 = line.substr(24,11);  // sensor 101
          cs102 = line.substr(35,11);  // sensor 102
        } else {
          cs101 = "";
          cs102 = "";
        }

        if (!cs101.empty()) {
          s101 = atof (cs101.c_str());  //
          s102 = atof (cs102.c_str());  //
        } else {
          s101 = -10;
          s102 = -10;
        }

        // signal the start of the new event StartEvent = TRUE
        if (StartEvent == 1) {
          eventId++;
          StartEvent = 0;
          max101 = threshold101;  // init. max for this new event
          min101 = threshold101;  // init. min for this new event
        }

        if (s101 > threshold101) {  // threshold for this sensor is value = 0
          prev++;

          if (!line.empty() ) {  // line from file cannot be empty
            c_jaar = line.substr(0,4);  // read strings
            c_mnd  = line.substr(5,2);
            c_dag  = line.substr(8,2);
            c_uur  = line.substr(11,2);
            c_min  = line.substr(14,2);
            c_sec  = line.substr(17,2);
            c_msec = line.substr(20,3);

          } else {  // line is empty -> nothing to read
            c_jaar = ""; c_mnd = ""; c_dag = ""; c_uur = ""; c_min  = ""; c_sec = ""; c_msec="";
          }

          if (!c_jaar.empty()) jaar = atoi (c_jaar.c_str()); else jaar = 0;
          if (!c_mnd.empty())  mnd  = atoi (c_mnd.c_str());  else mnd  = 0;
          if (!c_dag.empty())  dag  = atoi (c_dag.c_str());  else dag  = 0;
```

```
          if (!c_uur.empty())  uur  = atoi (c_uur.c_str());  else uur  = 0;
          if (!c_min.empty())  min  = atoi (c_min.c_str());  else min  = 0;
          if (!c_sec.empty())  sec  = atoi (c_sec.c_str());  else sec  = 0;
          if (!c_msec.empty()) msec = atoi (c_msec.c_str()); else msec  = 0;

          if (fout.is_open() ) {
            fout << eventId << kol;
            fout <<     i   << kol;
            fout << c_jaar  << "-" <<c_mnd << "-" <<c_dag << kol;
            fout << c_uur << ":" << c_min <<":" << c_sec  << kol;
            fout << c_msec << kol;
            fout << s101   << kol;
            fout << s102   << endl;
          } // if the outfile is open
//-------------------------------------------------------------------

          for (int index=0; index < cs101.length(); index++) {
            sendbuf[index] = cs101[index];
          }

          // Sends data to server...
          bytesSent = send (m_socket, sendbuf, strlen(sendbuf), 0);
          if(bytesSent == SOCKET_ERROR)
              printf("Client: send() error %ld. \n", WSAGetLastError());
          else {
                 printf("Client: send() is OK - Bytes sent: %ld \n", bytesSent);
                 printf("Client: The test string sent: \"%s\" \n", sendbuf);
          }
//-------------------------------------------------------------------

          if (StartSeq == 1) {
            prev = i;
            StartSeq = 0;
          } else { // prev != 0
            if ((i-prev) > gap) {
              StartEvent = 1;
              prev = i;
            } // if (i-prev) > gap
          }

          cout << eventId << " " << i << " "<< prev <<" " << "sensor101 "<<c_jaar << "-" <<c_mnd << "-" <<c_dag << " " <<c_uur << ":" << c_min <<":" << c_sec <<"," << c_ms
//

        } // if (s101 > threshold101)

      } // if i>2
      i++;

    } // while ! eof()

    fin.close();
    fout.close();
  } // if file open
  else
    cout << "Unable to open file";

  WSACleanup();
  return 0;
}
```

17 Appendix F: filter.cpp

```
// Filter program
// author Gratian Schiopu
// datum  19-08-2010
// ver.:  2.00b
// This program filters events from sensor data file given a specific threshold
// For socket communication add to your linker the library file: ws2_32.lib
// Add this to your project:
// Project->Properties (Alt+F7) Linker->input->Additional Dependencies: ws2_32.lib

#include "stdafx.h"
#include "threshold.h"
#include "fileio.h"
#include "sampling.h"
#include "fileioxml.h"
#include "socketxml.h"
#include "socketcsv.h"

#using <mscorlib.dll>


using namespace std;
using namespace System;

int main() {
 char ans='y';
 int operation;

 while (ans!='n'){
  cout<<"The input flat file will be read and the output files will contain"<<'\n';
  cout<<"the detected events based upon a threshold used on the sensor 101."<<'\n'<<'\n';
  cout<<"Make a choice of what output file do you want"<<'\n';
  cout<<"[1]=threshold "<<"[2]=csvFile "<<"[3]=Sample2Csv"<<"[4]=xmlFile "<<"[5]=sockCsv "<<"[6]=sockXML "<<'\n';
  cout<<'\n';
  cin>>operation;

  switch (operation) {
    case 1: {
      thresholdCalc();
      break;
    }

    case 2: {
      makeCsvFile();
      break;
    }

    case 3: {
     Sampling2Csv();
     break;
    }

    case 4: {
      makeXmlFile();
      break;
    }

    case 5: {
      sockCsv();
      break;
    }

    case 6: {
      sockXml();
      break;
    }

    default: {
      cout<<"This is an incorrect operation!!"<<'\n';
      cout<<'\n';
    }
  } //end-case

  cout<<"Do you want to try again? [y/n] "<<'\n';
  cout<<'\n';
  cin>>ans;
  cout<<'\n';
 } // enddo

 cout<<"\n";
 cout<<"\n";

 cout<<"\n";
 system("pause");
 return 1;
}
```

18 Appendix G: SocketServer.cpp

```
// ============================================
// SocketServer.cpp
// Server application
// Data sent from the filter is being received by TCP/IP socket communication
// ver. 1.01
// date: 27-06-2010
// auth.: G.Schiopu
// the server can be run on another computer, client and server must share the
// same communication port.
//============================================

#include "stdafx.h"
#include <stdio.h>
#include <winsock2.h>

using namespace System;

int main() {

WORD wVersionRequested;
WSADATA wsaData;

int wsaerr;

int bytesSent;
int bytesRecv = SOCKET_ERROR;
char sendbuf[12] = "Ok - Server";   // data buffer used by the server to communicate

// initialize to empty data...
char recvbuf[12] = "";   // data buffer to capture data from the client


// Using MAKEWORD macro, Winsock version request 2.2
wVersionRequested = MAKEWORD(2, 2);

wsaerr = WSAStartup(wVersionRequested, &wsaData);

if (wsaerr != 0) {
    /* Tell the user that we could not find a usable WinSock DLL.*/
    printf("Server: The Winsock dll not found!\n");
    return 0;
} else {
    printf("Server: The Winsock dll found!\n");
    printf("Server: The status: %s.\n", wsaData.szSystemStatus);
}



/* Confirm that the WinSock DLL supports 2.2.*/
/* Note that if the DLL supports versions greater    */
/* than 2.2 in addition to 2.2, it will still return */
/* 2.2 in wVersion since that is the version we      */
/* requested.                                        */

if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2 ) {
    /* Tell the user that we could not find a usable WinSock DLL.*/
    printf("Server: The dll do not support the Winsock version %u.%u!\n", LOBYTE(wsaData.wVersion), HIBYTE(wsaData.wVersion));
    WSACleanup();
    return 0;
} else {
    printf("Server: The dll supports the Winsock version %u.%u!\n", LOBYTE(wsaData.wVersion), HIBYTE(wsaData.wVersion));
    printf("Server: The highest version this dll can support: %u.%u\n", LOBYTE(wsaData.wHighVersion), HIBYTE(wsaData.wHighVersion));
}


//////////Create a socket///////////////////////
//Create a SOCKET object called m_socket.
SOCKET m_socket;



// Call the socket function and return its value to the m_socket variable.
// For this application, use the Internet address family, streaming sockets, and
// the TCP/IP protocol.
// using AF_INET family, TCP socket type and protocol of the AF_INET - IPv4

m_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);


// Check for errors to ensure that the socket is a valid socket.
if (m_socket == INVALID_SOCKET) {
    printf("Server: Error at socket(): %ld\n", WSAGetLastError());
    WSACleanup();
    return 0;
} else {
printf("Server: socket() is OK!\n");
}


///////////////bind////////////////////////////
// Create a sockaddr_in object and set its values.
sockaddr_in service;

// AF_INET is the Internet address family.
service.sin_family = AF_INET;

// "127.0.0.1" is the local IP address to which the socket will be bound.
// service.sin_addr.s_addr = inet_addr("127.0.0.1");
service.sin_addr.s_addr = inet_addr("192.168.100.177");  // service bound to server IPadr.: 192.168.100.177

// 55555 is the port number to which the socket will be bound.
service.sin_port = htons(55555);



// Call the bind function, passing the created socket and the sockaddr_in structure as parameters.
// Check for general errors.
if (bind(m_socket, (SOCKADDR*)&service, sizeof(service)) == SOCKET_ERROR) {
    printf("Server: bind() failed: %ld.\n", WSAGetLastError());
    closesocket(m_socket);
    return 0;
} else {
    printf("Server: bind() is OK!\n");
}


// Call the listen function, passing the created socket and the maximum number of allowed
// connections to accept as parameters. Check for general errors.

if (listen(m_socket, 10) == SOCKET_ERROR)
    printf("Server: listen(): Error listening on socket %ld.\n", WSAGetLastError());
else {
   printf("Server: listen() is OK, I'm waiting for connections...\n");
}


// Create a temporary SOCKET object called AcceptSocket for accepting connections.
SOCKET AcceptSocket;


// Create a continuous loop that checks for connections requests. If a connection
// request occurs, call the accept function to handle the request.

printf("Server: Waiting for a client to connect...\n" );
printf("***Hint: Server is ready...run your client program...***\n");

// Do some verification...
while (1) {
    AcceptSocket = SOCKET_ERROR;
    while (AcceptSocket == SOCKET_ERROR) {
        AcceptSocket = accept(m_socket, NULL, NULL);
    } // while (AcceptSocket == SOCKET_ERROR)


// else, accept the connection...
// When the client connection has been accepted, transfer control from the
// temporary socket to the original socket and stop checking for new connections.

printf("Server: Client Connected!\n");
m_socket = AcceptSocket;
break;
} // while (1)

// Send some test string to client...
printf("Server: Sending some test data to client...\n");
bytesSent = send(m_socket, sendbuf, strlen(sendbuf), 0);

if (bytesSent == SOCKET_ERROR)
printf("Server: send() error %ld.\n", WSAGetLastError());
else {
    printf("Server: send() is OK.\n");
    printf("Server: Bytes Sent: %ld.\n", bytesSent);
}

// Receives some test string from client...and client
// must send something lol...



// bytesRecv = recv(m_socket, recvbuf, 200, 0);
// if (bytesRecv == SOCKET_ERROR)
// printf("Server: recv() error %ld.\n", WSAGetLastError());
// else {
//    printf("Server: recv() is OK.\n");
//    printf("Server: Received data is: \"%s\"\n", recvbuf);
//    printf("Server: Bytes received: %ld.\n", bytesRecv);
// }

bytesRecv = recv(m_socket, recvbuf, 12, 0);
while (bytesRecv != SOCKET_ERROR) {
printf(recvbuf);
    bytesRecv = recv(m_socket, recvbuf, 12, 0);
// sendbuf[0] = "O";
// recvbuf = "";
} // while (bytesRecv != SOCKET_ERROR)



WSACleanup();
return 0;
} // end main()
```