Internal Report 2013–13 September 2013

Universiteit Leiden
Opleiding Informatica

Enhancing Relation Discovery
in Unmarked Spatial Temporal Data
using Visualisation

Rick van der Zwet

MASTER'S THESIS

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

Enhancing Relation Discovery in Unmarked Spatial Temporal Sensor Data using Visualisation.

Rick van der Zwet
<hvdzwet@liacs.nl>
Leiden Institute of Advanced Computer Science, The Netherlands

September 26, 2013

Abstract

The objective of this Master Thesis is to find out howto enhance automated relation discovery in untagged spatial temporal data, using visual aids and meta-data learning.

This paper will demonstrate why existing implementations of automated relation detection could benefit from adding meta-data and how a newly defined feedback loop involving a human operator will enhance the sensor data analytics and relation discovery methods.

This approach is demonstrated by walking through several use cases, like interactive real-time visualization using heatmaps.

In this study it was found that using visual thinking as addition to automated relation discovery enhances the ability to find new relations.

Contents

1 Preface

Welcome! Allow me to introduce myself and say thanks the persons which I am grateful for helping and supporting me.

1.1 Motivation

Embedded Systems and Wireless Network information Technologies caught my attention as early as my teenage years and it was then when I decided to study Computer Science later in life.

I very soon realized that I am not a typical student as such. I was spending quite some time with practical aspects of IT systems, especially UNIX and FreeBSD derivatives making me realize that mastering this information could not be learned from a book or study, but only by self-learning. Luckily Leiden has a large number of FreeBSD and UNIX enthusiasts focused around various small and larger companies like Cope IPS[1], Joost Technologies B.V[2], TransIP, Optiver and

[1]The company is now called Dimensional Insight Netherlands
[2]Ceased operations in the Netherlands

volunteer organisations like Stichting Wireless Leiden.

My decision to start studying part-time and to work in IT turned out to be a beautiful combination of both words enabling me to learn both theory and practical examples of this beautiful field of research.

During my studies I encountered various other wonderful fields of computer science, ranging from game theories, modelling of biological processes with Petri-Nets and massive parallelisation. Things I never would have expected in Computer Science.

With my Bachelor Thesis, I focused on the massive parallelisation on GPUs keeping me close to my embedded systems interest. The large amounts of data slowly caught my interest. I always liked the idea that it is very simple to generate massive amounts of data and that it takes a lifetime to analyse this data.

My Master Project focused on even larger data sets in the field of High Frequency Trading. This is the first time that I found myself drowning in data not able to find all relations, how simple it looked at the first place. And to be honest I liked it a lot.

For my master thesis I decided to take an even deeper dive. I found a small company which was willing to share a lot of data. This data is gathered from sensors of embedded systems at various times and places.

I took the challenge to see how much of my knowledge gathered during my research on HFT data and analytics with the GPU could be applied for this very interesting data.

The challenge has proven to be a really though cookie and I am proud that I choose this path. This path allowed me to apply the knowledge I gathered during the years at LIACS Leiden University to the full extent.

After exactly ten years, with my part-time choice in mind, I proudly present you the result of my journey in Computer Science.

The result of my work is both theoretical and practical oriented. I hope you enjoy it as much as I did writing it.

1.2 Acknowledgements

I like to provide my acknowledgements to Prof. H.A.G. Wijshoff for his patience and drive to get the best out of all his students, by providing them a wide knowledge view and interesting new viewpoints.

I would also like to thank Gerard Cats for providing his insights with regards to relations discovery in meteorological data and for providing a clear view, guidance and an eye for attention during our various discussions on the subject.

A word of thanks to Dr. Erwin M. Bakker for feedback on discussions with regards to storage and databases.

A big thanks to Nick Hibma from AnyWi.com for providing me with endlessly amounts of data and his clear explanation of real-life questions and issues he is facing. The discussions I had and his feedback on my ideas allowed me to filter the wild-ideas and test them to reality.

Lastly I would like to thank my parents Jan and Joke for their continued support and my wife Inge for her patience, motivational talks and endless love.

2 Introduction

The last 5 years integration of electronic devices in our world is increasing at a pace we could have never imagined. More and more electronics are equipped in our equipment and new electronic devices with communication capabilities are added to places we have never seen electronics.

This new generation devices have one thing in common. They are generating a fair chunk of sensor data and unlike traditional devices this data could now be shared in almost real-time due to the fast amount of communication possibilities available now-a-days.

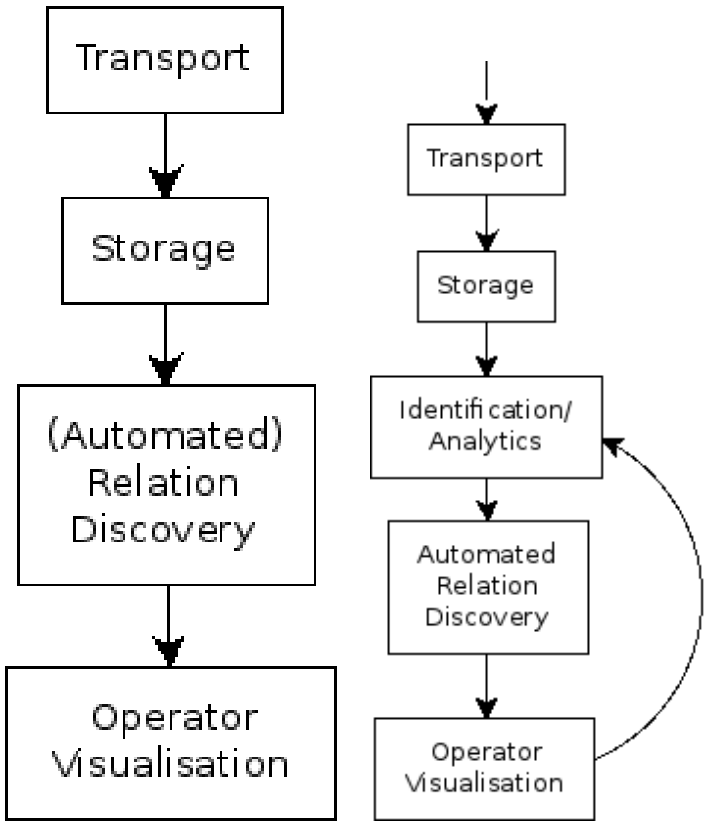
Gathering sensor data and processing them into centralized locations is done by various companies. I have been asked by an engineer of AnyWi.com to find out why there are connection drops in sensor transmission occurs within their setup, this setup is explained in detail in Section 5. When trying to solve this issue it was found that there is tooling missing to assist in *visual thinking* with relation to spatial temporal sensor data. Existing tooling and solutions were not

able to cope with the large amount of data. So I began to design an alternative, using human operator expert knowledge to provide extra knowledge to automated relation discovery process. This would allow faster detection of new relations. One possible way to make detection and exploration much more easy is by using Visual Thinking.

> Visual Thinking [1] is the phenomenon of thinking through visual processing. Visual thinking uses the part of the brain that is emotional and creative, to organize information in an intuitive and simultaneous way. Visual thinking is one of a number of forms of non-verbal thought, such as kinaesthetic, musical and mathematical thinking. Visual thinking may be more common for individuals with dyslexia and autism.
>
> Spatial-temporal reasoning is the ability to visualize special patterns and mentally manipulate them over a time-ordered sequence of spatial transformations. Spatial visualization ability is the ability to manipulate mentally two- and three-dimensional figures.
>
> Spatial-temporal reasoning is prominent among visual thinkers as well as among kinaesthetic learners (those who learn through movement, physical patterning and doing) and logical thinkers (mathematical thinkers who think in patterns and systems) who may not be strong visual thinkers at all.
>
> `https://en.wikipedia.org/wiki/Visual_thinking`

Processing large amounts of data with automatic discovery could take a considerable amount of time, as the searchable space is potentially very large. This time constrains causes the automatic discovery processes usually to be non-interactive batched process.

Analytics by automated discovery commonly follows the path as seen in Figure 1a. The data is sensed and transported to a storage location. Next automated discovery will try to process the data in order to find relations. Ones relations are found they are presented to the operator. Automatic Relation Discovery comes in many different forms and shapes, this methods will be further explained in



(a) Usual path of finding relations.

(b) Proposed alternative of finding relations.

Figure 1: Finding relations with automatic discovery is often a non-interactive process, where-as an interactive process potentially has benefits due to the expert knowledge of a human operator.

Section 10.2.

For this matter an alternative structure is proposed as shown in Figure 1b which will first uses analytics in earlier stages to annotate the raw data allowing automated discovery to go faster. For example if the analytics finds out a data column to be constant, this knowledge could for example be used by automated discovery algorithms to include or exclude certain columns during execution.

Furthermore I would like to add expertise of a human operator into the automated analytics loop. By speeding-up the automated discovery to levels which makes interactive usage possible it would allow a human operator to provide feedback on the automated discovery process.

The feedback by the automated discovery engine could for example present the operator a list of columns which could together potentially form a location readout. By visualisation of the different paths a human operator could select which sensors readings in fact represent a path, as the operator could quickly detect if a path is following a waterway or road when placed over geographical map. Ones a path has found the automated discovery could continue the search. It will next present sensors readings which could potentially have a relation to the path. One of such readings could be the speed of the object. By providing a listing of speeds to the human operator, the human operator could decide which speed read-outs are in fact possible for the object being examined. If the path showed that the object is travelling over waterways at all times, it could be identified as a ship. As such any speed of higher then $50km/h$ for this object would be questionable.

If the operator cannot make a decision, extra information could be requested. For example if the speed is presented the acceleration could also be calculated and displayed, such extra information in our speed example allows the operator to choose between a ship or an air-plane.

Sometimes the operator cannot make a decision and could for example mark several columns to potentially representing the speed. The automated discovery could continue the search and for example find and present the altitude possibilities together with the speed, allowing the operator to select the possible combinations.

The last operation possible by the operator is the discrimination of the presented alternatives, if neither of the options are possible. This requires the automated discovery to find an alternative column or assume the data is not present in the sensor data output.

The described automated human-assisted discovery for spatial temporal data,

would potentially make finding relations more intuitive and faster.

This introduction gives an outline of what to expect in this Master Thesis, the remainder of this document is structured as follows. In Section 3 an introduction to sensors is given. Section 4 will zoom in on the data generated by the sensors. In Section 5 transport of sensor data to the central storage engine will be discussed. The next Section 6 will describe howto store the data is gathered. Section 7 will describe why meta-data annotation on raw data is an important step to take. Section 8 will show howto perform filtering on raw-sensor data. Section 9 will touch analytics methods. Section 10 will provide a breakdown of existing analytics tooling solutions and why they are lacking functionality for spatio-temporal analytics using visual thinking. In Section 11 an example implementation of visualisation of sensor data for visualisation recognition by an operator is presented. The results of the experiments will be summarized in Section 12, following with recommendations for further work in Section 13.

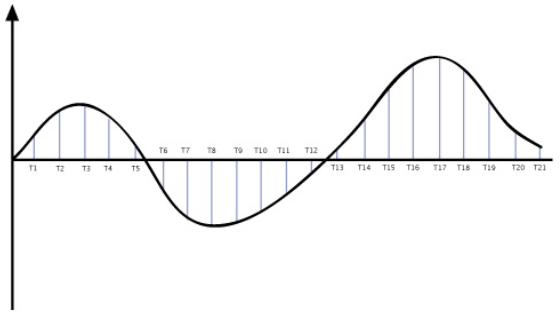
3 Sensors

Sensor data is data which is sensed by the sensor. The data is derived from it surrounding or environment. For example a temperature sensor could read the temperature from a certain object if attached to the object or it could read the temperature from the environment if the probe is reading the air temperature.

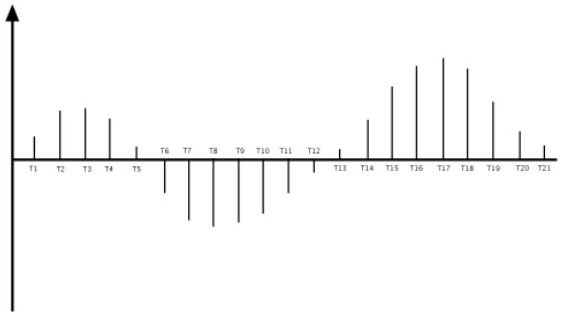
Apart from "external" measurements values from the environment, like temperature, speed and humidity, it could include measurements values about the electronics itself, like battery status, CPU usage and bandwidth usage.

There are two types of sensor readings, the temperature reading for example is a continuous analogue reading. The second type is the discrete digital readings, for example a door sensor which detect whether the door is open or not, only has two states.

The digital readings could be interpreted (and stored) right away, the analogue readings however needs to be converted before they are able to be processed. One way of converting the analogue reading to digital output is by using PCM sampling. PCM (Pulse Code Modulation) is a technique which samples the data by getting the reading at pre-defined time intervals. On example of PCM sampling is found in Figure 2.



(a) Audio Signal



(b) PCM signal

Figure 2: Conversion of analogue readings to digital output with PCM conversion.

PCM sampling will cause data readings to get lost, since there is no data storage of the intervals where no sampling takes place. To reconstruct this data one could try interpolation, however this techniques will not be able to retrieve the exact result any-more. If sampling at a smaller interval is requested a higher resolution sensor is required. Making higher resolution sensors however is more expensive, so there will always be a trade-off between sensor resolution and price willing to pay for the sensor.

This new devices now move around all over the globe and look to the real world in various areas. And instead of just a few of them, there are many and many of devices together gathering data at a rate we have never seen before.

And the innovation does not stop, it is only increasing at a bigger speed, the so called "Internet Of Things" [2] is a very accurate description where we are heading to; a totally inter-connected world where every device is connected to the internet and is able to communicate with it.

Storing new massive amounts of sensor data gets to a whole new dimension since there are so many different sensors around. This comes both in the storage requirements and in the format the data is stored. Storing the data in an unified format is not always possible any more. For example a heart-rate sensor has a very different set of properties than a sensor used to measure temperature on an oil-tanker.

There are various ways of storing this new data and they all try to allow flexible data inserts and fast searches. However traditional databases are hardly sufficient any more. The amount of entries (rows) grows beyond searchable/indexable possibilities. Secondly it becomes difficult to add new sensors (new columns) if the dataset gets really large. New database storage systems are invented to cope with this limitations, examples are storage using BigTable [3] and MapReduce [4]. More on storage is found in Section 6.

This new sensor data is stored in this new database types and is usually kept for a longer time. This itself however makes a challenge if the data is accessed at later stage. The data columns descriptions might not be present any more or it context is not clear any more. This poses huge issues when it comes to usability of the measured sensor data in time.

Time is not the only issue when it comes to storing sensor data. There are a lot of parameters to store. One special kind of data is the data gathered by mobile sensors. Mobile sensors move around in the location space, this space is not only limited to Earth, looking at sensor data from sensors in satellites and

space exploration equipment. This sensor data has normally contains at least 4 parameters. The first parameter is the time parameter, which contains the time the reading took place.

Secondly and third comes the mobility specific parameter. This parameter is written down in 2 coordinate parameters if the object is moving over a 2D grid.

Lastly the reading of the sensor itself. Optionally we could add an 3rd location parameter if the sensor object is moving in a 3D grid. Sometimes a sensor reports the distance of a trajectory travelled from a certain point. This is not considered location data if the trajectory itself is not stored, as there is no way to depict the location in a different matter.

For some (mobility) sensors the view direction is also crucial information for an accurate description of the sensor data. For example, if the sensor gets image information, knowing in which direction the picture was taken is really valuable information. This will then add 3 more parameters which are rotational parameters, namely the angles of the sensor head on the X en Y axis, as seen on Figure 3[3]. Please note that this sensor also contains a gravity sensor in addition to the x-axis and y-axis readings. This potentially could give different sensor reading, namely tilt, vibration, motion and acceleration, which could all be derived from the readings from this sensor.

Next the head position is required, which is usually done by a magnetic compass sensor or an alternative which uses implicit direction calculated on coordinate movements. The second is less precise as the direction cannot be calculated if the sensor is idle at one location.

Lastly the coordinates are measurements from a location sensor like a GPS (Global Positioning System) or its Russian counterpart GLONASS (GLObal'naya NAvigatsionnaya Sputnikovaya Sistema[4]) sensor.

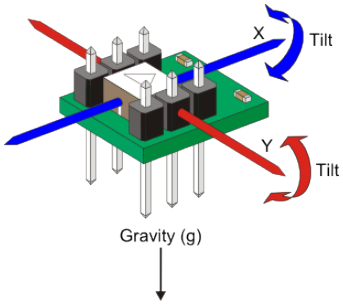
4 Sensor Data

Figure 4 shows a small snippet of data stored. This example illustrates the case of making it hard to re-discover the context the data get stored in. There are various columns of integers, floats and string values and not all rows are filled.

[3]Pictures from `http://learn.parallax.com/KickStart/28017`

[4]GLObal NAvigation Satellite System

(a) Sensor package with pin-outs for data readings and power supply.



(b) Schematic representation of measurements available.

Figure 3: A Memsic 2125 Dual-axis Accelerometer, is an example of a sensor which allow measurements of tilt of two angles.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	
1	[illegible]														
2										1301529601	00:0d:b9:1b:57:10				
3	48	195235.295134	2862782	155635.204983	445549676704.841	1.2	48.98654333	8.27074167	3D	1301529601	00:0d:b9:1b:57:10		1 no link		NETGEAR/0(
4	48	195240.757798	2862784	155635.096263	445549676734.682		48.986575	8.27079833		1301529601	00:0d:b9:1b:57:10				
5	48	195246.220462	2862786	155634.987543	445549676764.522		48.98660667	8.27085667		1301529601	00:0d:b9:1b:57:10				
6	48.6	195257.145791	2862789	155634.824469	445549676824.204		48.98663833	8.27091333		1301529601	00:0d:b9:1b:57:10				
7	48.6	195262.608455	2862791	155634.71575	445549676854.045		48.98667167	8.27097		1301529601	00:0d:b9:1b:57:10				
8	48.7	195268.071119	2862793	155634.607031	445549676883.885		48.98670333	8.27102833		1301529601	00:0d:b9:1b:57:10				
9	48.8	195273.533784	2862795	155634.498312	445549676913.726		48.986735	8.271085		1301529601	00:0d:b9:1b:57:10				
10	48.9	195278.996448	2862797	155634.389593	445549676943.567		48.98676667	8.27114333		1301529601	00:0d:b9:1b:57:10				
11	49	195284.473001	2862800	155634.22651	445549676973.559		48.98679833	8.2712		1301529601	00:0d:b9:1b:57:10				
12	49	195289.949554	2862802	155634.117791	445549677003.552		48.98683	8.27125833		1301529601	00:0d:b9:1b:57:10				
13	49	195295.426108	2862804	155634.009073	445549677033.544		48.98686167	8.27131833		1301529601	00:0d:b9:1b:57:10				
14	49	195300.902661	2862806	155633.900355	445549677063.537		48.986895	8.27137667		1301529601	00:0d:b9:1b:57:10				
15	49	195306.379214	2862808	155633.791637	445549677093.53		48.98692667	8.271435		1301529601	00:0d:b9:1b:57:10				
16	49	195311.881489	2862810	155633.68292	445549677123.805		48.98695833	8.27149333		1301529601	00:0d:b9:1b:57:10				
17	49	195317.397654	2862812	155633.574202	445549677154.233		48.98699	8.27155167		1301529601	00:0d:b9:1b:57:10				
18	49	195322.913818	2862814	155633.465485	445549677184.661		48.98702333	8.27161		1301529601	00:0d:b9:1b:57:10				
19	49	195328.429982	2862816	155633.356768	445549677215.089		48.987055	8.27167		1301529601	00:0d:b9:1b:57:10				
20	49	195333.946147	2862818	155633.248051	445549677245.517		48.987085	8.27173		1301529601	00:0d:b9:1b:57:10				
21	49	195339.4762	2862820	155633.139334	445549677276.098		48.98711667	8.27179		1301529601	00:0d:b9:1b:57:10				
22	50	195345.006253	2862822	155633.030617	44549677306.68		48.98715	8.27185		1301529601	00:0d:b9:1b:57:10				
23	50	195350.536306	2862824	155632.921901	445549677337.261		48.98718167	8.27190833		1301529601	00:0d:b9:1b:57:10				
24	50	195356.092081	2862826	155632.813184	445549677368.128		48.98721333	8.27196833		1301529601	00:0d:b9:1b:57:10				
25	50	195361.661746	2862828	155632.704468	445549677399.149		48.987245	8.27203		1301529601	00:0d:b9:1b:57:10				
26	50	195367.23141	2862830	155632.595752	445549677430.17		48.987275	8.27209		1301529601	00:0d:b9:1b:57:10				
27	50	195372.801074	2862832	155632.487037	445549677461.191		48.98730667	8.27215167		1301529601	00:0d:b9:1b:57:10				

Figure 4: A random sample of data.

The snippet column headers are blurred on purpose as we are working with *Untagged Data*. There is simply no usable identification available to find out what this data is all about. The identification of the columns is as useful as the column headers 'A', 'B', 'C", etc...

Having said this, we are not totally clueless about the data in Figure 4 as we know that the data is gathered on a ship. This valuable piece of so-called meta-data information tells us that the data at least contains a time-stamp column, 2 location columns and at least one sensor reading.

This snippet is a part of the actual data which later used on to generate plots. This paper will describe the various steps needed to take us from the data blob to

an usable piece of data which could be used for analytics purposes.

Secondly the question arrived what kind of other relations could be found from the data and more effectively howto improve the storage strategies such that effect re-use of the data for other purposes could be made possible. Currently for every new application the data model has to be re-invented and it is often found that data is missing. This follows on the fact that the data gets analysed and more questions are found. However for those questions the required sensor data might not be gathered in the first place.

The spatial temporal sensor data gathered contains a lot of data. Our example dataset of a single aggregator contains already 193 columns and roughly $27.8M$ rows. When combining all 7 aggregators the search space expands to roughly 1338 columns and $138.4M$ rows. A detailed listing could be found in Table 1.

identifier	columns	rows
00:0d:b9:1b:57:10	239	$43.8M$
00:0d:b9:1b:57:14	180	$8.6M$
00:0d:b9:1b:57:28	184	$13.2M$
00:0d:b9:1b:57:30	157	$2.3M$
00:0d:b9:1b:5d:a0	213	$37.3M$
00:0d:b9:1b:5d:c0	172	$4.8M$
00:0d:b9:20:46:24	193	$27.8M$
combined datasets	1338	$138.4M$

Table 1: Overview of sizes of provided datasets by various aggregators, the aggregators have different amount of rows due to the variation in run times. It also has a different amount of columns due to different sensor configurations on the aggregators.

Manual analytics of the data are not longer sufficient to provide answers. In order to handle the massive amounts of data, automated discovery is required to assist in finding relations in the data.

5 Transport

Before talking about transport, knowing where the data will be stored is a pre. The sensor data could be stored on 3 locations, the traditional method is to store

it on a centralized location. Local storage on the sensor could be an alternative. And the hybrid solution is to store data at intermediate nodes.

The choice where to store the data is depicted by two parameters, the storage capabilities and the transport capabilities.

Sensor data is usually gathered by devices with a very low power profile, traditional data transport consumes a fair amount of power, so there need to be improvements to ensure efficient data transport. This could be done for example by defining algorithms for sending data, which minimize the time the system needs to be active in data communication [5].

Another approach is store-and-forward, where the data is stored on nodes for a longer time and transmitted in batches. An alternative approach to the method is by sending the data to specialized storage nodes and have the storage nodes transport the sensor data to a centralized location in batches [6].

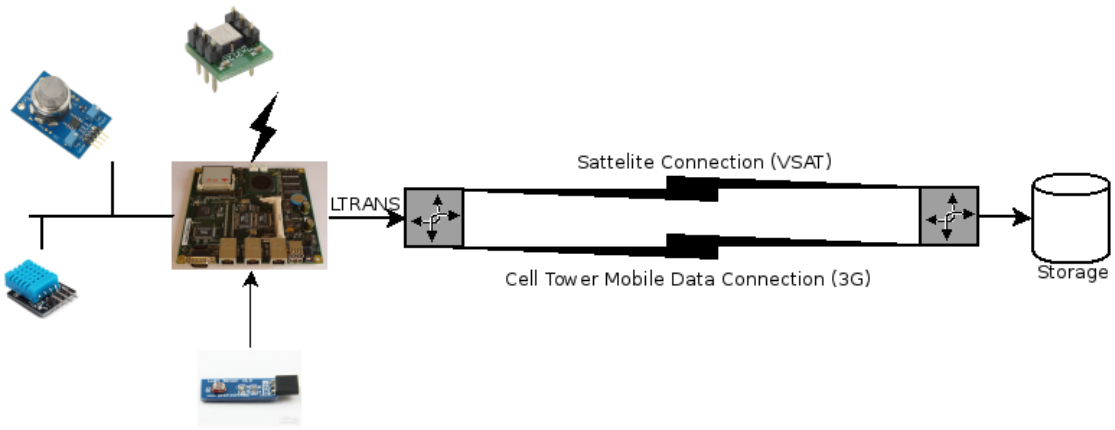


Figure 5: Data is gathered by sensors and transmitted to the aggregator device using various protocols, like I2C, UART, Ethernet, wireless connectivity protocols such as 802.11abgn (WiFi), ZigBee and various proprietary sensor transport protocols, like SenSite.

In our case the sensor data is gathered by sensors connected to an aggregator which is located on a ship. The aggregator is connected to the central power supply of the ship. This aggregators are small i386 embedded systems running FreeBSD operating systems, with 256MB of RAM and no writeable persistent storage. Figure 5, shows the complete path of which the sensor data follows from the moment it is “sensed” to the storage.

Satellite is preferred as most of the sensors are placed on ships which are

travelling in various countries within Europe. This ships travel over the main waters of Europe, which are rivers, canals and lakes. Within Europe data roaming over 3G mobile connection is expensive. To ensure redundancy the aggregators are connected through 3G mobile connection and an satellite connectivity to a central storage system.

The satellite connection is preferred above a 3G mobile connection as within Europe mobile data costs are in general very costly compared to satellite connectivity, due to roaming costs and different data-plans in every European country. On the flip-side the 3G mobile connection is in general more reliable than the satellite connection, as satellite connections do not have a connection if there is some object in between the satellite and the satellite received. A thundercloud or mountain in the Line Of Sight (LOS) to the satellite, could make communication with the satellite impossible. So the trade-off is to have both data communication paths available for use. The aggregator uses sensor reading to select the best gateway to use.

The aggregator however sometimes picks the wrong path or start switching paths frequently. Because every switch takes around 30 seconds to settle, switching a lot means that there is virtually no data communication possible. For the sensor readings this is not an issue as the data is buffered, however there are multiple other information systems using the line connections as well. Since some of them are interactive the users of this systems cannot use the systems during this time periods. Section 11 will zoom in on this issues and tries to find an answer.

The aggregators sent the sensor readings using the `LTRANS` protocol over one of the active links. The transmission uses the Internet to arrive to the storage location. The `LTRANS` Protocol and its history is described in section 5.

Since the power consumption is not an issue as the sensor unit is connected to the main power of the ship, allowing us to be flexibly when it comes to transport and storage costs. A real-time protocol is preferred as we have limited amounts of storage available on the units itself. Secondly storing and analysing the data is more easy and faster if the data is stored on a centralized place. For example when thinking on redundancy, when storing the data at the aggregators, data somehow needs to be duplicated to cater with hardware failure, whereas this data is stored on a centralized location, simple existing backup solutions could be used for redundancy requirements.

Where power is “free”, data transport however does not comes free. Every byte send over the 3G mobile phone connection or the satellite connection has a cost

price, so the main goal is to avoid sending too many data over the line.

For the matter a (simple) transport protocol `LTRANS` is designed, which copes with compression of data and avoiding data overhead.

Traditionally sensor systems are typically configured to sent their complete state at predefined intervals, for example every second, this will give the following list. The lat and lon readings are sent by a single sensor, next to this are 4 other sensors reporting readings. The data is structured in a table for readability purposes:

```
timestamp;lat;lon;sensorA;sensorB;sensorC;sensorD
1378635719;4.12345;52.6789;    ;     ;    ;
1378635719;       ;       ;10.1;     ;    ;
1378635719;       ;       ;    ;0.566;    ;
1378635719;       ;       ;    ;     ;15.0;
1378635719;       ;       ;    ;     ;    ;foobar

1378635720;4.12345;52.6789;    ;     ;    ;
1378635720;               ;10.2;     ;    ;
1378635720;       ;       ;    ;0.567;    ;
1378635720;       ;       ;    ;     ;15.0;
1378635720;       ;       ;    ;     ;    ;foobar

1378635721;4.12345;52.6789;    ;     ;    ;
1378635721;               ;10.1;     ;    ;
1378635721;       ;       ;    ;0.567;    ;
1378635721;       ;       ;    ;     ;15.0;
1378635721;       ;       ;    ;     ;    ;foobar

1378635722;4.12345;52.6789;    ;     ;    ;
1378635722;               ;10.2;     ;    ;
1378635722;       ;       ;    ;0.567;    ;
1378635722;       ;       ;    ;     ;15.0;
1378635722;       ;       ;    ;     ;    ;foobar

1378635723;4.12345;52.6789;    ;     ;    ;
1378635723;               ;10.2;     ;    ;
1378635723;       ;       ;    ;0.567;    ;
1378635723;       ;       ;    ;     ;15.0;
1378635723;       ;       ;    ;     ;    ;foobar
```

```
1378635724;4.12345;52.6789;    ;     ;    ;
1378635724;               ;10.2;     ;    ;
1378635724;       ;       ;    ;0.567;    ;
1378635724;       ;       ;    ;     ;15.0;
1378635724;       ;       ;    ;     ;    ;foobar

1378635725;4.12345;52.6789;    ;     ;    ;
1378635725;               ;10.1;     ;    ;
1378635725;       ;       ;    ;0.566;    ;
1378635725;       ;       ;    ;     ;15.0;
1378635725;       ;       ;    ;     ;    ;bert

1378635726;4.12345;52.6789;    ;     ;    ;
1378635726;               ;10.1;     ;    ;
1378635726;       ;       ;    ;0.566;    ;
1378635726;       ;       ;    ;     ;15.0;
1378635726;       ;       ;    ;     ;    ;bert

1378635727;4.12345;52.6789;    ;     ;    ;
1378635727;               ;10.1;     ;    ;
1378635727;       ;       ;    ;0.566;    ;
1378635727;       ;       ;    ;     ;15.0;
1378635727;       ;       ;    ;     ;    ;bert

1378635728;4.12345;52.6789;    ;     ;    ;
1378635728;               ;10.1;     ;    ;
1378635728;       ;       ;    ;0.566;    ;
1378635728;       ;       ;    ;     ;15.0;
1378635728;       ;       ;    ;     ;    ;bert

1378635728;4.12345;52.6789;    ;     ;    ;
1378635728;               ;10.1;     ;    ;
1378635728;       ;       ;    ;0.566;    ;
1378635728;       ;       ;    ;     ;16.0;
1378635728;       ;       ;    ;     ;    ;foobar
```

The above samples has a number of assuming set. At first it assumes that all sensor readings are sent in the same intervals and are aligned perfectly around the second. There is however a fair amount of overhead of sensor readings sent

as all sensors include the time-stamps, which thus gets duplicated multiple times. The first improvement to be done is by combining sensor readings of the same time-stamp range and sending them in one batch. This could be done using an aggregator. Do mind by using an aggregator it will update every reading as fast as the fastest sensor. For example if one sensor is reporting at 10 second interval and another one is reporting at a 1 second interval, all values get sent every 1 second:

```
timestamp;lat;lon;sensorA;sensorB;sensorC;sensorD
1378635719;4.12345;52.6789;10.1;0.566;15.0;foobar
1378635720;4.12345;52.6789;10.2;0.567;15.0;foobar
1378635721;4.12345;52.6789;10.1;0.567;15.0;foobar
1378635722;4.12345;52.6789;10.2;0.567;15.0;foobar
1378635723;4.12345;52.6789;10.2;0.567;15.0;foobar
1378635724;4.12345;52.6789;10.2;0.567;15.0;foobar
1378635725;4.12345;52.6789;10.1;0.566;15.0;bert
1378635726;4.12345;52.6789;10.1;0.566;15.0;bert
1378635727;4.12345;52.6789;10.1;0.566;15.0;bert
1378635728;4.12345;52.6789;10.1;0.566;15.0;bert
1378635729;4.12345;52.6789;10.1;0.566;16.0;foobar
```

There are various methods to reduce the amount of data to be transferred, for example by filtering or caching. This how-ever comes with a penalty as the sensor system will need more computing power and more (RAM) local storage. The choice whether to reduce the data sent is a trade-off between sensor costs and transport costs.

Reducing the amount of data to be sent could be done in multiple ways. First we could send the initial state and next the delta of the changes. The initial packet will contain the state of all sensors, updates are only sent out of any of the data packets changes. The entries received will look like:

```
timestamp;lat;lon;sensorA;sensorB;sensorC;sensorD
1378635719;4.12345;52.6789;10.1;0.566;15.0;foobar
1378635720;4.12345;52.6789;10.2;0.567;15.0;foobar
1378635721;4.12345;52.6789;10.1;0.567;15.0;foobar
1378635725;4.12345;52.6789;10.1;0.566;15.0;bert
1378635729;4.12345;52.6789;10.1;0.566;16.0;foobar
```

The amount of entries is now reduced by 50% by not sending the lines which are not changed, but there are still quite some unchanged entries in the data received.

To save some more data we could leave out entries which are unchanged. This gives us the benefit of even less data being transferred.

```
timestamp;lat;lon;sensorA;sensorB;sensorC;sensorD
1378635719;4.12345;52.6789;10.1;0.566;15.0;foobar
1378635720;       ;       ;10.2;0.567;    ;
1378635721;       ;       ;10.1;     ;    ;
1378635725;       ;       ;    ;0.566;    ;bert
1378635729;       ;       ;    ;     ;16.0;foobar
```

This is not a free lunch, as we are in trouble if the receiver misses an update, due to transport errors for example. There is no way to recover as we do not receive a "full-state" periodically. So if the transport protocol is unreliable there is an extra requirement in this method to make sure packets are always received by for example acknowledging the packet or by sending "full-states" on periodic basis, such that "unknown" states could be avoided.

For the case of `LTRANS` it is fixed by sending acknowledgements on every packet received. If no acknowledgement is received for a period of time the sensor will start discarding new data readings if the sensor internal storage is full. When data communication is re-established the sensor will (re)start the communication stream with a full packet to announce its full current state.

When it comes to saving there is one extra item which could potentially make a difference. There is a lot of white-space transmitted and control characters transmitted, this comes due to fact the sensor transmissions are sent using a fixed column structure.

Having fixed amount of columns is useful if the sensor count is constant, however sensor counts these days are changing. For example a sensor starts to reporting an extra information field or there are extra control fields added depending on the state of the system. A GPS sensor for example has a field which tells whether it has a "fix" e.g. knows howto calculate an accurate location. When this field is set an alternative field is created showing how accurate the calculations are and if the sensor is probed to display more verbose information it for example start displaying in an extra field the satellite IDs is used in the calculations.

For all the changes there is no flexibility of adding and removing sensors in fixed-column set-up, apart from re-initialising the entire array every time a sensor field is added or removed. The behaviour is sometimes considered to be somehow

inflexible so there is a second way defined of sending sensor data. The sensor reading could also be sent as key=value pairs. For our example below will this yield to the following result:

```
timestamp;
1378635719;lat=4.12345;lon=52.6789;sensorA=10.1;sensorB=0.566;     \
  sensorC=15.0;sensorD=foobar
1378635721;sensorB=0.567
1378635725;sensorB=0.566;sensorD=bert
1378635729;sensorC=16.0;sensorD=foobar
```

The key=value method as alternative to the column based storage allows us to be as flexible as we want when it comes to sending sensor data to the store system. However within the key=value method there is extra data introduced by the key= entries in the transmission, which could lead to a larger data-stream. Sometimes a key almost always changes, for example time-stamp and the location parameters, which is in our case **lat** and **lon**. The choice of which keys are "tabular" and which keys are of "key=value" type is currently set manually by the operator. This leads to the following result:

```
timestamp ;lat    ;lon    ;
1378635719;4.12345;52.6789;sensorA=10.1;sensorB=0.566;             \
  sensorC=15.0;sensorD=foobar
1378635721;       ;       ;sensorB=0.567
1378635725;       ;       ;sensorB=0.566;sensorD=bert
1378635729;       ;       ;sensorC=16.0;sensorD=foobar
```

The sensor in this case was idle at one location, so it would actually be a better choice if the location parameters where set as "key=value" entries instead. This would have reduced the data overhead in this case.

6 Storage

The logical choice for storage of sensor data would be a relational database. This could either be a row oriented or column oriented storage, giving you the benefits of being able to process and query your data in a standard and well defined fashion,

for example with SQ queries. With sensor data however there are a few catches when using this kind of databases.

Firstly in order to store your data you need to come up with a database design upfront. With sensor data here lies the catch. Sensor data is highly flexible and in practice it is not known what is going to be stored. The database design needs to cope with this changing requirements in a very dynamic way. Adding and changes in a relational data is an expensive operation and takes up a lot of resources.

Secondly, resources are also an issue when it comes to search indexing, since there are so many indexes you could search from. As the sensor array is dynamic there is also no telling which will be important in the future. In real-life database storage and database usage are two totally different fields of applications. So it is actually best to index in a more dynamic way. This is a little bit more then just telling the type of the field. This so-called meta-data properties of the data will be of great concern at later stages. More on meta-data and indexing in section 7.

Lastly, sanitizing the data before it is insert into a database, might also be a tempting idea as you for example do not need the precision of certain sensors anyway or might discard certain entries completely. However this is again an assumption which is very dangerous with dynamic sensor data storage. The precision might not be needed as of this moment, but in a few years from now you find yourself thinking that you should have stored the data in its original form.

Sensor data has only one property which is indexable, namely the time of the reading. Every sensor reading will have an associated time-stamp and the time-stamp will be increasing for new readings of a sensor.

For spatial sensor information the location property could also be identified as an always indexable field, but only if the location property is chosen in such way that it uniquely identifies a position. GPS coordinates are a good example of such location parameters, since you could unique map any location on earth with such coordinates [5]. The distances to the nearest 3 cell towers is a bad set of parameters as cell towers could be changed in time.

One entry often forgotten in data storage is the fact that the data also needs to be readable after many years from now. Storing the data in today's technologies beautiful binary blobs makes the risk of not being able to read your data any more in a few years from now, due to technical or licensing limitations. The most

[5] If your sensor is going to the moon or further you might want to consider an alternative coordinate position systems for your sensors

trivial and simple solution to this issue is to store the data in a format mankind could potentially read in many more years from now. The only format to fit this requirement for the past 50 years of computer science is simple plain text storage.

The central storage system in our case is modelled after plain text storing it is running FreeBSD and has a large storage array for storing all the raw data uncompressed. The records are stored in `LTRANS` format. `LTRANS` is a format defined by AnyWi.com and is combined tabular and key=value pair flat file storage solution. An example file is found in Appendix C. The full protocol description is found in Appendix O.

Searching in plain text data could be done very easy, by either a human or a computer however it is potentially very time consuming when walking over a lot of data. Sensor data unfortunately happens to be a lot of data, so we require some tricks to be able to search in data in the first place.

Firstly the sensor readings will be stored in files identified by timeslots, for example a reading which took place at Sunday 8 September 2013 at 14:16 CEST, will be stored at the following location `2013/09/08.txt`. The `/` is a directory delimiter. There is a limitation on this approach, since file systems have a limit on how large a file can grow. When receiving a lot of data consider using a storage strategy which includes hours or minutes, like this `2013/09/08/12/16.txt`. See that time is stored in a different format, by storing all entries as UTC (Coorinated Universal Time), or a similar base set, it will make sure that the local time setting of the sensor does not affect our storage strategy.

The above proposed system works really well for single sensors, however it does no longer work if there are multiple sensors of the same kind reporting sensor data to the storage system. To circumvent this issue there needs to be an identification transmitted within the sensor data stream. This identification number needs to be unique, both now and in the future. Various implementations of this so-called Universally Unique identifier (*UUID*) exists, as defined by the ITU in RFC 4122 [7]. The data will then be stored in the following format:

```
/storage/<UUID>/2013/09/08/12/16.txt
```

Most sensor systems include such a number, however make sure that the number is `really` unique. There are various cases where the number as present by the vendor or implementation is not actually unique. For example one case used the serial number of the sensor as unique identifier, however the sensors from competing vendors used the same numbers range, making it impossible to identify the

sensor data.

A second case used the `MAC` address of the sensor node as unique identification for the sensor. This node was attached to a ship collected all kind of data of the ship. Due to the fact the ship was replaced the sensor was transferred to a new ship. This caused all kind of issues on the data analytics part as there was no knowledge of the unique ship identifier stored in the sensor data. One could argue that the data storage process was not broken due to the change of the aggregator to the new ship, however there is an important point to emphasize over here. Make sure the identification used for your sensor is uncoupled from psychical properties of the sensor node in use, by storing this information within the sensor node readings right from the start.

Retrieval data in this stored method is very fast when it comes to getting sensor data for certain time intervals. Storing data based on the time-stamps is not so efficient when for example a request for sensor of a specific area is needed, there are two alternatives to consider with this regard. The first solution would be to also store your sensor data in a location style storage system. For example a sensor reading taken at location latitude 52.12345 and longitude 5.4567 is than stored at `52.12/5.45/67.txt`. Storing based on location is somehow harder as you need to find a way to segment your location space in blocks, which could scale big enough for further needs.

This extra storage method will be a very expensive option as you will need to store all your data twice, so this option should be used sparely or to be avoided. The alternative solution is written down in Section 7.

As for storage, there are a few other important benefits when storing the sensor data in plain text files. Adding storage is extremely easy as today's file-systems are virtually unlimitedly expandable [8].

Secondly data could be easily queried by all programs. Instead of being limited to a database API, database access comes free, making it flexible to implement own query methods on the data. This allows making different kind of database access possible. The information index meta-data as written down in Section 7 would potentially allow unlimited variables on howto flexible index the data and search through the data.

7 Meta-data

Meta-data are annotations to the sensor readings. Meta-data provides context of the environment the sensor was placed or about the sensor properties itself. An other type of meta-data is the information which summarizes the data content itself.

7.1 Virtual Meta-data

Not all properties of a sensor are transmitted by the sensor itself. For example properties likes the sensor type and sensor accuracies are normally not included, since if this static information was included in every transmit it would generate a fair bit of overhead. With the proposed transmitting schema however this is no issue anymore. The meta-data properties of the sensor could be included as "key=value" pair, making them included only ones by the to-be-transmitted-data by default, since this keys do not change anymore.

If sensor meta is not transmitted by itself a "virtual sensor" could be defined. The virtual sensor will transmit the sensor meta information, allowing the meta information stream to be included into the sensor readings.

It is vital to keep the meta-data of the sensor up-to-date and included. Without this information it is very hard to know what the properties are when the data blob is read at a later stage. One could safely assume that not including the meta-data into the sensor output will render the data useless by any person/system who needs to read the data at a later stage.

Take for example a sensor which is reporting a value of 115.12356. This could be a temperature sensor which is either a hot temperature in degree Celsius, but it could might as well be measuring the temperature in Fahrenheit, making it a moderate temperature (46.1111 degree Celsius).

It even gets more interesting if the knowledge about the fact the sensor is a temperate sensor is lost. Without any kind of meta information description this sensor, it could might as well be an angle, height or speed sensor.

Next comes the accuracy, when looking at various readings one could conclude that sensor is operating in a 0.00001 range of accuracy. Unfortunately there are large number of sensors around which report a greater detail as they are actually

designed for. For example this sensor is designed to be used as boiling temperature measurements which will report its readings in steps of 1 degrees. Any sensor reading produced by this sensor will have 0 digits of precision. This means that a reading of 63.1231 is to be read as 63, the remainder 0.1231 is to be discarded. Without knowing this limitation during the analytic phase one could draw a fast kind of wrong conclusions.

An other case for keeping the meta-data of the sensor finds it use in data validation and generating the informational meta-data as found earlier in Section 7. Also without the data it gets very hard to pre-filter measurement readings, which will be explained in Section 8.

Most of the time the "Virtual Meta-data or the so called properties of a sensor are not stored at all or are not stored within the relation of the data to be stored. This will essentially render the data useless if the data is exposed to a new system or new operator. It is safe to say: "The meta-data is the key to unlock its database".

7.2 Meta-data as Summary

If a sensor would generate one reading every second, you will get 86400 entries a day, roughly 2.6M entries a month and 32.2M entries a year. This is just for a single sensor reading. Sensors usually come in many different kinds, so you can quickly see the entries growing to numbers where is it no longer maintainable in traditional databases.

Searching for large sets of data becomes slow as indexing gets more difficult for larger entries, so instead of indexing the whole dataset, we propose index classification of the data based on the subsets stored like Section 6.

Instead of searching in the whole file, only the index will have to be read which could be generated initially during the storage phase. Such indexes could also be considered meta-data of the sensor readings as they provide information about the data itself.

There are a number of properties to store in the index. A list of indexing fields useful for spatial temporal sensor data are:

type Classification as found in the data, this could either be a string, float or integer

total Amount of entries to be found in this dataset. This for example allows to check if it is worth checking the file at all.

unique (optional) To be used together with the total counter to see if the entry is to be defined as unique identifier of the dataset, unique calculations could potentially memory consuming so this field might not always be filled.

maximum Maximum number as found in the dataset, this is useful for range checking operations.

minimum Minimum number as found in the dataset, this is useful for range checking operations.

count_minimum The amount of times the minimum entry appears in the dataset, useful in detection the normal state of a (boolean) sensor.

count_maximum The amount of times the maximum entry appears in the dataset, useful in detection the normal state of a (boolean) sensor.

changes (optional) If the data is limited to only a certain amount of values represent the values it could take over here. This field is memory consuming to calculate, so there is a fixed preset defined calculation to avoid the whole dataset to appear over here.

choices (optional) Sometimes a sensor consist of a composite list of values, for example the satellites identifiers currently visible to the sensor, this list gets put to the sensor as string with a separation characters. This field will show the amount of choices present over here. This field might not be present or complete due to memory constrains during parsing for example.

The rather flexible set of index meta information provides a good start when searching trough the data. There is a potential that all datasets needs to be searched anyway, if the value searched for is presents in all datasets at some point. By first parsing the index meta information there could be an indication given on how many entries needs to parsed in the first place, allowing flexible adoption of the search query or notification to the operator that the query might take a long time.

8 Filtering

Roughly said there are three types of filtering. The resolution of the data can be reduced, the data can be removed from the dataset or it can be corrected.

8.1 Reduction

Within the world of filtering the incoming sensor data there are big considerations to make with regards to storage and computing power. Most ideally all sensor data is kept, however this sometimes renders to quite some large number of data points. So decisions has to made to for example aggregation, basically reducing the resolution of the dataset. An example of such reduction could be found at Ganesan2003 [9] and using database storage like RRDTool [10].

There is one catch with new kind of sensor storage, since the question is usually asked when all data is gathered it could very well be that not all data is available any more. Which makes the dataset useless for answering this question. For this purposes it is proposed to store all data regardless of the usage required. This bulky data transfer could be somehow overwhelming at first glance, luckily storage is no longer an issue with the storage strategy introduced in Section 6. So the virtually unlimited amount sensor data allows to answer more questions, as the resolution is as high as the original source.

The original source resolution is usually decided during design. Keep in mind during design that the resolution of the readings will determine the resolution of the data at a later stage, given the described unlimited storage solution. The aim is to make the resolution as high as possible, allowing to answer as many questions as possible at a later stage. Also consider the resolution number to be an important meta-data property of the data, as it could be used at a later stage to determine the maximum and minimum resolution possible in a fast and efficient way. This is valid for both sensors with fixed intervals and sensors which allow flexible resolutions.

8.2 Removal

There is another important property to consider, when it comes to filtering. Since the boundaries of the sensor are known, as defined in Section 7.1, filtering out

errors could take place. This seems tempting to consider at first glance, however it could cause multiple issues.

By removing the errors it will not be able to answer for example which sensor is defective. A sensor reporting bogus data seems useless at first glance, however it actually is very valuable data. It "tells" that the sensor is still operational, this is very important data to verify the current functionality of a wireless (mesh) network for example.

Secondly if measurement errors are increasing it could mean the sensor is in need of new parts, for example a battery of probe. So the error rate could help us telling if a sensor has been properly serviced.

The sensor information meta-data could also be incorrect, leading to wrong sensor information being discarded.

Removal of error-ed data is not a clever idea, both from a service and analytics point of view, however there is also a valid case of filtering the data at start. Filtering is the process of cleaning the data by removing anomalies of the data. By filtering the raw data during storage, one could avoid processing the data every time. Processing the filtering operation every time could potentially result in spending large amounts of time doing the same work over and over again in different implementation of the queries of the data.

Since removal the data is a bad thing and not cleaning the data as well is also a bad idea, it seems stuck in a loop-hole. The solution in duplications of data if filtering needs to take place, instead of removing an entry create a new "key=value" field telling to discard a certain sensor reading. This valuable data is also to be classified in the meta-data in Section 7 such that analytics on error rates could also be simplified.

For values which are to be altered, adding field if the field changed is rather confusing and should be avoided. Having a query program to search for two cases of the data is not efficient at all. Image a query program which for every field needs to check if a corrected value exists is rather in-efficient on lookups. Thus when a value needs to be converted or altered, it is best to consistently re-create the value into a new column or "key=value" pair. For example for the temperature sensor from Section 7 which reported values like '115.12345', where it could only reported in rounded 5 entries, it is wise to add a new field called "temperature_corrected". Do not alter the original data, since this data could be used in different statistics. Since the "*_corrected" fields are also included in the information meta-data it is up to the query program to decide which version to use. This allows maximum

flexibility with regards to later queries.

Different kinds of filtering could be applied to the data, firstly all values of the sensor which are clearly out-of-range could be corrected. For example a temperature sensor reporting values in Kelvin could not never report below zero[6]. If it does report itself below zero the value could be corrected to the minimum level for example. In this case it is better to discard the value as the sensor is clearly reporting faulty information.

Sometimes sensor looks valid at first glance. For example a speed sensor reporting $28m/s$ (roughly $100km/h$) a time-stamp $x + 1$ could be a perfect valid reading. However the delta of the reading might not be valid. If sensor was reading $0m/s$ a second ago (x), it would give the object an almost impossible acceleration. As a consequence the partial sensor entry of $28m/s$ at $x + 1$ should be discarded.

Analytic and correction of this sensor details as described above could only be done if accurate meta-data of the sensor is available. This meta-data could be provided by the sensor, but could also be provided by the relations of the sensor with it surroundings. A speed sensor on a container ship should never have acceleration (positive or negative) of a certain kind and should also not be able to go any faster than it possible by the physical properties of the ship. This meta-data with regards of the surroundings should preferably be inserted when the data is still accurate and available.

8.3 Correction

Sometimes a set of date end up with gaps in the data as a sensor might be stalled or unreachable for various reasons. This will cause the sensor to start dropping data if its (limited) buffer is no longer able to store the previous readings. When dealing with data analytics it is sometimes required to try to restore the data to be able to find out what might have happened during a certain time period. The most trivial way of recovering the missing data is to "draw" a line between the last known entry and the first known entry and try linear interpolation to reconstruct the missing. There are far more advanced methods available to be able to reconstruct missing sensor data. Some are using extensive data-mining [11] others using neural networks [12] or auto-associative regression machines [13].

[6]If it does and your measurement validates, you have redefined the absolute zero temperature point.

Please do mind that reconstructing data is, from a sensor point of view, the same principle as filtering the data. So make sure not to replace the data, instead create a new column and make sure to mark the data as corrected as described in Section 8. The same principles apply over here, knowing that sensor data was not generated is very valuable data for different analytics tooling.

For example during analytics it might be needed to assign weighting or probabilities to the sensor readings gathered. This is required to assist in analytics and automatic relation discovery. When for example multiple sensor readings are combined and two different sensors are both reading the same environmental value. The "corrected" sensor readings could be treated with a lower confidence, allowing "real" sensor to be preferred during analytics or automatic relation discovery.

9 Analytics

There are multiple types of analytics to be done on the data. The first kind of analytics involves looking at single columns. This analytic could for example generate histograms and distributions of the numbers present. This allows discovery of the space used by a sensor. It could for example also be used to answer questions like; "What is the error-rate of a sensor?". This question assumes that the meta-data for the sensor is already present. A second type of analytics will take multiple columns and tries to find a correlation between the columns.

9.1 Data Identification

All cases now considered have the properties of meta-data fully presented and defined. But what happens when this meta-data is not present or invalid. The example of Figure 4 at page 15 is such example. When the identification is unknown ways have to be found to add the right label to the data.

There are numerous problems arising from this subject, ranging from duplicate data definitions and an impossible amount of format. There is one key crucial for most identification purposes. Knowing the time-stamps would be a very nice start. Timestamps is defined as an entity which uniquely defines a point in time relative to a certain fixed data, time entry. Timestamps classification could be done in various ways.

When it comes to identification a reference point would be a great point to start. For example if the data is still arriving, checking of the incoming fields against the current time could give a nice pointer about a possible column identifying the time. Do mind that this could very well be a compound field as seen in Section 9. The best way is to get a number of samples and verify its suspected time-stamp column is an increasing values at rates which the sensor is expected to report.

Sometimes the column time-stamp could also be detected automatically. The time-stamp column is a field which is always increasing and never resets, blindly assuming the clock will not be reset or out-of-sync at some point and that the reports arrive in chronological order..There might be a number of other fields which could also have the same characteristics. If one characteristic is a 1:1 relation to the local time, then you are sure that the field is time. So when applying the logic as described there might be two or more fields matching the description. Now comes an interesting case as multiple entries could in theory describe the time-stamp column. To classify the right column, we find out what the difference is between the entries in the potential columns. If this difference is constant, both fields will describe the time only with a different base value. When a difference is found it gets more troublesome to find out which entry is in fact the time-stamp column. The best way will be to take a look at the differences. Take for example the two rows found in Table 2. By looking at it "A" seems the best candidate for the time-stamp column. However "B" also has the unique property and has increasing numbers. In this case a decision could simply not be made, so it will leave no choice to mark both columns as potential time-stamp columns.

A	B
1	1
2	4
3	5
4	6
10	10
20	15
21	16

Table 2: Two columns which could potentially be the time-stamp column.

Even with more potential candidates for the time-stamp column there might be an option to find the right one, if some information is known about the sensor information. For example that one of the sensors is the distance travelled in km. This allows to use both potential time-stamp columns to calculate the speed the

object is travelling. If the speed is at ranges beyond physical properties of the object, it it safe to assume that the column could be eliminated as potential time-stamp column.

Knowing which sensors should have existed in the first place could also be a trigger to find out the correct relation. In this case assume knowledge about the fact there is at least one sensor which reports the distance travelled in km and there is also a sensor reporting the speed in km/h. This extra information could provide us with the answer, as the time-stamp calculation could be derived from this values.

Next comes pattern recognition, as time-stamps are specified in numerous formats, to list a few:

- "Wed 11 Sep 16:39:00 CEST 2013"
- "2013-09-11T16:39:48+0200"
- "5 minutes from 5 Jan 2013"
- "1378910479" (epoch)

By using pattern recognition it might be possible to find the column which represents the time-stamps.

The three logical deductions are potentially very hard for a program to make, thus automation is only possible in a limited set-up. Sometimes it is best for an operator to look at the data itself and see if it could help the program by adding the meta-data or adding new recognition patterns.

An example to identify the time-stamp column based on its unique properties is found in Appendix D.

9.2 Relation Discovery

Correlation finding between data is important to find out if certain events are related to each-other. For example one would like to know if a temperature sensor is related to the location. By splitting the destination space into smaller spaces you could "bin" the temperature readings to specific areas. For every area "bucket" you

could see if data is related to each-other. A positive match would give a correlation between the space and the temperature.

Unfortunately it might not be that simple. Which gives the 3rd type of analytics. The temperature could for example be related to multiple columns. Both the location and the direction of the ship would yield to a positive match. The reason for mentioning direction is that a ship travelling in a certain direction, could place the sensor in a different angle to the sun, making the sensor heat up.

Finding linear relations using multiple tables quickly exploded to an unsearchable amount of combinations of sensors to query. If there are 10 sensor remaining readings besides the temperature is could be as such that every combination of sensor readings could be the relation for the temperature. Since a function is either included or not it yields to an search set of $2^{10} - 2 = 1023$ combinations. The $000\ldots$ case where no function is considered relevant is of course not investigated. This will give a complexity of $O(2^n)$, so searching all sensor combinations will not be possible.

This simple linear relation detection could grow to more levels of complexity when to consider alternative comparison methods. The "binning" approach as described above might not be sufficient to detect the relation, every new combination algorithm would trigger on full round of searching for all possible combinations.

Sometimes relations are not simple and require advanced relations in preprocessing. Imagine a sensor reading which present the delta to certain fixed base time and a second sensor reporting the fixed base time. By adding the values a time-stamp could be created. If not known which values to add together (due to missing database) for example, all columns needs to be added together and presented as new data. Next all new columns needs to be correlated with the current columns to find out if the operation had any effect. Adding such entries makes an explosion in search place.

Another set of entries which are needed are for example related data. If a time-stamp entry is present a new requirement might be to add a new column which represents the day of the month. This way you could for example answer the question if the ship is going faster at certain days of the month. Making this automated translations of the data is considered extremely difficult as there are various properties to consider. First of all the program must be able to understand the question and know what data is required. Secondly it should know which sensor reading to convert the time-stamp reading to a new index, namely day of the month.

Lastly another field of sensor data is using combined sensor readings to provide one result. This could be done by combining sensor readings at a fixed-location An example would be to detect human presence by combining sensor readings of a light, motion and sound sensors [14]. An alternative would be to combine sensor readings from multiple locations taken at the same time. An example would be temperature readings taken at the same time from different locations inside a rain forest to make predictions on forest fires [15].

Finding relations in the data is a process which follows an iterative approach. First a question is raised, next the data is found and the results are presented to the operator. If the result is sufficient the operator present the results, else the operator adjusts the knobs to generate new results from the dataset. The repeating nature of the process, makes it a good fit for a real-time analytics tool. However handling large amounts of data in a real-time fashion is somehow hard to-do.

Gathering statistics of temporal and spatial is not a problem as from a data point of view this could be considered a very big table of entries. There is no problem loading the data into the statistics programs. The output of this statistic analytics however could sometimes be disappointing, since the data is displayed in a table or as a list of entries. This makes it difficult to interpret. There would be a better added usage if the values could be plotted onto a map, with method like heatmap plots.

> A heatmap is a thematic mapping technique in which a (typically) diverging, (usually) thermal-like colour scheme is used to represent density in a continuous fashion.
>
> `https://en.wikipedia.org/wiki/Heat_map`

Heat-maps are technically generated by plotting (filled) circles around a centre point with a decreasing radius. The alpha is increased as the radius goes bigger. And the colour is scaled (linear) from a primary colour to another primary colour. For example from red to green. Thus every heatmap point is constructed as seen in figure 6. The process of filling a circle is not a trivial task for an computer (for this matter it could better to use squares) as it first need to calculate which blocks to fill and next.

Being an lengthy process it is not useful for embedding in real-time analytics

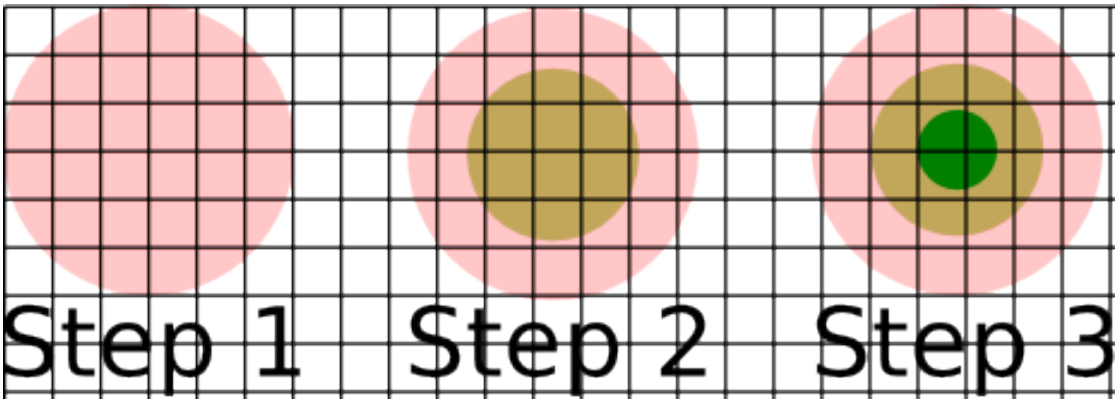


Figure 6: Build-up of a heatmap dot in steps

programs, making the operator wait a long time for the visualisation of the results becomes available. For this matter none of the analytics tools include live visualisation of mapped data.

10 Existing Solutions

There are many different solutions on the market already claiming they can handle large amounts of spatial temporal data. The solutions use different methods to provide analyse or automation relation discovery. This section will provide an overview of the available tooling and methods used.

10.1 Tooling

Numerous amount of tooling is available for data analytic. A few big commercial players in the market are MatLab[7], IBM SPSS[8]. From an open source point of view it is worth looking at PSPP[9]. PSPP is open source implementation of SPSS, featuring most features. A second player strongly presented in the open source world is Rattle[10]. Rattle is a graphical user interface for data-mining using R [16]. One particular useful addition it GGobi [17] tool-kit. It allows the user to click

[7] http://www.mathworks.nl/products/matlab/
[8] http://www-01.ibm.com/software/nl/analytics/spss/
[9] http://www.gnu.org/software/pspp/
[10] http://rattle.togaware.com/

around and view relations of data, for example one as found in Figure 7. By (manually) scanning through the graphs one could quickly get an idea of the data relations present.

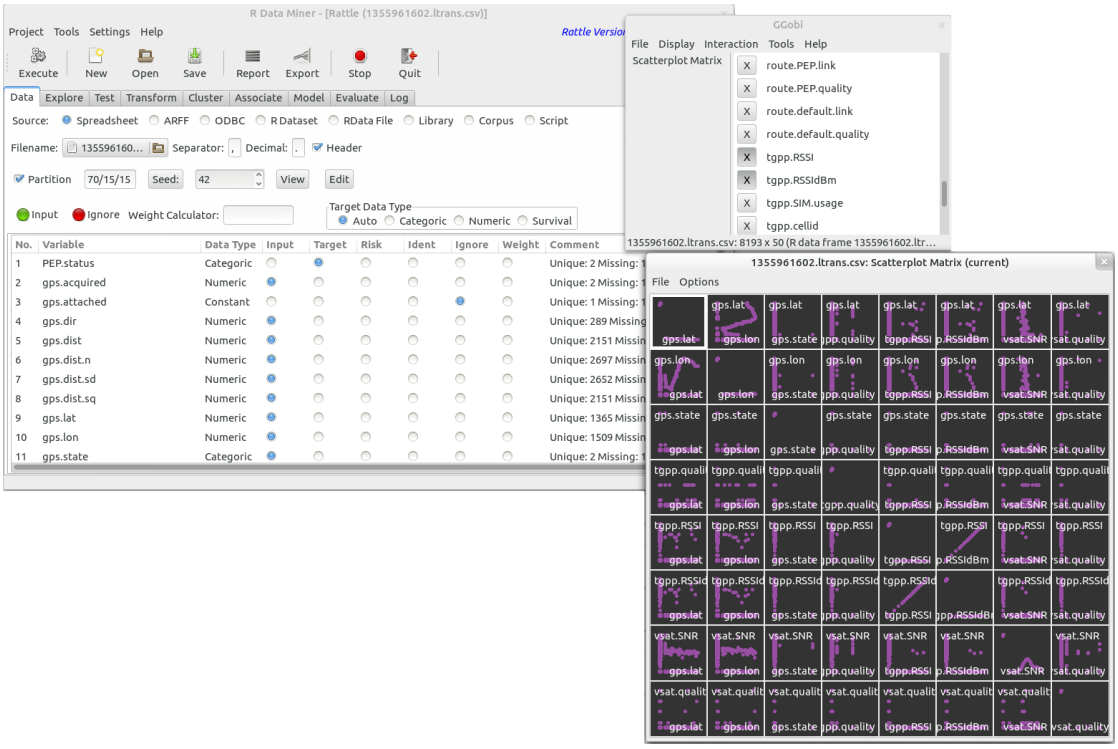


Figure 7: Rattle running in R, showing the automated deductions made on the data and an GGobi matrix window showing relations between different parameters

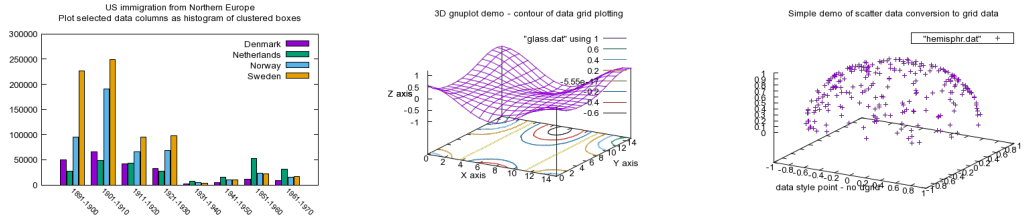
Rattle and friends have one big limitation when it comes to interactive discovery using visual thinking, there is no mapping possible to a surface map, with for example heatmap visualisation. The matrix visualisation included however provides a powerful way inspect the data dynamically. The build-in data classification methods seems redundant if proper meta-data descriptions are available.

Next comes the tooling designed for automated relation discovery within large datasets. A few big Open Source players in this field are SCaViS (Scientific Computation and Visualization Environment)[11], KNIME (Konstanz Information Miner)[12], RapidMiner[13], UIMA (Unstructured Information Management Architecture)[14],

[11] `http://jwork.org/scavis`
[12] `http://www.knime.org`
[13] `http://sourceforge.net/projects/rapidminer`
[14] `http://uima.apache.org`

Weka (Waikato Environment for Knowledge Analysis)[15] and ELKI (Environment for Developing KDD-Applications Supported by Index-Structures) [18].

This tooling focuses on various aspects of automated relation discovery, with assisting visualisation tooling and profiling tools. It will for example interactively visualize data in 2D or 3D plots. Provide histogram, contour en scatter plots, examples of those could be found in Figure 8.



(a) histogram (b) contour (c) scatter

Figure 8: Plot style examples by Gnuplot

10.2 Methods

When it comes to automated discovery and analyse the available tooling consists of different techniques to enable such discovery. For every "main" method there are many more alternatives with specific improvement areas or specified for a certain application. This makes them currently difficult to apply them in a generic matter to untagged spatial temporal data. Highlights of the ones I find most applicable for spatial temporal sensor data analytics are described in the following chapters. There is special focus on limitations for deployment within automated relation discovery. The limitations will be used in Section 11 to explain challenges in feedback loop implementation.

10.2.1 Neural Networks

Neural Networks [19] are an artificial representation of a neural network, capable of "learning" a response given a set of inputs. This is done by adding a (few) hidden layer(s) of nodes between the input layer and the output layer as seen in Figure 9. All nodes have a certain weight and a trigger level, on which the values are

[15] `http://www.cs.waikato.ac.nz/~ml/weka`

"learned" by applying a test-set of data. After being initialized the neural network is able to make decisions what the answer will be given a set of input variables. This could be used in temporal spatial sensor data analytics by learning known good cases and detections of outliers for example. The downside is the requirement for a known good test-set, which is usually not present at initial stage.

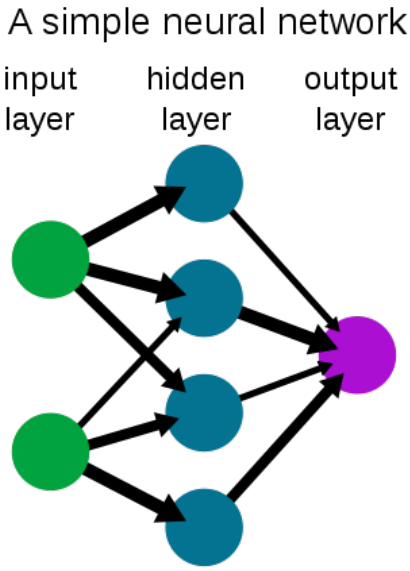


Figure 9: Neural Network with one Hidden Layer

Neural Networks require data to train the network. Within our spatial temporal datasets, training data is not available making it impossible to initialize the network, without help of other algorithms to provide a training set. Secondly since it is unknown which parameters to use as explanatory variables, making a choice and variant leads to an explosion of the search space.

10.2.2 Linear Regression

Linear Regression aims to find a relation between *dependent variable* and one of more *explanatory variables*. A dependent variable represent (or is tested whether it is) the output of effect, where-as an explanatory variables represents (or is tested whether it is) the inputs or causes.[16] For example in Figure 10 the x-axis contains the explanatory variable and the y-axis contains the dependent variable. Depending on the choice of variables this could also be reversed. When relations are found

[16] `https://en.wikipedia.org/wiki/Explanatory_variable`

between the two variables, this relations could be used to predict further variables and could also be used to calculate missing variable. Note that the linear regression aims to provide a fitting curve, making all points fall on this line, making the error difference for every point as small as possible, however this might not always be the case as seen in our example. Linear regression finds good use in fitting possible relations within spatial temporal sensor data, by using this method to detect linear relations between the variables.

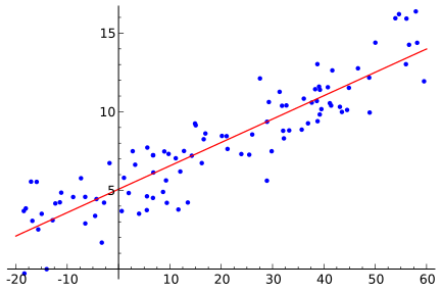


Figure 10: Linear regression which has one explanatory variable

Finding linear regression relations between data works well for initial discovery when used within overviews like the one with Rattle in Figure 7 on page 39. Automatic Analytics of Linear Regression analytics over all columns, does not work well. With automatic recognition it is unclear how the data itself relates to each-other. The reason for this limitation seems to be caused by the so-called Anscombe's quartet as seen in Figure 11[17].

10.2.3 Curve Fitting

Curve Fitting [20] is an application which uses a polynomial equation to try to fit polynomial curves. For example for a 4^{th} order polynomial equation is shown as $y = ax^4 + bx^3 + cx^2 + dx + e$. By choosing a, b, c, d and e accordantly one could be able to fit the curve on the points available. More orders within the polynomial equation means that a better mapping could be found. A example is shown in Figure 12[18].

There are multiple algorithms which can calculate the constants such that the polynomial curve fits on the points available, an example of such algorithm is Gauss-Newton method [21].

[17] `https://en.wikipedia.org/wiki/Anscombe's_quartet`
[18] `https://en.wikipedia.org/wiki/Curve_fitting`

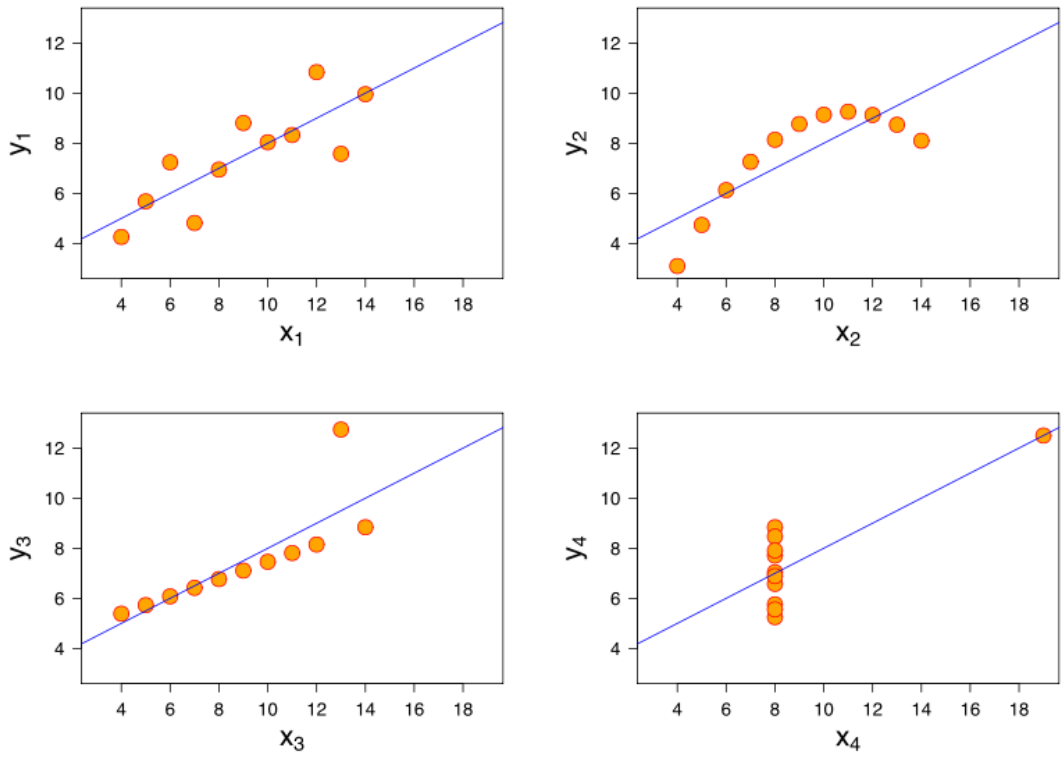


Figure 11: These sets are identical with regards to linear regression when examined using simple summary statistics, but vary considerably when the individual points plotted.

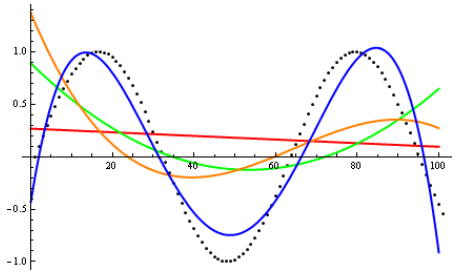


Figure 12: Polynomial curves fitting points generated with a sine function. Red line is a first degree polynomial, green line is second degree, orange line is third degree and blue is fourth degree

One implementation of Gauss-Newton could be found in R [22], with the function **nls** [23]. An implementation example could be found at Appendix L, which produces results as seen in Figure 13.

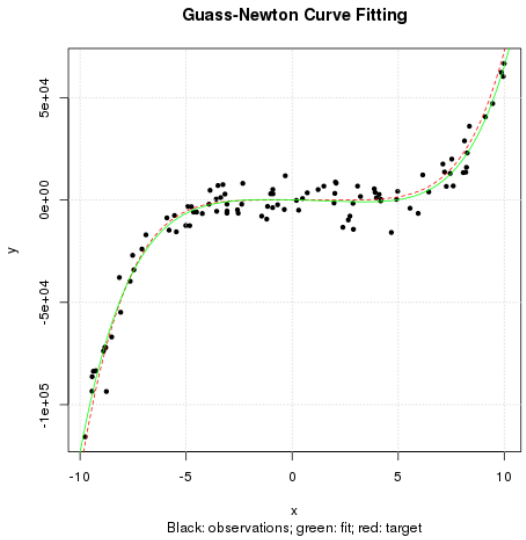


Figure 13: Gauss-Newton algorithm finding $x^5 - 3 * x^4 + 3 * x^3 - 2 * x^2 - 5$

Curve Fitting has limitations: it is time consuming to try fitting two parameters with each-other, when doing automatic analytics. Trying to correlate all variables becomes an endless job. There are multiple approaches to improvement of the process one worth noting is "Beyond Eyeballing: Fitting Models to Experimental Data" [24].

10.2.4 Cluster Analytics

Cluster Analytics is the process of mapping n observations into k clusters where each observation belonging to one cluster with the nearest mean (K-means clustering) If observation could be part of multiple clusters it is called Fuzzy C-means algorithm [25]. K-means clustering also has it variants which preform better on certain datasets. For example EM clustering [26] which has benefits over K-means clustering as shown in Figure 14[19].

The clustering methods finds it usefulness if the sensor data is distributed over an area with both the x-axis and the y-axis, if the data is concentrated over a path, such as a trajectory of a ship the results are not useful, see Figure 15 for an explanation.

There are many more variants of clustering defined, however no clustering

[19] `https://en.wikipedia.org/wiki/K-means_clustering`

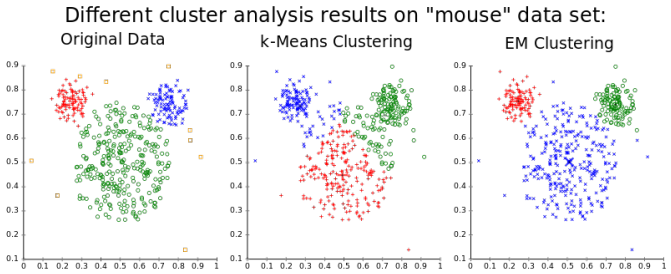


Figure 14: k-means clustering and EM clustering on an artificial dataset ("mouse"). The tendency of k-means to produce equi-sized clusters leads to bad results, while EM benefits from the Gaussian distribution present in the data set.

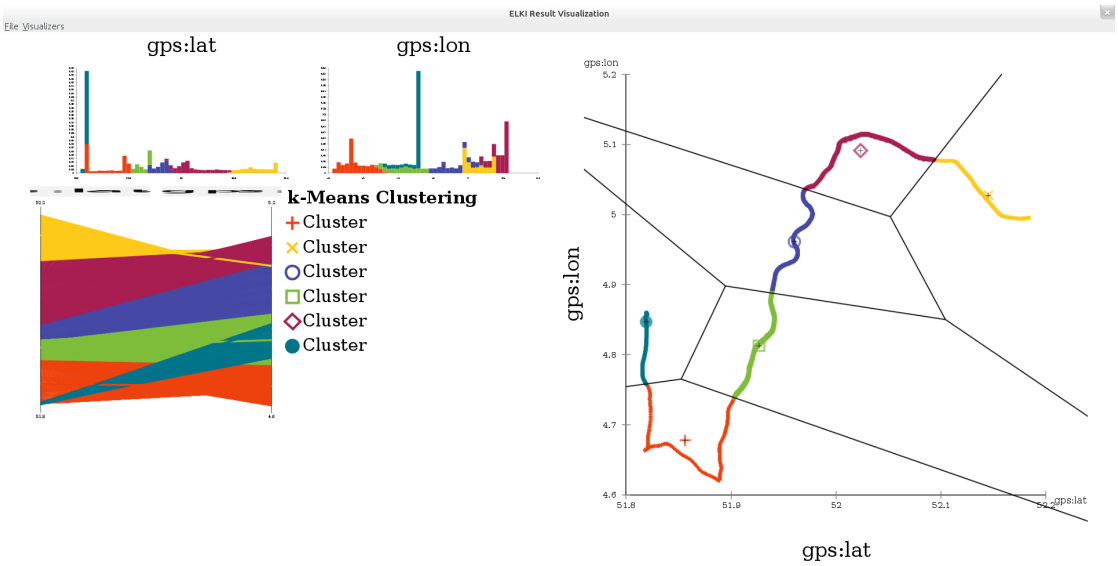


Figure 15: K-means algorithm with 6 sections, by the ELKI tool-kit, shows the path spliced in 6 parts, there is however no cluster mapping of the sensor value itself.

method is found implemented by the ELKI Tool-kit[20] which also takes the time element into consideration, so the method finds its use in grouping data collected over a larger area, where-as path and time relations are not considered in this case.

Cluster Analytics is not useful for path data as seen in Figure 15. The clusters either become without context or are not grouping the right values at all.

[20] `http://elki.dbs.ifi.lmu.de/wiki/Algorithms`

10.2.5 Time Series

Time Series are sequences of observations (data points) measured typically at successive points in time spaced at uniform time intervals.[21]

This classification nicely fits to the sensor data, when using time series analysis to find meaningful statistics and relations with regards to time. Such statistics and relations could be finding trends within data. One example is trying to find the frequencies found in signals, by the means of Fast-Fourier Transform (FFT) [27]. FFT comes in wide variety of implementation ranging from simple complex-number arithmetic to group-theory and number theory.[22] Because there are so many variants of FFT around, selecting the proper one for the job is not so easy, especially when the data structure is not known in advance, which is usually the case with sensor data.

10.2.6 Autocorrelation

Autocorrelation is the cross-relation of a signal with itself. Sensor reading are a discrete set of readouts and no continuous signal anymore, due to the sampling as seen in Section 2. However most original sensor sources used to be a continuous signal, take for example temperature and direction readouts. With autocorrelation we could find repeating patterns within a certain source. This could for example be useful to find out if a sensor readout is for example following some periodic behaviour. Note this is vastly different from the other types described over here, which involves finding relations between two or more sensor readings, this is strictly looking at one readout and tries to learn more about it.

Autocorrelation is limited by finding sequence with regards to time. There are efforts [28] to apply the same field on spatial sequences, allowing to find repeating sequences on a 2D spatial field, however no implementation has found.

Time Series Analytics using FFT or Autocorrelation tell which frequencies are present within a sensor, this provides a great deal of more meta-data potentially allowing data-analytics to be more easy.

The raw sensor readings has a fair amount of noise, which needs to be filtered in order to find out frequencies within the data. The noise might be responsible

[21] `https://en.wikipedia.org/wiki/Time_series_prediction`
[22] `https://en.wikipedia.org/wiki/Fast_Fourier_transform`

for adding all kind of bogus frequencies in the output. Filtering noise however is troublesome if the source type is not known. Noise filtering to detect human sound [29] is different from removing noise from an image CCD (Charge-coupled Device) sensor [30]. One particular field of interest FFT would be to able to detect "natural" rhythms with sensor output, one could for example think of natural sequences as seen in Table 3 or man-made sequences like engine rotation, carrier wave and periodic behaviour like "lights on 8 hours a day".

sequence	Sample applications
day	temperature, power usage, internet usage and light
month	water hight
season	temperature, satellite coverage (clouds)
year	temperature, yearly-activities

Table 3: Some examples of periodic patterns to be found in sensor data

10.2.7 Cellular Automata

Cellular Automata is a somehow unexpected member of this list.

> A cellular automaton consists of a regular grid of cells, each in one of a finite number of states, such as on and off (in contrast to a coupled map lattice). The grid can be in any finite number of dimensions. For each cell, a set of cells called its neighbourhood is defined relative to the specified cell. An initial state (time $t = 0$) is selected by assigning a state for each cell. A new generation is created (advancing t by 1), according to some fixed rule (generally, a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighbourhood. Typically, the rule for updating the state of cells is the same for each cell and does not change over time, and is applied to the whole grid simultaneously, though exceptions are known, such as the stochastic cellular automaton and asynchronous cellular automaton.
>
> `https://en.wikipedia.org/wiki/Cellular_automaton`

Cellular automata have a notion of both space and time, in particular the two-dimensional cellular automata. This makes them an interesting fit for modelling behaviour within spatial temporal sensor data. There is however no analytic component, such that the tooling could only be used for testing models.

Cellular Automata are a nice theoretical fit for modelling spatial temporal relations, a example is found in modelling spatial and temporal processes of urban growth [31].

11 Human-assisted Automated Relation Discovery

An implementation is made for testing Human-assisted Automated Relation Discovery as described in Section 2. This is done by creating a set of tools which allows to operator to quickly view the data. Since the data is already stored the focus is given on exploring the data and providing visualization with a real-time interface for the operator.

For rapid purpose prototyping the scripting language Python[23] on a Linux Fedora 19 (x64) Intel Core i7 laptop is used, the systems has 6GB of memory and 4 CPU cores. With this setup, most recent computers should have no issues running the provided code.

Heatmap generation as visualization concept is interesting when combined with interactive usage, allowing to show new images in less than 5 seconds for 1000 heatmap points. How-ever existing traditional implementations to generate heatmaps like matplotlib [32] and gnuplot[24] where to slow. The heatmap gradient points, as described in Figure 6 on page 38, are complex to generate as shown in Figure 16.

The attempts of optimization of the code by trying to fully handle the heatmap generation code ourself instead of plotting it via the "python-gnuplot" or "python-matplotlib" API turned out to be still too slow. The biggest limitation with all the implementations came with the fact that it is an expensive operation to draw and fill a circle within an array, due to the fair amount computation needed to find the pixels of which the circle consists.

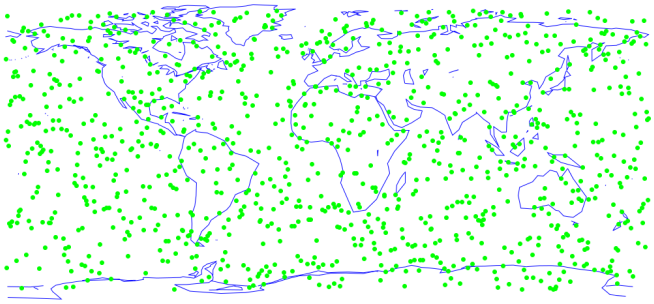
The solution came from a rather unexpected angle, namely the "game" world. Within the "gaming" industry it is rather common to draw circle, so this feature has been implemented within the GPU (Graphical Processing Unit) as standard operation. This hardware acceleration allows faster printing of circles on a "canvas".

A python module called pygame[25] which allows direct access to the advanced drawing engines by the GPU [33]. The next nice feature is the fact that pygame allows us to update parts of the screen, which making drawing circles with our advanced alpha profile really fast, as we do the drawing of every dot localized.

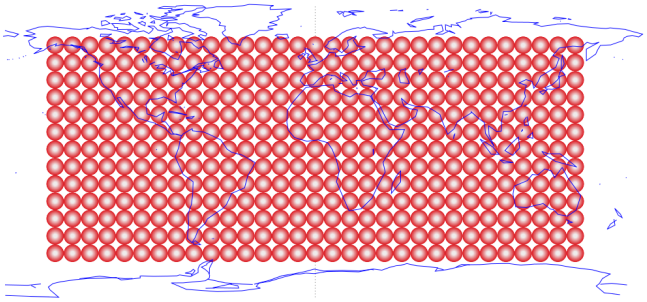
An implementation of heatmap generation can be found in Appendix F. The next thing needed was a parser for the `LTRANS` data. This is shown in Appendix G.

A heatmap is interesting, however plotting a heatmap on a blank canvas is somehow disappointing as seen in Figure 17a. It would be better to overlay the heatmap over a map of the world. For generation the base-map comes the package matplotlib to the rescue[26], this tool-kit includes a map generator, which is fast enough for our usage.

[23] `http://python.org`
[24] `http://gnuplot.info`
[25] `http://www.pygame.org`
[26] `http://matplotlib.org/basemap/users/geography.html`



(a) 1000 Random dots generated in $0.01s$, code in Appendix J

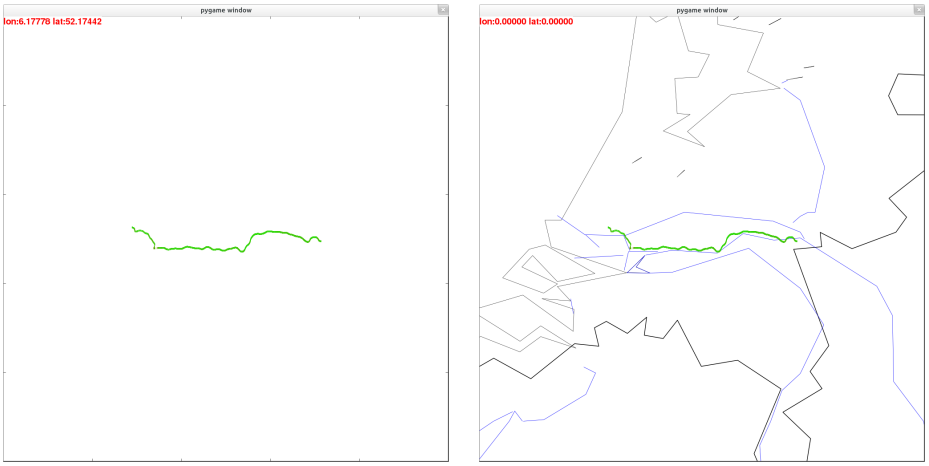


(b) 330 Gradient dots generated in $5.95s$, code in Appendix K

Figure 16: It takes a lot more time with gnuplot to generate gradient points instead of solid filled circles, as a heatmap gradient circle is drawn using many circles in ones.

The combination of matplotlib and pygame gives us a nice base-map and a heap-map plotting in real-time, the path from above combined with a base-map gives the result as seen in Figure 17b.

The base-map has 5 resolutions ranging from low to high: crude, low, interme-



(a) Plotted sensor path on a blank canvas.

(b) Plotted sensor path with map, shows it is travelling over the river.

Figure 17: The difference between a plot without and one with a base-map.

diate, high and full. The difference in detail is shown in Figure 18, timing of the various resolutions is found in Table 4. When using the base-map for interactive usage it is best to set resolution to "low" allowing fast responses for interactive pictures.

resolution	time used
crude	$0.254s$
low	$0.641s$
intermediate	$2.555s$
high	$9.224s$
full	$54.480s$

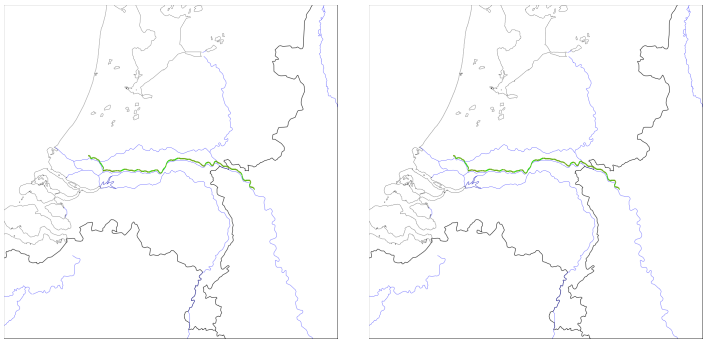
Table 4: Time required to generate the base-map images

When looking at Figure 18e some reading does not seems to plotted within the river. The zoomed Figure 19a makes this even better visible. When looking at the map data at Figure 19b[27] the river is wider plotted in Figure 19a.

Topological values are usually correct, keep in mind that the world, changes so the data might be outdated. Secondly sometimes there are errors deliberately introduced within card information to serve as Copyright Trap[28].

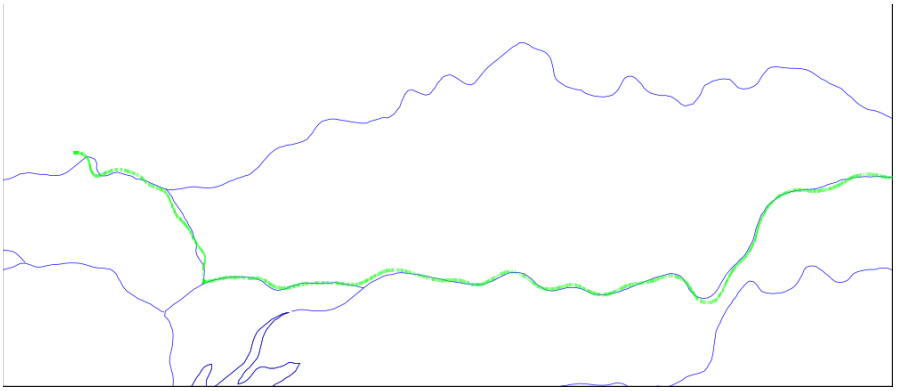
[27]Data by OpenStreetMap.org contributors under CC BY-SA 2.0 license.
[28]`http://en.wikipedia.org/wiki/Fictitious_entry`

(a) crude (b) low (c) intermediate

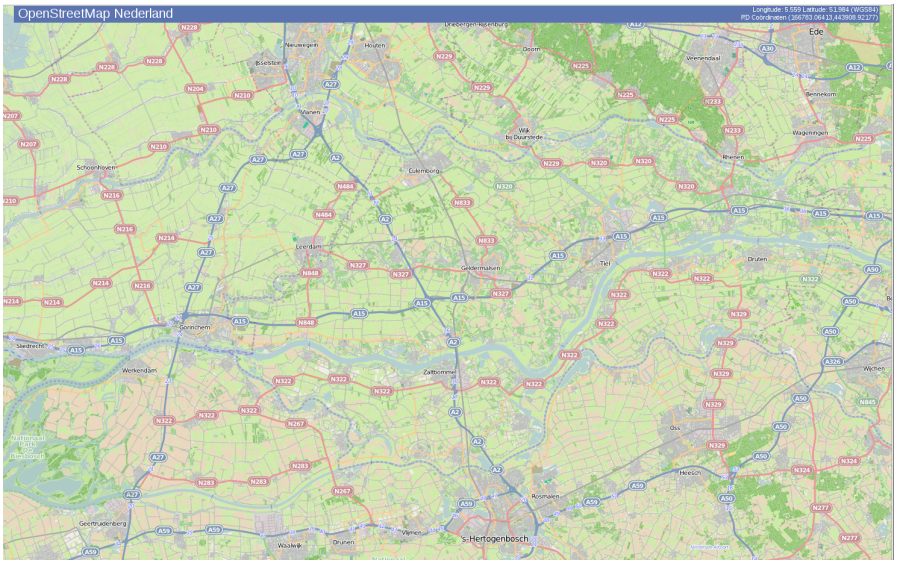


(d) high (e) full

Figure 18: The difference between the various resolutions of the base-map



(a) Zoomed version of Figure 18d, where all readings are in green and without transparency to highlight the path.



(b) Map visualisation by OpenStreetMap.org focusing on a part of the area of Figure 19a, observe the variable width of the waterways present.

Figure 19: The path showing its offset and a map showing a part of the track

As this mismatch is appearing at multiple locations, it will rule out topological errors, so the error must exist in the sensor or in the display of it.

There are no readings or reference sensor data available about the distance between the ship and the shore, leaving several explanations available. The ship might not always be travelling in the centre of the river which partially explains the offset. Another possible cause could be inaccurate sensor readings, positioning the ship at the wrong location. GPS sensors for example might auto-correct readings and assume the object is travelling in a straight direction. When a turn is made it

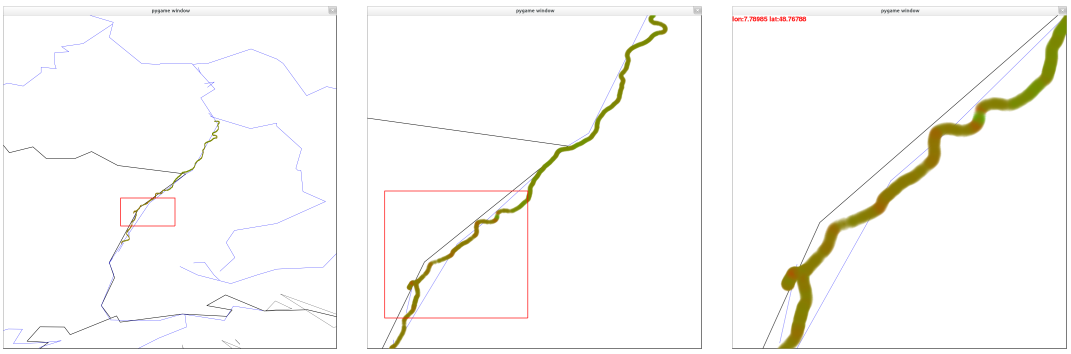
requires some time before this movement is detected and included into the sensor readout.

Both assumptions could be validated and explained by including more sensor data. Since this data is not available, an answer to the question what is causing the offset is not possible with the current available sensor readings.

The next experiment aimed to provide insights on the connection loss issues, as described in Section 2. By calling the program:

```
$ ./src/ltrans-parser.py --watcher vsat:SNR --heatmap <LTRANSFILE>
```

with a dataset to analyse and a parameter to display, in our case `vsat:SNR` which represent the satellite connection quality, we generate the picture as shown in Figure 20a.



(a) Zoom level 1 (b) Zoom level 2 (c) Zoom level 3

Figure 20: Signal level where red is bad and green is good seen from various zoom levels.

Our picture shows some interesting red dots, which indicates a loss in connection. By zooming in we could directly view the hotspots in Figure 20b. Zooming in one step more gives us Figure 20c. In this picture it is shown that the red dots focus around corners within the pad. This gives an indication that it might be related to activities caused by the rotation of the ship.

One of the activities involving rotation and satellite is the Maritime *VSAT*[29] (Very-small-aperture terminal). This piece of equipment comes as a satellite dish equipped with two motors which are used by the satellite guidance system to keep the dish aimed to the satellite in use.

[29] `https://en.wikipedia.org/wiki/Very_small_aperture_terminal`

Maybe the ship satellite guidance system is unable to cope with fast changes by the ship rotation. This causes the satellite connection to get lost as aiming get inaccurate, making the connection unstable. The satellite guidance system will need to be adjusted to receive information about ship movement or their correction algorithms needs to be adjusted to cope with this kind of ships. There is unfortunately no sensor output available for the Maritime VSAT systems to verify this claim.

A river is sometimes pretty wide and the ship could travel on the river on multiple location as seen in Figure 21, so by travelling an alternative path over the river the satellite data coverage could also potentially be improved.

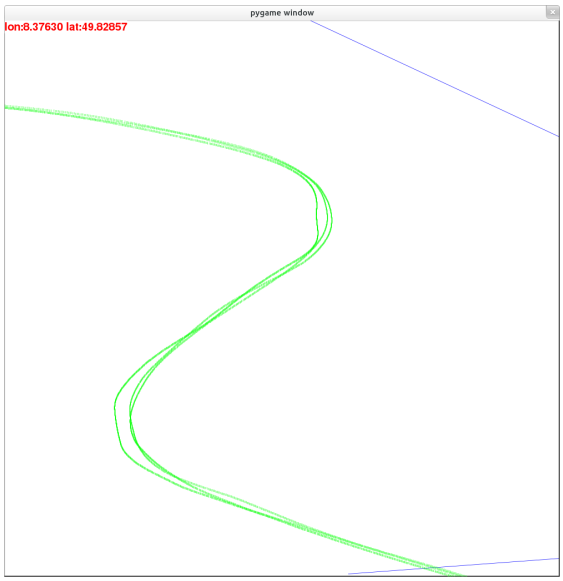


Figure 21: By loading multiple trajectories done by the ship on a specific part of the river, it is show the ship is not always travelling the exact way through the river.

The next "question" was to find out whether it was possible to automatically the time-stamp field in the `LTRANS` data. For this purpose a simple identification program was written as seen in Appendix D. The program was able to detect the "offset" column as identification column.

There are many automatic data analytics classification, relation and analytics methods around as seen in Section 10.2. However the application to untagged spatial temporal sensor data turns out to be challenging. The existing methods are often only able to work under specific cases or require extra annotation or helper parameters to function within the data. An algorithm which is able to automati-

cally select a method and fill in the proper parameters is not yet implemented, due to time constrains. The feedback loop however as seen above has shown practical use, by extending functionality it would be able to cope with more cases.

12 Conclusions

Spatial temporal data is an interesting sub-field of sensor data when it comes to researching sensor data, the data has interesting properties to discover and there are quite some challenges available in visualisation, analytics and automatic relation discovery. The data has a great visualisation potential since it takes places in time and space, potentially making it great research subject for those who like to work with Visual Thinking.

As for the available data, sensor prices dropped and become more and more available, this is leading to an explosion of data, measured from sensor networks. The information gathered is not always storable in a traditional database anymore. New storage and transport formats are being defined and created, coping with the ability to store all this new data feeds. When it comes to transporting data from sources where power is no limitation and transport costs is charged per byte, the `LTRANS` protocol seems to fit nicely. From a storage point of view, the plain text storage seems to come back as an alternative storage solution. This trend is however caused by limitations in current database solutions, new database storage types which focuses on storing a huge amount of key=values entries for a longer period are actively being researched and designed and will see light the next coming years.

However it is crucial to ensure the generated and stored data is annotated with proper meta-data ensuring the date could be analysed at a later stage. The sensor itself however is rarely reporting meta-data about itself, the meta-data should be inserted by either an aggregator or the storage system itself.

Even when this meta-data is not available at first, it could possible be included later. The proposed solution is to use human-assisted automatic relation discovery. Human-assisted automatic discovery will enhances the automatic relation discovery by adding a pre-step which adds extra meta-data to the raw sensor data and a new feature which allows the operator to interact with the automatic relation discovery engine.

This approach has been useful in a small subset of visualization of the data

and providing enhanced details in an interactive program allowing the operator to make decisions.

By using a "game" engine inside the visualisation possibilities on the spatial dimension it became possible to generate near real-time responses, allowing the operator to interactively interact with the data. This feature allows the operator to "hover" through the data providing insights and allows the operator to add his/her expert expertise to the solution.

The feedback loop has been tested in a limited fashion by first classifying the data in the analytics phase and next having the operator tell which types of data to discard and which sensors to further examine.

For the analysis option, there are various interesting features which could be applied to spatial temporal data, such as Curve Fitting, Clustering, Time Series and Numerical Classification. There is however no structured way to storing the results of the analysis done on the data. A basic framework with fields like; type, total, unique, maximum, minimum, count_minimum, count_maximum, changes and choices has been defined, however this list is far from complete. The basic features allows quick search and indexing of the data for access of the data, however there is more need for standard meta-data entries which could be used for the automatic discovery methods.

The biggest lessons I learned came from the field of analytics, as I felt in the trap of thinking that algorithms written are as generic as broad-spectrum antibiotics. The use of analytic tooling however is turn out difficult. Attempts with frequency detection within sensor data using Fast-Fourier Transform (FFT) provided unsatisfactory results, many analytics tooling is written with a certain pre-condition of the data in mind, like reporting at a fixed interval. If such pre-conditions of the data are not known applying analytics tooling is like flying blind as such analytics solutions needs proper tweaking to match the input parameters of the data. Such basic requirements of the analytics tooling should be available even before more advanced analytics can take place. However, a listing of the features required is not yet available.

The various interesting automatic discovery methods for spatial temporal data, such as Neural Networks, Linear Regression, Curve Fitting, Cluster Analytics, Time Series, Autocorrelation and Cellular Automata are implemented into different toolkits, they are however not implemented as a "block". An implementation as a block will potentially allow them to combined together into another solution. Currently if automatic discovery of some kind needs to take place the data has to

be made available in the toolkit and loaded with parameters to make such specific automated discovery possible.

One solution would be a newly to-be-written automatic relation discovery engine which could make use of the provided meta-data by the analytics. This will next select the most interesting automatic relation discovery method on the data and present the result for review to the operator. Attempt to combine multiple analytics into toolkits exists, however they rely on an operator to select the tooling and secondly mostly they do not provide their results appealing for easy interpretation of the operator. Spatial temporal data has great potential for use in such a toolkit due to its visualisation possibilities.

For simple cases it has been possible for the analytics toolkit to provide information to the automatic relation discovery toolkit. This has mostly been done by removing not interesting columns. It has been found possible to annotate the data in such a way that the automatic relation discovery will understand it and use this meta-data information for the decision-making process.

The automatic relation discovery toolkits should be able to interact more with the operator. By gathering its input and including the knowledge in early stages it could mean that results could be generated more efficiently. Manual post-discovery of the data and finding the meta-data and relations automatically seems to be the keyword in processing the sensor data.

The visualisation features as seen in Section 11 should together with existing tools like GGobi bring more knowledge to the operator on the "journey" through the spatial temporal data, allowing the operator to be a powerful extension of automatic relations discovery methods.

The consequence is that toolkits need to become more interactive. The initial steps of interactive data analytics are slowly appearing inside the toolkits, however only small progressions are made for the special purpose applications like temporal spatial sensor data.

13 Further Work

There are various analytics methods available, but algorithm which combine the algorithms for efficient execution are lacking. A Swiss-army-knife solution for adding various meta-data properties in a highly automated fashion is yet to be invented.

This seems a rather new field as not knowing which meta-data to add with analytics is a new area in database research. Previously it was known in advance which question was to be answered. Answering the question: "I have raw-sensor data, of which I do not know any properties, please add as many meta-data you can find to it" is a new type of question to be answered in the interesting field of sensor data.

Analytics toolkits are discarding meta information provided in advance. This means that there are various passes required by the analytics tools to come to "obvious" conclusions. For example if columns are marked as constant data there should be no need to find relations with this columns since there is nothing to relate to.

The interface between the analytic results and the automatic relation discovery is very loosely coupled. There exists no generic format of storing the results of the analytics in such way that they could be re-used by various automatic relation discovery entities. A reference example is provided, further extention is required to fit all meta-data types.

Also toolkits are missing visualisation of data towards world map views, with the ability to zoom and "play" with the data in a real-time fashion, implementations for static visualisation exists such as ggmap[30] for the R Project for Statistical Computing and basemap[31] for Python matplotlib module. The dynamic features however are not implemented yet.

The same limitations as described to the analytics methods, the automatic combination of different methods using an algorithm, applies to automated relation discovery. Relation discovery methods usually tackles a (very) specific area. An algorithm which tries to find all relations within data, using a combination of the existing algorithms as building blocks for example, is not yet defined.

Research with the field of automated relation discovery in Big Data seems interesting to be applies to sensor data as well. However sensor data is more difficult with regards to automated relation discovery. Sensor data has more unknowns like for example missing header information. Secondly the quality of the data is also not known. Innovation in automated relation discovery for sensor data should provide us also more insights on howto handle databases with similar structures and properties.

[30] `http://cran.r-project.org/web/packages/ggmap/index.html`
[31] `http://matplotlib.org/basemap/users/geography.html`

The nature of the sensor data, makes the algorithms and implementations more robust and also allows them to be applicable to bigger areas. This makes sensor data a very interesting candidate for research in automated relation discovery.

Spatial temporal data has more benefits over Big Data. Big Data is usually stored in large databases of corporations and is sometimes hard to acquire or share. Spatial temporal sensor these days is easy to create since it usually applies to day-to-day details, like temperature and humidity. Sharing the large datasets is not an issue, making the datasets great candidates for extended research purposes.

There are also many-and-many specialized algorithms available for automatic discovery or specific analytics. Most algorithms assume the data to be a certain format. However there is less consistency found when trying to combine them. Normally an algorithm will be reimplemented for every new implementation of the algorithm, a modular approach of automatic relation discovery methods and analytics methods should make it more easy to combine methods and implement new methods when needed.

The proposed sensor transport and storage solution works well for discrete sensor observations. It is not efficient for continuous observations like video and sound recordings. Improvements to the protocol and storage setup are required to make this possible.

References

[1] R. Arnheim, *Visual thinking.* University of California Pr, 1969.

[2] M. Chui, M. Löffler, and R. Roberts, "The internet of things," *McKinsey Quarterly*, vol. 2, pp. 1–9, 2010.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] D. Niyato, E. Hossain, and A. Fallahi, "Sleep and wakeup strategies in solar-powered wireless sensor/mesh networks: Performance analysis and optimiza-

tion," *Mobile Computing, IEEE Transactions on*, vol. 6, no. 2, pp. 221–236, 2007.

[6] M. Li, "Data management and wireless transport for large scale sensor networks," 2010.

[7] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace." RFC 4122 (Proposed Standard), July 2005.

[8] O. Rodeh and A. Teperman, "zfs-a scalable distributed file system using object disks," in *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pp. 207–218, IEEE, 2005.

[9] D. Ganesan, D. Estrin, and J. Heidemann, "Dimensions: Why do we need a new data handling architecture for sensor networks?," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 143–148, 2003.

[10] T. Oetiker, "Rrdtool," 2005.

[11] L. Gruenwald, H. Chok, and M. Aboukhamis, "Using data mining to estimate missing sensor data," in *Data Mining Workshops, 2007. ICDM Workshops 2007. Seventh IEEE International Conference on*, pp. 207–212, IEEE, 2007.

[12] D. Dong and T. J. McAvoy, "Nonlinear principal component analysisbased on principal curves and neural networks," *Computers & Chemical Engineering*, vol. 20, no. 1, pp. 65–78, 1996.

[13] S. Narayanan, R. Marks, J. L. Vian, J. Choi, M. El-Sharkawi, B. B. Thompson, *et al.*, "Set constraint discovery: missing sensor data restoration using autoassociative regression machines," in *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, vol. 3, pp. 2872–2877, IEEE, 2002.

[14] L. Gu, D. Jia, P. Vicaire, T. Yan, L. Luo, A. Tirumala, Q. Cao, T. He, J. A. Stankovic, T. Abdelzaher, *et al.*, "Lightweight detection and classification for wireless sensor networks in realistic environments," in *Proceedings of the 3rd international conference on Embedded networked sensor systems*, pp. 205–217, ACM, 2005.

[15] L. Yu, N. Wang, and X. Meng, "Real-time forest fire detection with wireless sensor networks," in *Wireless Communications, Networking and Mobile Computing, 2005. Proceedings. 2005 International Conference on*, vol. 2, pp. 1214–1217, IEEE, 2005.

[16] G. J. Williams, "Rattle: a data mining gui for r," *The R Journal*, vol. 1, no. 2, pp. 45–55, 2009.

[17] D. F. Swayne and A. Buja, "Exploratory visual analysis of graphs in ggobi," in *COMPSTAT 2004Proceedings in Computational Statistics*, pp. 477–488, Springer, 2004.

[18] E. Achtert, H.-P. Kriegel, E. Schubert, and A. Zimek, "Interactive data mining with 3d-parallel-coordinate-trees," in *SIGMOD Conference* (K. A. Ross, D. Srivastava, and D. Papadias, eds.), pp. 1009–1012, ACM, 2013.

[19] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[20] P. Lancaster and K. Salkauskas, "Curve and surface fitting. an introduction," *London: Academic Press, 1986*, vol. 1, 1986.

[21] R. W. Wedderburn, "Quasi-likelihood functions, generalized linear models, and the gaussnewton method," *Biometrika*, vol. 61, no. 3, pp. 439–447, 1974.

[22] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[23] D. Rossiter, "Technical note: Curve fitting with the r environment for statistical computing," *International Institute for Geoinformation Science & Earth Observation (ITC), Enschede (NL),[Online] Available: http://www. itc. nl/personal/rossiter*, 2009.

[24] A. Christopoulos and M. J. Lew, "Beyond eyeballing: fitting models to experimental data," *Critical Reviews in Biochemistry and Molecular Biology*, vol. 35, no. 5, pp. 359–391, 2000.

[25] J. C. Bezdek, R. Ehrlich, and W. Full, "Fcm: The fuzzy c-means clustering algorithm," *Computers & Geosciences*, vol. 10, no. 2, pp. 191–203, 1984.

[26] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 1–38, 1977.

[27] L. R. Rabiner and B. Gold, "Theory and application of digital signal processing," *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1975. 777 p.*, vol. 1, 1975.

[28] P. Legendre, "Spatial autocorrelation: trouble or new paradigm?," *Ecology*, vol. 74, no. 6, pp. 1659–1673, 1993.

[29] J.-l. Shen, J.-w. Hung, and L.-s. Lee, "Robust entropy-based endpoint detection for speech recognition in noisy environments.," in *ICSLP*, vol. 98, pp. 232–235, 1998.

[30] G. Hopkinson and D. Lumb, "Noise reduction techniques for ccd image sensors," *Journal of Physics E: Scientific Instruments*, vol. 15, no. 11, p. 1214, 1982.

[31] J. Cheng and I. Masser, "Understanding spatial and temporal processes of urban growth: cellular automata modelling," *Environment and Planning B*, vol. 31, no. 2, pp. 167–194, 2004.

[32] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, pp. 90–95, 2007.

[33] W. McGugan, *Beginning Game Development with Python and Pygame*. Will McGugan, 2007.

A List of Figures

B List of Tables

C LTRANS example

The data has been formatted by including newlines, wrapping and indentation for readability. It contains samples of various parts of the filename. The original filename is called `1355961602.ltrans`.

AnyWi.com is willing to share the full dataset for other research as well. Terms and conditions could be retrieved from AnyWi.com, by contacting them[32].

```
ltrans=1.0;unitid=00:0d:b9:1b:57:10;timestamp=1355961602
timestamp
0
1;gps:lon=4.85950000;gps:lat=51.81950000;wifi:attached=1;wifi:mode=;wifi:quality=none;wifi:quality:reason=Interface not active;wifi:speed=0.0; \
   wifi:type=IEEE 802.11 Wireless Ethernet;wifi:up=1;tgpp:attached=1;tgpp:IMEI=353567040544331;tgpp:IMSI=204086588008657;tgpp:PIN:status=READY;\
   tgpp:RSSI=23;tgpp:RSSIdBm=-67;tgpp:ccode=NL;tgpp:cellid=30174;tgpp:manufacturer=Conel/Sierra Wireless, Incorporated;tgpp:mcc=204;           \
   tgpp:mnc=08;tgpp:model=MC8705;tgpp:quality=medium;tgpp:quality:reason=UMTS/HSDPA connection;                                              \
   tgpp:revision=T1_0_1_1AP R309 CNSZXL00000015 2011/01/21 18:28:30;tgpp:sysmode=3G;tgpp:syssubmode=WCDMA;tgpp:SIM:ICCID=8931084711060668570;  \
   tgpp:SIM:ccode=NL;tgpp:SIM:ccgroup=0;tgpp:SIM:label=Dekatel NL (KPN);tgpp:SIM:quotum=8053063680;tgpp:SIM:reason=country code;               \
   tgpp:SIM:slot=1;tgpp:SIM:usage=54326395;gps:acquired=1;gps:attached=1;gps:dir=339.00;gps:dist=807903.821193;gps:dist:n=887795;             \
   gps:dist:sd=4.743308;gps:dist:sq=4946286.832725;gps:state=stationary;gps:tms=1355961754;gps:vel=0.000;route:PEP:link=VSat;                \
   route:PEP:quality=high;route:default:link=VSat;route:default:quality=high;route:DNS1:link=3G2;route:DNS1:quality=medium;                  \
   route:DNS2:link=VSat;route:DNS2:quality=high;route:NTP:link=VSat;route:NTP:quality=high
5;vsat:attached=1;vsat:hardware=Newtec;vsat:quality=high;vsat:quality:reason=operational;vsat:SNR=11.2
13;PEP:status=Ok
59;ipstats:vsat:out:packets=7983952;ipstats:vsat:out:bytes=2708771659;ipstats:vsat:in:packets=8053018;ipstats:vsat:in:bytes=5907398602;       \
   ipstats:3g:out:packets=2376128;ipstats:3g:out:bytes=483302980;ipstats:3g:in:packets=2011071;ipstats:3g:in:bytes=884639494;                \
   ipstats:wifi:out:packets=0;ipstats:wifi:out:bytes=0;ipstats:wifi:in:packets=0;ipstats:wifi:in:bytes=0;ipstats:dnsint:out:packets=87881;    \
   ipstats:dnsint:out:bytes=7006425;ipstats:dnsint:in:packets=79212;ipstats:dnsint:in:bytes=11845200;ipstats:dnspub:out:packets=25402;        \
   ipstats:dnspub:out:bytes=2066259;ipstats:dnspub:in:packets=23259;ipstats:dnspub:in:bytes=3593141;ipstats:dnspub:out:2:packets=67513;       \
   ipstats:dnspub:out:2:bytes=5497109;ipstats:dnspub:in:2:packets=67204;ipstats:dnspub:in:2:bytes=10267875;ipstats:dnspub:out:3:packets=0;    \
   ipstats:dnspub:out:3:bytes=0;ipstats:dnspub:in:3:packets=0;ipstats:dnspub:in:3:bytes=0;ipstats:vpn:out:packets=0;ipstats:vpn:out:bytes=0;  \
   ipstats:vpn:in:packets=0;ipstats:vpn:in:bytes=0;ipstats:vpn:ssh:out:packets=0;ipstats:vpn:ssh:out:bytes=0;ipstats:vpn:ssh:in:packets=0;    \
   ipstats:vpn:ssh:in:bytes=0;ipstats:vpn:http:out:packets=0;ipstats:vpn:http:out:bytes=0;ipstats:vpn:http:in:packets=0;                     \
   ipstats:vpn:http:in:bytes=0;ipstats:vpn:https:out:packets=0;ipstats:vpn:https:out:bytes=0;ipstats:vpn:https:in:packets=0;                \
   ipstats:vpn:https:in:bytes=0;ipstats:vpn:pop3:out:packets=0;ipstats:vpn:pop3:out:bytes=0;ipstats:vpn:pop3:in:packets=0;                  \
   ipstats:vpn:pop3:in:bytes=0;ipstats:vpn:imap:out:packets=0;ipstats:vpn:imap:out:bytes=0;ipstats:vpn:imap:in:packets=0;                   \
   ipstats:vpn:imap:in:bytes=0;ipstats:vpn:smtp:out:packets=0;ipstats:vpn:smtp:out:bytes=0;ipstats:vpn:smtp:in:packets=0;                   \
   ipstats:vpn:smtp:in:bytes=0;ipstats:vpn:ipcomp:out:packets=0;ipstats:vpn:ipcomp:out:bytes=0;ipstats:vpn:ipcomp:in:packets=0;             \
   ipstats:vpn:ipcomp:in:bytes=0;ipstats:vpn:expand:out:packets=0;ipstats:vpn:expand:out:bytes=0;ipstats:vpn:expand:in:packets=0;            \
   ipstats:vpn:expand:in:bytes=0;ipstats:vpn:other:out:packets=0;ipstats:vpn:other:out:bytes=0;ipstats:vpn:other:in:packets=0;               \
   ipstats:vpn:other:in:bytes=0
61;tgpp:SIM:usage=54335701 74;vsat:SNR=11.1
100;gps:lon=4.85933333;gps:dir=333.00;gps:dist:n=887895;gps:dist:sd=4.742867;gps:state=moving;gps:tms=1355961854
107;vsat:SNR=11.2
118;ipstats:vsat:out:packets=7983987;ipstats:vsat:out:bytes=2708780293;ipstats:vsat:in:packets=8053049;ipstats:vsat:in:bytes=5907406768;       \
   ipstats:3g:out:packets=2376199;ipstats:3g:out:bytes=483308782;ipstats:3g:in:packets=2011124;ipstats:3g:in:bytes=884642926
119;vsat:SNR=11.3 121;tgpp:SIM:usage=54353259 130;vsat:SNR=11.2

...

599;ipstats:vsat:out:packets=7984363;ipstats:vsat:out:bytes=2708860395;ipstats:vsat:in:packets=8053386;ipstats:vsat:in:bytes=5907484405;       \
   ipstats:3g:out:packets=2376758;ipstats:3g:out:bytes=483348916;ipstats:3g:in:packets=2011550;ipstats:3g:in:bytes=884668740;                \
   ipstats:dnsint:out:packets=87884;ipstats:dnsint:out:bytes=7006626;ipstats:dnsint:in:packets=79215;ipstats:dnsint:in:bytes=11845638;         \
   ipstats:dnspub:out:packets=25404;ipstats:dnspub:out:bytes=2066421;ipstats:dnspub:in:packets=23261;ipstats:dnspub:in:bytes=3593385;         \
   ipstats:dnspub:out:2:packets=67514;ipstats:dnspub:out:2:bytes=5497193;ipstats:dnspub:in:2:packets=67205;ipstats:dnspub:in:2:bytes=10267975
602;tgpp:SIM:usage=54418935
616;vsat:SNR=11.3
627;vsat:SNR=11.0
639;vsat:SNR=11.2

...

83108;tgpp:SIM:usage=149449101
83114;PEP:status=Ok
83140;vsat:SNR=10.1
83159;ipstats:vsat:out:packets=8231993;ipstats:vsat:out:bytes=2755236511;ipstats:vsat:in:packets=8348552;ipstats:vsat:in:bytes=6146494388;     \
   ipstats:3g:out:packets=2550904;ipstats:3g:out:bytes=499147697;ipstats:3g:in:packets=2165187;ipstats:3g:in:bytes=963907263
83164;vsat:SNR=10.2
83169;tgpp:SIM:usage=149457351
83175;vsat:SNR=9.9
83186;vsat:SNR=10.2
83208;vsat:SNR=10.0
83210;tgpp:quality=high;tgpp:quality:reason=HSPA or HSPA+ connection;tgpp:syssubmode=HSPA+

...

86346;tgpp:SIM:usage=149911406
86348;vsat:SNR=9.9
86359;vsat:SNR=9.6
86365;gps:dir=352.00;gps:dist:n=974159;gps:dist:sd=4.616789;gps:tms=1356048134
86382;vsat:SNR=9.7
```

[32] https://www.anywi.com/contact/

D ltrans-relations.py

Proof Of Concept for automatic identification of the time-stamp entries in `LTRANS` files, the input for the command requires a CSV converted `LTRANS` file.

ltrans-relations.py

```
#!/usr/bin/env python
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#
'''
  Find relations by brute-forcing through the code
'''

import argparse
import sys
import csv
import logging
from collections import defaultdict

logger = logging.getLogger()
logger.setLevel(logging.INFO)

fmt = logging.Formatter('# %(levelname)s: %(message)s')

ch = logging.StreamHandler()
ch.setFormatter(fmt)
ch.setLevel(logging.DEBUG)

logger.addHandler(ch)

def set_prefix(prefix):
  fmt._fmt = ('# [%30s] ' % prefix) + '%(levelname)s: %(message)s'


parser = argparse.ArgumentParser(description='Check some data')
parser.add_argument('--constant', action='store_true', default=False)
parser.add_argument('--unique-mapping', action='store_true', default=False)
parser.add_argument('--debug','-g', action='store_true', default=False)
parser.add_argument('--infile', default='./bert.csv')
parser.add_argument('fields', nargs='*')
args = parser.parse_args()

logger.debug(args)

if args.debug:
  logger.setLevel(logging.DEBUG)


def check_constant_row(headers, rows, i):
  ''' See if there is only one specific value in the row '''
  column = []
  for row in rows:
    column.append(row[i])

  uniq_count = len(set(column))
  logger.debug("Unique entries for row %s: %i", headers[i], uniq_count)

  return (uniq_count == 1)



def check_uniq_mapping(headers, rows, i, j):
  ''' All values together are unique sets '''
  sets = []
  for row in rows:
    sets.append((row[i], row[j]))

  uniq_count = len(set(sets))
  total_count = len(rows)
  logger.debug("Unique sets: %i (total: %i)", uniq_count, total_count)

  return (uniq_count == total_count)


def check_linear_relation(headers, rows, i, j):
  diff = None
  for row in rows:
    if row[i] != None and row[j] != None:
      new_diff = abs(row[i] - row[j])
      if diff != None and new_diff != diff:
        logger.debug("Old diff: %f, New diff: %f", diff, new_diff)
        return False
  return (diff != None)



if __name__ == '__main__':
  # Store unique mappings for time indentification purposes
  uniq_mapping = defaultdict(list)

  # Open file and read headers
  logging.info("Reading file %s", args.infile)
  csv_reader = csv.reader(open(args.infile, 'r'))
  headers = csv_reader.next()

```

```
92    logging.info("Going to process headers: %s", headers)
93
94    # Find out which fields to check
95    if not args.fields:
96      field_ids = range(len(headers))
97    else:
98      field_ids = []
99      for nr, field in enumerate(headers):
100       # Include fields which have a partial matchting
101       if filter(lambda x: x in field, args.fields):
102         field_ids.append(nr)
103     logger.debug("Trying to relate the folllowing fields: %s", [headers[x] for x in field_ids])
104
105
106   # TODO(rvdz): Not all functions could be able to read all entries, when growing
107   # to large it could lead to memory excaustion.
108   logging.debug("Converting all data to float entries")
109   rows = []
110   for row in csv_reader:
111     rows.append(map(lambda x: None if x == '' else float(x), row))
112
113   # Compare headers with each-other (single relation)
114   while field_ids:
115     i = field_ids.pop(0)
116     if args.constant:
117       is_constant = check_constant_row(headers, rows, i)
118       if is_constant:
119         logger.info("%s is constant: %s", headers[i], is_constant)
120     for j in field_ids:
121       set_prefix('%s vs %s' % (headers[i], headers[j]))
122       if args.unique_mapping:
123         is_uniq_mapping = check_uniq_mapping(headers, rows, i, j)
124         if is_uniq_mapping:
125           logger.info("%s has unique mapping with %s", headers[i], headers[j])
126           # Unique mapping are two-way relations
127           uniq_mapping[i].append(j)
128           uniq_mapping[j].append(i)
129       is_linear_relation = check_linear_relation(headers, rows, i, j)
130       if is_linear_relation:
131         logger.info("%s has linear relation with %s", headers[i], headers[j])
132
133
134   # We now check which key has unique mappings to all other keys, this are
135   # our primary indentifiers.
136   for i in range(len(headers)):
137     if len(uniq_mapping[i]) == len(headers) - 1:
138       logger.info("%s is an identifier", headers[i])
```

E ltrans2csv.py

Helper to convert LTRANS files to CSV formatted files.

ltrans2csv.py

```
#!/usr/bin/env python
#
# LTRANS conversion to CSV
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#

__version__ = '$Id: ltrans2csv.py 13788 2013-07-17 12:30:20Z rick $'

import argparse
import csv
import logging
import sys

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger()

# In order to store the data in CSV format we need to find out how many columns
# we need to store, secondly we temponary store the data somehow.
#
# Going to trick a bit, by storing the complete entries in memory, when
# optimalisation is needed we need two-pass logic or alternative methods.
def parse_file(filename, args):
  """
  Process file and watch special keys for changes
  """
  rows = []
  keys = set([])
  with open(filename,'r') as fh:
    logger.info("Processing %s", filename)
    # Process the the header
    header = fh.readline()
    if not header.startswith('ltrans='):
      raise IOError("Invalid format")
    # TODO(rvdz): Meta keys are to be included in all entries
    meta = dict(map(lambda x: x.split('=', 2), header.rstrip().split(';')))

    # Get the field listing, since it part of the protocol
    fieldlist = fh.readline().strip().split(';')

    # Get actual values
    for line in fh.xreadlines():
      # The first value is timestamp offset, this is how-ever implicitly mentioned
      line = 'offset=' + line
      v = dict(map(lambda x: x.split('=', 2), line.strip().split(';')))
      v.update(meta)
      rows.append(v)
      # Update key mapping to include relevant new keys
      keys |= set(v.keys())

  # Assign every key it's corresponding row, for debugging purposes make listing sorted
  key2row = dict(zip(sorted(keys), range(0, len(keys))))

  with open(args.output, 'wb') as csvfile:
    writer = csv.writer(csvfile)
    # First create empty list and secondly fill the list with entries on the
    # right spot and append it to the CSV output
    #
    # First entry is trick to print header
    for row in [dict(zip(key2row, key2row))] + rows:
      t = [''] * len(key2row)
      for k,v in row.iteritems():
        t[key2row[k]] = v
      writer.writerow(t)



if __name__ == '__main__':
  parser = argparse.ArgumentParser(description='LTRANS converter to CSV')
  parser.add_argument('files', metavar='file', type=str, nargs='+', help="Filename to parse")
  parser.add_argument('--quiet', '-q', action='store_true', default=False)
  parser.add_argument('--debug', '-g', action='store_true', default=False)
  parser.add_argument('--output', '-o', default='/dev/stdout')
  args = parser.parse_args()

  if args.quiet:
    logger.setLevel(logging.ERROR)
  elif args.debug:
    logger.setLevel(logging.DEBUG)
  else:
    logger.setLevel(logging.INFO)
  for filename in args.files:
    parse_file(filename, args)
```

F pygame_heatmap_plot.py

Library to generate heatmap plots based on Python pygame Gaming Engine.

python_heatmap_plot.py

```
#!/usr/bin/python
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#
"""
Using pygame to display heatmaps with interactive posibilities
"""

__author__ = "$Author: rick $"
__version__= "$Revision: 14083 $"
__date__   = "$Date: 2013-09-23 15:24:16 +0200 (Mon, 23 Sep 2013) $"


import gradients
import math
import matplotlib
matplotlib.use("Agg")
import matplotlib.backends.backend_agg as agg
import numpy as np
import pygame
import pylab
import random
import time

from mpl_toolkits.basemap import Basemap, cm

# Where can we find this button variables?
pygame.mouse.LEFT, pygame.mouse.MIDDLE, pygame.mouse.RIGHT, pygame.mouse.UP, pygame.mouse.DOWN = range(1,6)

# http://stackoverflow.com/questions/15736995/how-can-i-quickly-estimate-the-distance-between-two-latitude-longitude-points#answer-15737218
from math import radians, cos, sin, asin, sqrt
def haversine(lon1, lat1, lon2, lat2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)
    """
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    km = 6367 * c
    return km


def make_map(min_lon, max_lon, min_lat, max_lat, xdim, ydim, resolution='low', batch=False):
  print "New zoomlevel: ", min_lon, max_lon, min_lat, max_lat
  fig = pylab.figure(
    figsize=[xdim/100, ydim/100], # Inches
    dpi=100,          # 100 dots per inch, so the resulting buffer is 400x400 pixels
    frameon=False,
  )

  # Convertion of fullnames to shortnames
  res_map = dict(crude='c', low='l', intermediate='i', high='h', full='f')

  fig.subplots_adjust(left=0.0, right=1, top=1, bottom=0.0)
  ax = fig.gca()
  m = Basemap(ax=fig.gca(), projection='merc',
                resolution = res_map[resolution], area_thresh = .1,
                llcrnrlat = min_lat, llcrnrlon = min_lon,
                urcrnrlat = max_lat, urcrnrlon = max_lon,
                fix_aspect=False,
               )


  # draw coastlines, country boundaries, fill continents.
  ##m.bluemarble()
  m.drawcoastlines(linewidth=.4)
  m.drawstates()
  m.drawcountries(linewidth=1)
  #m.fillcontinents(color='white',lake_color='blue')
  m.drawrivers(color='blue')
  # draw the edge of the m projection region (the projection limb)
  m.drawmapboundary()
  # draw lat/lon grid lines every 30 degrees.
  m.drawmeridians(np.arange(0, 360, 30))
  m.drawparallels(np.arange(-90, 90, 30))

  canvas = agg.FigureCanvasAgg(fig)
  canvas.draw()
  renderer = canvas.get_renderer()
  raw_data = renderer.tostring_rgb()
  size = canvas.get_width_height()
  surf = pygame.image.fromstring(raw_data, size, "RGB")
  return surf

def make_app(points, boundry=(3.0, 11.25, 50.0, 55.0), min_signal=5, max_signal=15, resolution='low', batch=False):
      # Screen start location and sizing
```

```
        min_lon, max_lon, min_lat, max_lat = boundry

        ratio = (max_lat - min_lat) / (max_lon - min_lon)

        xdim = 1000
        ydim = 1000

        pygame.init()
        screen = pygame.display.set_mode((xdim,ydim), pygame.SRCALPHA)
        pygame.key.set_repeat(100, 30)

        # Initial controls
        map_changed = True
        layer_changed = True
        zoom_active = False
        zoom_changed = False
        aspect_free_zooming = True

        last_zoom_change = 0
        prev_coords = [(-10.0,30.0,30.0,60.0)]

        font=pygame.font.Font(None,30)
        curr_lon = 0
        curr_lat = 0

        while True:
            try:
                event = pygame.event.wait()
            except KeyboardInterrupt:
                break
            if event.type == pygame.QUIT:
                break
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_q:
                    break
                elif event.key == pygame.K_SPACE:
                    pass
                elif event.key in (pygame.K_RSHIFT, pygame.K_LSHIFT):
                    aspect_free_zooming = False
            elif event.type == pygame.KEYUP:
                if event.key in (pygame.K_RSHIFT, pygame.K_LSHIFT):
                    aspect_free_zooming = True
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == pygame.mouse.LEFT:
                  zoom_active = True
      zoom_start = event.pos
                  zoom_end = event.pos
            elif event.type == pygame.MOUSEBUTTONUP:
                if event.button == pygame.mouse.LEFT:
                  # Determine new array to display
                  if zoom_active:
                    zoom_end = event.pos
                    prev_coords.append((min_lon, max_lon, min_lat, max_lat))
        # Get left buttom the lower bound and right top as upper
        # bound, note that coordinates start left-top, where-as
        # lonlat starts at left-bottom
                    lb = (min(zoom_start[0], zoom_end[0]), max(zoom_start[1], zoom_end[1]))
                    rt = (max(zoom_start[0], zoom_end[0]), min(zoom_start[1], zoom_end[1]))

                    min_lon = min_lon + (max_lon - min_lon) * float(lb[0])/xdim
                    max_lon = min_lon + (max_lon - min_lon) * float(rt[0])/xdim
                    min_lat = min_lat + (max_lat - min_lat) * float(xdim - lb[1])/xdim
                    max_lat = min_lat + (max_lat - min_lat) * float(xdim - rt[1])/xdim

                    zoom_active = False
                    map_changed = True
                elif event.button == pygame.mouse.RIGHT:
                   if prev_coords:
                     min_lon, max_lon, min_lat, max_lat = prev_coords.pop()
                     map_changed = True
            elif event.type == pygame.MOUSEMOTION:
                curr_lon = min_lon + (max_lon - min_lon) * float(event.pos[0])/xdim
                curr_lat = min_lat + (max_lat - min_lat) * float(xdim - event.pos[1])/xdim
    if zoom_active:
                  if aspect_free_zooming:
                    zoom_end = event.pos
                  else:
                    # Square zoom to preserve image ratio
                    dx = max(event.pos[0], zoom_start[0]) - min(event.pos[0], zoom_start[0])
                    dy = max(event.pos[1], zoom_start[1]) - min(event.pos[1], zoom_start[1])

                    dyx = max(dx, dy)
                    new_dx = zoom_start[0] + (dyx if event.pos[0] > zoom_start[0] else dyx * -1) * ratio
                    new_dy = zoom_start[1] + (dyx if event.pos[1] > zoom_start[1] else dyx * -1)

                    zoom_end = (new_dx, new_dy)


            if map_changed:
    t1 = time.time()
                map_layer = make_map(min_lon, max_lon, min_lat, max_lat, xdim, ydim, resolution)
                t2 = time.time()

                # Check the amount of km per pixel
                dx_km = haversine(min_lon, min_lat, max_lon, min_lat) / xdim
                dy_km = haversine(min_lon, min_lat, min_lon, max_lat) / ydim
                d_km = max(dx_km, dy_km)
                print "KM per pixel: %s" % d_km

                max_radius_in_km = 1.0
                # Plot all points on right location
                for point in points.items():
                  (lon,lat),signal = point
```

```
                    # The higher the signal the bigger the radius
                    #radius = (max_radius_in_km / d_km) * ((signal - min_signal) / (max_signal - min_signal))
                    # Fixed radius
                    radius = (max_radius_in_km / d_km)
                    x = (lon - min_lon) / ((max_lon - min_lon) / xdim)
                    y = ydim - ((lat - min_lat) / ((max_lat - min_lat) / ydim))

        # Hack for route display
        radius = 2

                    # Plotting less then 1 pixel is not possible
                    if radius < 1:
                      continue


                    # Green till Red depending on the signal
                    quality = (signal - min_signal) / (max_signal - min_signal)
                    c_fixed = (int(255.0 * (1.0 - quality)), int(255.0 * quality), 0)
                    c_start = c_fixed + (150,)
                    c_end = c_fixed + (0,)

                    # Hack for route display - Fixed colors and alpha (no gradient)
                    c_start = (0,255,0,150)
                    c_end = (0,255,0,0)

        # Do plot entries which do fall on the map, mind that we
        # like to get the 'shadows' of points just outside the map
                    if x > (0 - radius) and x < (xdim + radius) and y > (0 - radius) and y < (ydim + radius):
                      map_layer.blit(gradients.radial(int(radius), c_start, c_end), (x, y))
          t3 = time.time()

              # Display coordinates and update acordenly
              label = font.render("lon:%7.5f lat:%7.5f" % (curr_lon, curr_lat), 1, (255,0,0), (255,255,255))
              pygame.display.update(screen.blit(label, (0,0)))

              if map_changed or layer_changed or zoom_active:
                  if zoom_active:
                    #Avoid updating more than 100 miliseconds at the time, as it makes it slow
                    if time.time() - last_zoom_change < 0.1:
                      continue
                    else:
                      last_zoom_change = time.time()

                  map_changed, layer_changed = False,False
                  # Background
                  screen.fill((255,255,255,0))

                  # Actual map
                  screen.blit(map_layer, (0,0))


                  # Zoom layer (if applicable)
                  if zoom_active:
                    zoom_diff = (zoom_end[0] - zoom_start[0], zoom_end[1] - zoom_start[1])
                    pygame.draw.rect(screen, (255,0,0), zoom_start + zoom_diff, 2)

                # Update screen
              print "Map generation time: ", t2 - t1
          print "Plotting all points: ", t3 - t2
      print "Resolution used    : ", resolution
                  print "--------------------"
                  pygame.display.flip()
      pygame.image.save(screen, 'screenshot-%s.png' % (resolution))
      if batch:
        time.sleep(1)
              break

          print "Shutting down, please wait ..."
          pygame.quit()


if __name__ == '__main__':
      points = {
        (52.4, 3.15) : 1,
      }

      make_app(points)
```

G ltrans-parser.py

ltrans-parser.py

```
#!/usr/bin/env python
#
# LTRANS v1.0 parser, state changes sent over effiently
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#

import argparse
import gzip
import itertools
import logging
import re
import sys
import yaml

from decimal import Decimal, InvalidOperation
from collections import defaultdict,Counter
from datetime import datetime

import pygame_heatmap_plot

#import matplotlib
#matplotlib.use('GTKCairo')
#print matplotlib.get_backend()

import heatmap
import realmap

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import matplotlib.dates as mdates
import numpy as np


# NOTSET = 0
# DEBUG = 10
# INFO = 20
# WARN = 30
# WARNING = 30
# ERROR = 40
# FATAL = 50
# CRITICAL = 50

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger()

meta = {'timestamp' : 0}
state = {}

# 'Recursive' metadata to store
# - type    : [int, float, string, list]
# - max     : Maximum value
# - min     : Minimum value
# - changes: Amount of point changed
stats = {
    'changes' : defaultdict(int),
    'type' : defaultdict(itertools.repeat(type(None)).next),
    'min' : defaultdict(itertools.repeat(sys.maxint).next),
    'max' : defaultdict(itertools.repeat(sys.maxint * -1).next),
    'choices': defaultdict(set),
    'avg_delta' : defaultdict(int),
    'avg_start' : defaultdict(int),
    'count_min' : defaultdict(itertools.repeat(sys.maxint).next),
    'count_max' : defaultdict(itertools.repeat(sys.maxint * -1).next),
    'same' : defaultdict(int),
    'errors' : defaultdict(int),
}

float_re = re.compile('^\d+\.\d+$')
foobar = defaultdict(list)
lonlat = {}

def make_list(raw_value, delim=','):
   raw_list = raw_value.split(delim)
   if all(map(lambda x: x.isdigit(), raw_list)):
     return map(int,raw_list)
   elif float_re.match(raw_list[0]):
     return map(float,raw_list)
   else:
     return raw_list

def parse_field(field, raw_value):
  """
  Process field returning it's processed value
  """
  # timestamp file is 'special' as it is a offset of the regular timestamp
  # base, as field could both be in field array or field k=v pairs, make
  # the parsing generic
  # XXX: We need metadata for convertion of fields
  if field == 'timestamp':
    return meta['timestamp'] + int(raw_value)
  elif field == 'wifi:scan':
    return map(lambda x: x.split('/'), raw_value.split(','))
  elif field == 'gps:sats:track':
    return map(lambda x: x.split(':'), raw_value.split(','))
  elif raw_value.isdigit():
    return int(raw_value)
```

```
  elif float_re.match(raw_value):
    return float(raw_value)
  elif ',' in raw_value:
    return make_list(raw_value, ',')
  elif ':' in raw_value:
    return make_list(raw_value, ':')
  else:
    return raw_value


def has_prefix(field, prefix_array,empty_ok=False):
   """ Return True if the prefix array is found in field """
   if not prefix_array and empty_ok:
     return True
   else:
     return any(map(lambda x: field.startswith(x), prefix_array))

def zopen(filename):
    ''' Open file regardless of its compression '''
    if filename.endswith('gz'):
      return gzip.open(filename, 'rb')
    else:
      return open(filename, 'r')

def parse_file(filename, args):
  """
  Process file and watch special keys for changes
  """
  with open('test.dat','w') as data_fh:
    with zopen(filename) as fh:
      logger.info("Processing %s", filename)
      # Process the the header
      header = fh.readline()
      if not header.startswith('ltrans='):
         raise IOError("Invalid format")
      for item in header.rstrip().split(';'):
        k,v = item.split('=',2)
        meta[k] = parse_field(k,v)

      # Get the field listing
      fieldlist = fh.readline().strip().split(';')
      for field in fieldlist:
        state[field] = None


      def change_field(field, raw_value):
        oldvalue = state[field] if state.has_key(field) else None
        value = parse_field(field, raw_value)

  # According to the protocol this should only happen if the value did
  # change multiple times during 5.0 -> 6.0 -> 5.0 for example
        if oldvalue == value:
          # logger.debug("Field %s has same old and new value: %s", field, value)
          stats['same'][field] += 1
          return

  CHANGE_PER_TICK = 0.3
        if field == 'gps:dir' and type(value) in (int, float) and oldvalue:
          # Maak mogelijke delta
          #max_delta = float(state['timestamp'] - stats['avg_start'][field]) * stats['avg_delta'][field]
          max_delta = float(state['timestamp'] - stats['avg_start'][field]) * CHANGE_PER_TICK
          if max_delta > 0 and abs(value - oldvalue) > max_delta:
            stats['errors'][field] +=1
            return
          else:
            stats['avg_start'][field] = state['timestamp']
            #stats['avg_delta'][field] = abs((value - oldvalue) * CHANGE_PER_TICK)

        # Add new value to the database
  stats['changes'][field] += 1
        state[field] = value

        if has_prefix(field, args.watchers):
          logger.debug("Key %s, old: %s -> new: %s", field, oldvalue, value)
          #data_fh.write('%s ' % tate['timestamp'])
          #for watch in args.watchers:
          #  foobar[watch].append((state['timestamp'], value))
            #data_fh.write(' %s' % (state[field] if (watch == field) else '?'))
          #data_fh.write('\n')
          #data_fh.write('%s %s\n' % (state['timestamp'], state[field]))
          #if state.has_key('gps:lat') and state.has_key('gps:lon'):
          #  data_fh.write('%s %s %s %s\n' % (state['gps:lat'], state['gps:lon'], state[field]))

        # Ignore reset for now
        if value == None:
          return

        if type(value) == list and type(value[0]) != list:
          stats['choices'][field] = set(stats['choices'][field]) | set(value)
          stats['count_min'][field] = min(stats['count_min'][field], len(value))
          stats['count_max'][field] = max(stats['count_max'][field], len(value))
        elif type(value) in (int, float):
          if not has_prefix(field, args.exclude):
            foobar[field].append((state['timestamp'], value))
          if has_prefix(field, args.watchers) and not field in ('gps:lon', 'gps:lat'):
            if state['gps:lon'] and state['gps:lat']:
              lonlat[(state['gps:lon'], state['gps:lat'])] = value

          # Set the maximum value and keep counters
          if value > stats['max'][field]:
            stats['count_max'][field] = 1
            stats['max'][field] = value
          elif value == stats['max'][field]:
```

```
201             stats['count_max'][field] += 1
202
203           # Set the minimum value and keep counters
204           if value < stats['min'][field]:
205             stats['count_min'][field] = 1
206             stats['min'][field] = value
207           elif value == stats['min'][field]:
208             stats['count_min'][field] += 1
209
210         elif type(value) == str:
211           stats['choices'][field] = set(stats['choices'][field]) | set([value])
212
213         # Find out if this is the first time we have spotted this key (DayOfBirth?)
214         if not stats['type'].has_key(field):
215           #logger.debug("Field %s is of type %s", field, type(value).__name__)
216           stats['type'][field] = type(value)
217
218       # Start processing the lines
219       for line in fh:
220         items = line.rstrip().split(';')
221         # First get all the field values
222         for field in fieldlist:
223           value = items.pop(0)
224           if value:
225             change_field(field, value)
226
227         # Next the key=value pairs
228         for item in items:
229           field, value = item.split('=',2)
230           change_field(field, value)
231
232       # Boolean flags hiding in a integer output
233       for k in state.keys():
234         if stats['min'][k] != stats['max'][k] and \
235           (stats['count_min'][k] + stats['count_max'][k]) == stats['changes'][k]:
236           stats['type'][k] = bool
237           stats['choices'][k] = [0,1]
238
239       # Float/Int values with limited set of choices and lot of changes are
240       # technicallly speaking lists and/or are 'flapping' between two discrete
241       # levels.
242       # XXX: This should be suggestions and not definitive answers
243       forced_ranges = ('gps:lat', 'gps:lon')
244       for k in set(state.keys()) - set(forced_ranges):
245         if stats['type'][k] in (int, float) and stats['changes'][k] > 100 and foobar.has_key(k):
246           x,y= zip(*foobar[k])
247           # Hack to make sure not run trough all if list is very big
248           if len(Counter(y[:100]).keys()) > 20:
249             continue
250           # Make sure we have less than XX choices
251           cnt = Counter(y)
252           if len(cnt.keys()) <= 20:
253             stats['type'][k] = list
254             stats['choices'][k] = cnt.keys()
255
256       for k in state.keys():
257         if stats['type'][k] in (int, float) and stats['changes'][k] > 100 and foobar.has_key(k):
258           x,y= zip(*foobar[k])
259           # 1st derivitive
260           y1 = [0.0] + [y[i] - y[i-1] for i in range(1,len(y))]
261           #tmp = sorted(y1)[int(len(y1)*0.1):-int(len(y1)*0.1)]
262           y1_avg = sum(y1) / len(y1)
263
264           # 2nd derivitive
265           y2 = [0.0] + [y1[i] - y1[i-1] for i in range(1,len(y1))]
266           #tmp = sorted(y2)[int(len(y2)*0.1):-int(len(y2)*0.1)]
267           y2_avg = sum(y2) / len(y2)
268           #logger.warn("1st derivative - min:%s avg:%s min:%s", min(y1), y1_avg, max(y1))
269           #logger.warn("2nd derivative - min:%s avg:%s min:%s", min(y2), y2_avg, max(y2))
270
271       # Cleanup of string values
272       for k in state.keys():
273         if stats['type'][k] in (str,list):
274           stats['min'][k] = sys.maxint
275           stats['max'][k] = sys.maxint * -1
276           stats['count_min'][k] = 0
277           stats['count_max'][k] = 0
278
279
280       # Print all states found
281       with open(filename + '.meta','w') as meta_fh:
282         meta_fh.write("ltrans_meta=1.0\n")
283         key_list = state.keys() if not args.result else filter(lambda x: has_prefix(x, args.result), state.
              keys())
284         for k in sorted(key_list):
285           meta_dict = {}
286           for t in sorted(stats.keys()):
287             if not stats[t][k]:
288               v = 'NaN'
289             elif t == 'type':
290               v = stats[t][k].__name__
291             elif t == 'choices':
292               v = ','.join(map(str,stats['choices'][k]))
293             else:
294               v = stats[t][k]
295             meta_dict[t] = v
296             logger.log(logging.INFO if has_prefix(k, args.watchers) else logging.DEBUG, "key=%-30s field=%-10
                  s value=%s",k, t, v)
297           logger.debug("===")
298
299           meta_line = ';'.join(['key=%s' % k] + ["%s=%s" % x for x in sorted(meta_dict.items())])
300           meta_fh.write(meta_line + "\n")
301
```

```
302
303 def draw_image():
304   if args.grid:
305     heatmap.draw(lonlat)
306   elif args.realmap:
307     realmap.draw_heatmap(lonlat)
308   elif args.heatmap:
309     boundry = (
310       stats['min']['gps:lon'] - 1, stats['max']['gps:lon'] + 1,
311       stats['min']['gps:lat'] - 1, stats['max']['gps:lat'] + 1
312     )
313
314     pygame_heatmap_plot.make_app(lonlat, boundry=boundry, min_signal=stats['min'][args.watchers[0]],
          max_signal=stats['max'][args.watchers[0]], resolution=args.resolution, batch=args.batch)
315   elif args.plot or args.hist:
316     print "Plotting a image of keys %s" % args.watchers
317     plt.close('all')
318     fig,ax = plt.subplots(1)
319     for key in sorted(state.keys()):
320       if not stats['type'][key] in (int, float):
321         continue
322       if has_prefix(key,args.exclude):
323         continue
324       if args.watchers and not has_prefix(key,args.watchers):
325         continue
326       if not foobar.has_key(key):
327         continue
328       if stats['errors'][key] > args.max_errors:
329         logger.error("Not plotting key %s due to many errors %s (allowed:%s)", key, stats['errors'][key],
              args.max_errors)
330       x,y= zip(*foobar[key])
331       y = np.array(y)
332       if not y.max():
333         continue
334       if args.norm:
335         y = y / float(y.max())
336       ## 1st derivitive
337       # y1 = [0.0] + [y[i] - y[i-1] for i in range(1,len(y))]
338       ##2nd derivitive
339       # y2 = [0.0] + [y1[i] - y1[i-1] for i in range(1,len(y1))]
340       if args.hist:
341         ax.hist(y,label=key)
342         logger.info(Counter(y))
343       else:
344         x = map(lambda u: datetime.utcfromtimestamp(u), x)
345         ax.step(x,y,'.-',label=key)
346     if not args.hist:
347       fig.autofmt_xdate()
348     ax.grid(True)
349
350     formatter = ticker.EngFormatter(places=5)
351     ax.yaxis.set_major_formatter(formatter)
352
353     # ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d %H:%M:%S'))
354     # Shink current axis's height by 10% on the bottom
355     box = ax.get_position()
356     ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])
357     # Put a legend below current axis
358     ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
359
360     #plt.ylabel('||'.join(sorted(foobar.keys())))
361     plt.show()
362
363
364 if __name__ == '__main__':
365   parser = argparse.ArgumentParser(description='Process some integers.')
366   parser.add_argument('files', metavar='file', type=str, nargs='+', help="Filename to parse")
367   parser.add_argument('--watchers', '-w', type=str, help='Watch changes in watcher keys', action='append',
        default=[])
368   parser.add_argument('--exclude', '-x', type=str, help='Keys to ignore in display and debug', action='append
        ', default=[])
369   parser.add_argument('--result', '-r', type=str, help='Only display summary for keys', action='append',
        default=[])
370   parser.add_argument('--quiet', '-q', action='store_true', default=False)
371   parser.add_argument('--debug', '-g', action='store_true', default=False)
372   parser.add_argument('--hist', help='Show histogram', action='store_true', default=False)
373   parser.add_argument('--norm', help='Normalize Graphs', action='store_true', default=False)
374   parser.add_argument('--plot', help='Show graph plot', action='store_true', default=False)
375   parser.add_argument('--grid', help='Show lon/lat graph plot', action='store_true', default=False)
376   parser.add_argument('--realmap', help='Show lon/lat graph plot at real map', action='store_true', default=
        False)
377   parser.add_argument('--heatmap', help='Show lon/lat graph plot at heat map', action='store_true', default=
        False)
378   parser.add_argument('--max-errors', help='Maximum of errors allowd', type=int, default=100)
379   parser.add_argument('--single', help="Plot every picture individualy", action='store_true', default=False)
380   parser.add_argument('--batch', help="No interactive display possible", action='store_true', default=False)
381   parser.add_argument('--resolution', help="Resolution of basemap", default='low')
382   args = parser.parse_args()
383
384   if args.quiet:
385     logger.setLevel(logging.ERROR)
386   elif args.debug:
387     logger.setLevel(logging.DEBUG)
388   else:
389     logger.setLevel(logging.INFO)
390   for filename in args.files:
391     parse_file(filename, args)
392     if args.single:
393       draw_image()
394
395   if not args.single:
396     draw_image()
```

H ltrans-filter.py

Example code of filtering values which cannot be used by for numeric statistics and analytics. The input for this command requires a CSV converted `LTRANS` file.

ltrans-filter.py

```
#!/usr/bin/env python
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#
'''
  Filter out values which are non-interesting
'''

import sys
import csv
import logging
from collections import defaultdict

logging.basicConfig(level=logging.DEBUG, stream=sys.stderr)
logger = logging.getLogger()

fields = None
infile = '../ltrans-data/00:0d:b9:1b:57:10/buffered/1301529601.ltrans.csv'

# Filename to parse
if len(sys.argv) > 1:
  infile = sys.argv[1]

# Fields with this names to store
if len(sys.argv) > 2:
  fields = sys.argv[2:]

class mixed(str): pass

# Open file and read headers
logging.info("Reading file %s", infile)
csv_reader = csv.reader(open(infile, 'r'))
headers = csv_reader.next()


def classify(value):
    # Do not process empty values
    if value == None or value == '':
        return None
    elif value.isdigit():
        return int
    elif all(map(lambda x: x.isdigit(), value.split('.', 1))):
        return float
    else:
        return str

# Walk through lines, for every line check type of value
header_type = defaultdict(lambda: None)
for row in csv_reader:
    pass
    value_types = map(classify, row)
    for (h,t) in zip(headers, value_types):
        if t == None: continue
        if header_type[h] == None:
            header_type[h] = t
        elif header_type[h] != t:
            # Combine new type with existing type (if found)
            header_type[h] = mixed
            print "Type-re-identified: %s %s", header_type[h], t


# Filter out columns which are not interesting
store_headers = []
for h,t in  header_type.iteritems():
  if t in (int, float):
    store_headers.append(h)


logger.info("Storing trimmed version with keys %s", sorted(store_headers))
# Write result to file
csv_writer = csv.writer(sys.stdout)

# Avoiding memory hop, re-reading entries
csv_reader = csv.reader(open(infile, 'r'))

# Filter extra fields if needed
if fields:
  store_headers = filter(lambda x: any(map(lambda y: y in x, fields)), store_headers)

# Masking of headers to keep
b = [x in store_headers for x in headers]
for row in csv_reader:
    # Filter out relevant rows
    row = map(lambda x: x[1], filter(lambda x: x[0], zip(b, row)))
    # Only output the rows which has output entries
    if any(row):
      csv_writer.writerow(row)
```

I ltrans-expand.py

Example code of expanding the `LTRANS` CSV file, such that all fields gets filled in again.

ltrans-expand.py

```
#!/usr/bin/env python
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#
'''
  Expand cell entries which are missing
'''

import sys
import csv
import logging
from collections import defaultdict

logging.basicConfig(level=logging.DEBUG, stream=sys.stderr)
logger = logging.getLogger()

infile = sys.stdin

# Filename to parse
if len(sys.argv) > 1:
  infile = open(sys.argv[1], 'r')

# Open file and read headers
logging.info("Reading file %s", infile)
csv_reader = csv.reader(infile)
headers = csv_reader.next()
header_len = len(headers)

logger.info("Storing expanded version with keys %s", sorted(headers))
# Write result to file
csv_writer = csv.writer(sys.stdout)

# Headers please
csv_writer.writerow(headers)

# Fill cells which are missing
prev_row = csv_reader.next()
# Make sure to write the row if possible
if len(filter(None, prev_row)) == header_len:
  csv_writer.writerow(prev_row)

# Process all remaining entries
for row in csv_reader:
  # Use old value in case of blank field
  new_row = map(lambda x: x[0] if x[0] else x[1], zip(row, prev_row))

  # We need to store the last values for next interation
  prev_row = new_row

  # As soon we have all values filled, print the line
  # NOTE: this could potenially leave all kind of values missing, since it
  # require _ALL_ values to appear ones, before output takes place
  if len(filter(None, new_row)) == header_len:
    csv_writer.writerow(new_row)
```

J random-dots.gnu

random-dots.gnu

```
#!/usr/bin/gnuplot
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#
set terminal pngcairo notransparent enhanced font "arial,10" fontscale 1.0 size 1024,768
set output 'random-dots.png'

unset key; unset border; unset tics
set size ratio -1

set samples 1000
plot 'world.dat' with lines lt 3, \
     (rand(0) * 160) - 80 with points pt 7 ps 1
```

K radient-dots.gnu

radient-dots.gnu

```
#!/usr/bin/gnuplot
#
# Draw points with radiant
#
# Inspiration: http://www.gnuplotting.org/electron-and-positron/
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#

reset

# png
set terminal pngcairo size 1024,768 enhanced font 'Verdana,10'
set output 'radient-dots.png'
set yzeroaxis linetype 0 linewidth 1.000

unset key; unset border; unset tics
set size ratio -1
max = 50
s = 5

# Functions
size(x,n) = s*(1-0.8*x/n)
r(x,n) = floor(19.0*x/n)+221
g(x,n) = floor(216.0*x/n)+24
b(x,n) = floor(209.0*x/n)+31
posx(X,x,n) = X + 0.03*x/n
posy(Y,x,n) = Y + 0.03*x/n
red(x,n) = sprintf("#%02X%02X%02X",r(x,n),g(x,n),b(x,n))

object_number = 1

do for [x=-150:150:10] {
# Draw positron and electron
  do for [y=-60:60:10] {
    # Draw circles
    set for [n=0:max-1] object n+object_number circle \
        at posx(x,n,max/1.0),posy(y,n,max/1.0) size size(n,max/1.0)
    set for [n=0:max-1] object n+object_number \
        fc rgb red(n,max/1.0) fillstyle solid noborder lw 0
    object_number = object_number+max
  }
}
plot 'world.dat' with lines lt 3
```

L curve-fitting.r

curve-fitting.r

```
#!/usr/bin/Rscript
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#
png('curve-fitting-example.png')
# Creating initial dataset
set.seed(510)
len <- 100
min <- -10

x <- runif(len, min=min, max=10)
# Dummy function to follow
y <- x^5 - 3*x^4 + 3*x^3 - 2*x^2 -5 + rnorm(len, 0, 7000)
ds <- data.frame(x = x, y = y)

# Gauss-Newton
m <- nls(y ~ I(a*x^5) + I(b*x^4) + I(c*x^3) + I(d*x^2) + I(e*x) + f, data = ds, start = list(a=1, b=1, c=1, d
    =1, e=1, f=1), trace = TRUE,
  control=list(minFactor=0, maxiter=100, warnOnly = TRUE))
summary(m)

s <- seq(min, 100, length = 1000)
dt <- s ~ predict(m, list(x =s))

# Plotting the variables
plot(y ~ x, main = "Guass-Newton Curve Fitting", sub = "Black: observations; green: fit; red: target", pch
    =20)
grid()

lines(s, s^5 - 3*s^4 + 3*s^3 - 2*s^2 -5, lty = 2, col = "red")
lines(s, predict(m, list(x = s)), lty = 1, col = "green")

dev.off()
```

M GNUmakefile

Various ways of calling the program, including the listing of extra packages required.

GNUmakefile

```
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#
FILENR ?= 1
LTRANSFILE ?= $(shell grep -l -m 1 vsat:SNR `find ltrans-data -type f -name '*.ltrans' -exec ls -1t '{}' \+` | head -$(FILENR) | tail -1)

all:
  ./src/ltrans-parser.py --batch --watcher vsat:SNR --heatmap ${LTRANSFILE}

plot-route:
  ./src/ltrans-parser.py --watcher gps:dist:sd --heatmap `cat track.txt`

find-relation:
  ./src/ltrans-filter.py > bert.csv
  ./src/ltrans-relations.py

install:
  yum install -y python-yaml pygame python-matplotlib python-basemap python-basemap-data-hires java-1.7.0-openjdk
```

N simple-search.sh

A very simple way of parsing the data to get the minimum and maximum values of certain tags, without storing the data, every time these tags are required the full file has to be searched again.

simple-search.sh

```
#!/bin/sh
#
# Very simple identification of maximum and minimum values for specific keys.
#
# Rick van der Zwet <info@rickvanderzwet.nl>
#

# Temponary storage location
TFILE=`mktemp`

# For all requested files on CLI
for FILE in $*; do
  for KEY in gps:lon gps:lat; do
    # Parse ltrans keys and find specific keys
    zcat $FILE | grep '^[0-9]*[0-9];' | tr ';' '\n' | grep "$KEY" | awk -F= '{print $2}' | sort -n > $TFILE
    # Display the result
    printf "%s - %s - %15s - %15s\n" $FILE $KEY `head -1 $TFILE` `tail -1 $TFILE`
  done
done

# Cleanup when done
rm $TFILE
```

O LTRANS Protocol Documentation

This included document will describe the LTRANS protocol documentation. The LTRANS documentation is written by Nick Hibma, AnyWi.com.

AnyWi Technology BV
Plantsoen 97
2311 KL Leiden
The Netherlands

T +31 (0)71 5131396
E info@anywi.com
I www.anywi.com

Technical Documentation
Itrans protocol v1.0

History

Rev.	*Date*	*Changes*
0.0	2008/04/01	Document creation
0.1	2008/05/13	First draft
0.2	2008/06/30	Added documentation for ltransgw communication links
0.3	2008/07/18	Added a remark on the forced escaping done in the ltrans lib now.
0.4	2008/09/24	Document ltransgw data-out connections
0.5	2008/10/29	Document more of the daemons
0.6	2013/07/23	Indicate that '.' (period) can be used as a hierarchy indicator as well

Contents

1. Introduction

This document describes a simple communication protocol for efficient communication of large quantities of sensor data.

This protocol has been developed by AnyWi. This documentation contains proprietary information and is provided within the context of a project for completion of that project. For licensing information please contact AnyWi Technologies, Leiden, the Netherlands.

The ltrans protocol defines one-way communication of key/value pairs from transmitter to receiver. It is intended to support a continuous flow of slowly changing sensors ('column fields') as well as transmission of events (key/value pairs or 'pairs'). The minimum update interval is 1 second.

Example:

```
ltrans=1.0;unitid=00:11:22:33:44:55:66;timestamp=1205306392
timestamp;lat;lon;vel;dir
34;50.75123;4.67843;0.0;186;active=0;descr=Text with spaces;sensor1=42
36;;;;;sensor1=43;newval=14
```

Communication starts with an identification line (ident line) being sent from transmitter to receiver, indicating the protocol version used, plus some additional information. Depending on the information to be transmitted the transmitter might need more information before it can proceed. The receiver can send this information by replying with its own ident line including the missing information as key/value pairs (this is the only time the receiver needs to send anything to the transmitter). Following that is the list of column fields that are going to be reported in every data line sent.

The first data line contains values for all known values. The second line contains updates to the first data line only (column fields), plus events (key/value pairs). Note the missing entries in the 2nd through 4th column fields, indicating that these values have not changed. In this example timestamp, lat, lon, vel, and dir are considered context and active, descr, sensor1, and newval are considered events.

2. Communication

2.1. Direction

Data flows in one direction only, from transmitter to receiver. The only exception is ident lines which can be sent from client to server. The fields available there are dependent on the actual implementation.

The transmitter/receiver nomer indicates the transmitter and receiver of the data flow. The connection set-up is independet of that. In general however it is advisable to setup the connection in the same direction as the data flow to make sure the transmitter becomes aware of a dropped connection and reopens the connection if lost.

2.2. Process

(This assumes that a (TCP) connection has already been set up.)

1. The transmitter sends an ident line, including parameter values.
2. The receiver responds with its own ident line potentially passing parameter values, like a starting timestamp.
3. If the transmitter decides that its parameters have changed due to the information sent by the receiver it sends again an ident line with the changed/additional information; the process continues at step 2. Otherwise at step 4.
4. The fieldlist is sent from transmitter to receiver, identifying the information that is sent in the column fields at the start of each line. The client recognises the fieldlist by the lack of 'ltrans=' (typical for ident lines) at the beginning of the line.
5. The first data line must be sent immediately, containing values for all known column fields and key/value pairs. Column fields with an unknown value are left empty. If no values are known an empty data line should be sent.
6. Additional data lines are sent if updates are available.

At any time the transmitter or receiver may close the connection indicating protocol error, an inability to handle the request or other error.

2.3. Format

2.3.1. Key/value format

Keys

Characters used in keys must be limited to:

```
'a'-'z'
'A'-'Z'
'0'-'9'
':' (colon)
'.' (period)
'_' (underscore)
```

Other characters may be acceptable depending on implementation but should not be used. ':' and '.' are assumed to indicate hierarchy, for example:

```
gps:lat
gps:lon
gps:sats:active
```

The following characters must NOT be used in keys:

```
'='
';' (semi-colon)
' ' (space)
',' (comma)
'\r' (CR)
'\n' (LF)
```

Values

The following characters must NOT be used in values:

```
\0
';'
\r
\n
```

Note

Any escaping of keys and values needs to be done at the application level, above the ltrans protocol level, where the data is produced. Injecting any of the 'must-not' characters in keys or values leads to undefined results. Ltrans protocol layers should discard or replace the unwanted characters above.

2.3.2. Ident line

The first line after connection setup is the identification line specifying the protocol version understood and identifying parameters for the connection:

```
ltrans=1.0;unitid=00:11:22:33:44:55:66;timestamp=1205306392
```

Note: 'timestamp' is handled specially: The 'timestamp' in the ident line sent by the transmitter is the starting timestamp. Any 'timestamp' field (either in a column field or key/value pairs) passed in each data line is relative to this value. If the 'timestamp' field in the ident line is not present, the value 0 is assumed for this initial timestamp.

A receiver must respond to an ident line by sending an ident line itself, echoing the information, and potentially including additional information if applicable to the communication.

In the case where fields sent in the first ident line need to be updated (most notably the timestamp field, if the reciever has specified a different starting timestamp), the transmitter can resend the ident line. The receiver can distinguish the ident line from the fieldlist line (below) by looking for the mandatory 'ltrans=' at the start of the ident line.

2.3.3. Fieldlist line

The next line is the fieldlist line, containing a list of keys that determine the column fields in each data line. Example:

```
timestamp;lat;lon;vel;dir
```

indicates that in each data line transmitted the first 5 columns are column fields, with keys 'timestamp', 'lat', 'lon', 'vel', 'dir'. See also the example above for the 'data line'.

To change the list of transmitted fields after the fieldlist has been sent the connection must be closed reopened with a new fieldlist. This can occur if the transmitter dynamically reconfigures itself for additional fields.

2.3.4. Data line

Every data line contains a fixed number of column fields followed by 0 or more key/value pairs. The number of and names for the column fields is defined in the fieldlist line. The number of events is not limited.

The first data line is sent immediately containing values for all known variables. If no columns where specified in the fieldlist, and no values are known, an empty line must be sent.

Example (first line, containing values for all known keys):

```
34;50.75123;4.67843;0.0;186;active=0;descr=Text with spaces;sensor1=42
```

Column fields are: `34, 50.75123, 4.67843, 0.0, 186`. The remainder of the line is events. This results in the following data being transferred:

```
timestamp = 34 + 1205306392
lat = "50.75123"
lon = "4.67843"
vel = "0.0"
dir = "186"
active = "0"
descr = "Text with spaces"
sensor1 = "42"
```

Please note that all values are considered binary data, in most cases representable as strings, except 'timestamp' as the value of the 'timestamp' field in the ident line (or 0 if that is not present) is added to the 'timestamp' field in every data line.

To compress the communication only updates are passed on. Example (second line, containing updates):

```
36;;;;;sensor1=43;newval=14
```

resulting in the following data

```
timestamp = 36 + 1205306392         # updated
lat = "50.75123"                    # unchanged
lon = "4.67843"                     # unchanged
vel = "0.0"                         # unchanged
dir = "186"                         # unchanged
active = "0"                        # unchanged
descr = "Text with spaces"          # unchanged
sensor1 = "43"                      # updated
newval = "14"                       # new value
```

- '\n' is the line separator. Splitting the input on '\n' divides the input data into lines. '\n' cannot be included anywhere in the input other than as a line separator.
- '\r' (or any other control characters or high-ASCII) should not be transmitted anywhere in the

data to avoid communication problems over serial lines, or to confuse receivers parsing data coming from Windows and Unix hosts. Receivers should cope with this kind of input data nevertheless by ignoring these characters.

- ';' is the field separator. Splitting the line on ';' divides the line into the column fields and pairs. ';' cannot be included anywhere in the line other than as a field separator.
- '=' is the key/value pair separator. a key/value pair should be split on the first '=' in the field. The left part is the key, the part on the right, potentially containing additional '=' characters, is the value.

2.4. Transmitter

The transmitter keeps a list of key/value pairs. If one of the values has changed, the transmitter starts generating a new line. For each column field either the value is transmitted if it has changed or an empty field is transmitted if it hasn't. Each field is separated by a ';'. If any other key/value pairs need to be transmitted these are appended to the line preceding each by a ';'. The line is terminated with a '\n'.

2.5. Receiver

The receiver receives a line by waiting for an '\n' character. The '\n' character must be chopped off.

The received line must be split on ';'. The first fields are to be entered as the values for the keys in the fieldlist line if any value is present. An empty string is considered no value (compare to 'empty value'). Any fields following must be considered key/value pairs. Each key/value pair must include at least one '='. Splitting on the first '=' in the string separates the key from the value.

On any parsing error the connection should be dropped.

2.6. Implementation notes

- Line length is unlimited. If a line exceeds the amount of buffer space available the connection should be closed.
- The frequency of transmission should be once a second or less frequent.
  Note: Internally, between daemons this can be higher if that's more convenient.
- Timestamps are given relative to the timestamp in the ident line.
- If a column value needs to be set to empty this can be done through setting it as a key/value pair in the events.