



VAASAN AMMATTIKORKEAKOULU
VASA YRKESHÖGSKOLA
UNIVERSITY OF APPLIED SCIENCES

Michael Ambaye

DEVELOPING A REMOTE MAINTENANCE SYSTEM

Technology

2014

ACKNOWLEDGMENTS

Foremost I would like to sincerely express my gratitude to my supervisor, Moghadampour Ghodrat, for his continuous support, advice, and patience throughout the entire process of this thesis. He has always been responsive even in times I know to be very hectic and busy. He has done everything possible in the development and writing processes of this thesis. I could not imagine a better advisor for this thesis.

I would also like to thank the entire work force in Devatus Oy. They have been a true inspiration for the past year. Mika Filander, CEO at Devatus, who is a true genius and the most hard working person I have come in contact with. I have learned a lot from him. Janne Kauppila, Senior Project Manager at Devatus, who has shown me what perfection at work truly means.

Lassi Pukkinen, CEO at MaintScope Ltd, and Anssi Pukkinen, CTO at MaintScope Ltd, who has come up with the idea of this project to begin with has been a delight to work with. The long hours Anssi Pukkinen put to work every day to have the project up and running was inspirational, and I would like to thank him for the support and opportunity he has given me to work with the web application and mobile application.

Finally, I would like to thank my father, Afework Ambaye Abay, for all the moral support, encouraging advices and the good and bad times he has stood by my side.

ABSTRACT

Author Michael Ambaye
Title Developing a Remote Maintenance system
Year 2014
Language English
Pages 70 + 1 Appendix
Name of Supervisor Moghadampour, Ghodrat

Computerized Maintenance Management Systems (CMMS) is a method by which we can bust the efficiency of a company's maintenance system. By using CMMS a company can easily predict future maintenance dates, costs and so on. For example, if a company owns a machine worth millions of dollars, then they intend to keep it working all the time. But because of accidental breakdown they could lose not only repair cost but also disappointed customers. /5/

This project was meant to address issues of monitoring the state of machines on a factory floor. This is needed so that the management can have better statistical information about how much their machines are being used, how often problems occur, how much time the machines stay idle and so on.

To be able to do this a remote terminal unit called, Teltonika FM 1100 was chosen.

To be able to do this a remote terminal unit called, Teltonika FM 1100 was chosen. This device was configured to send status information about the machine it is connected to through the digital and analog input interfaces. After receiving the status information, the data is saved to a database and manipulated to produce maintenance related information to the customer. This information could be something like, how long a machine stays functional without breaking, what part of the machine needs repairing, what caused the damage, how frequently does it happen, how long will it take to fix it, how many personnel should be assigned for that particular task, and what spare parts and tools is to be used and which department to contact, etc..

The management then can make an easy and informed decision on maintenance related issues and production will flow with less problems. The customers of that company would also be in turn satisfied for getting a service that is efficient and timely. This application is complete and is up for sale on the Maintbox official website. <http://www.maintbox.com/>

Keywords Teltonika FM1100, Digital Input (DIN), Analog Input (AI), Remote Terminal Unit (RTU), Servlet, Database

CONTENTS

ABSTRACT	4
1 INTRODUCTION	7
2 RELEVANT TECHNOLOGIES	9
3 APPLICATION DESCRIPTION	11
3.1 Teltonika FM 1100	11
3.2 Configurator	13
3.2.1 System	15
3.2.2 GSM.....	17
3.2.2.1 GPRS	17
3.2.2.2 SMS	18
3.2.2.3 Operator List	19
3.2.3 Data Acquisition Modes	20
3.2.3 IO.....	23
3.2 Sequence Diagram	25
3.3 State Diagram	27
4 Implementation	29
4.1 Implementation of Different Parts of the Servlet	31
4.1.1 TLReceiver Service Listener.....	32
4.1.2 Singleton.....	36
4.1.3 Request Handler Thread.....	38

4.1.4	The Database	50
5	TESTING.....	61
5.1	One DIN	61
5.2	Two RTUs.....	64
5.3	Saving State on the Server	66
6	CONCLUSION.....	69
	REFERENCES	71

1 INTRODUCTION

Devatus Oy is a private owned software company located in Vaasa. The main services the company provides are mostly mobile and web applications. Usually the services provided are based on clients requests. But recently the company has been doing its own application software for different platforms which can be used to incorporate the general wish of various companies and individuals who think will be able to utilize the functionalities provided. /6/ The idea of monitoring big machineries has come up and hence this project came into the picture.

The purpose of this task is to control the status of a machine that is worth a lot of money and is in production process all the time. Weather the machine is on, off, malfunctioning, etc can be monitored by connecting a small remote terminal unit (RTU), in our case, the Teltonika FM 1100 device.

This device has three digital inputs and one analog input, which we are interested in. The first digital input is reserved for ignition status monitoring, which will indicate whether or not the machine is turned on.

This RTU also has an external GSM (**G**lobal **S**ystem for **M**obile **C**ommunications) antenna connected to it, which means you can insert a SIM card into it. Therefore, the data that come though the digital input and analog input can be sent via a GSM or SMS as your wish.

For example: If digital input is 1, which indicates the machine is turned on, this information will be sent via the GSM antenna.

To receive this data there is a Java servlet code written which will be running on a Tomcat server, which is the main task of this project. This servlet will receive the signal from the RTU and places the data in the database. In the case of the previous example, we will have a value 1 stored in the database for digital input 1.

Now that we have the data in the database, we can pick it up using a web service and display it in a way that can be easily interpreted by anyone.

In the previous example the data can be interpreted as ON, and we can add a timestamp to it to show when the event has happened. which can be shown for how long it has been on. Or it can be color-coded using green to show the interval it has been on and the color red to show the time interval it has been off.

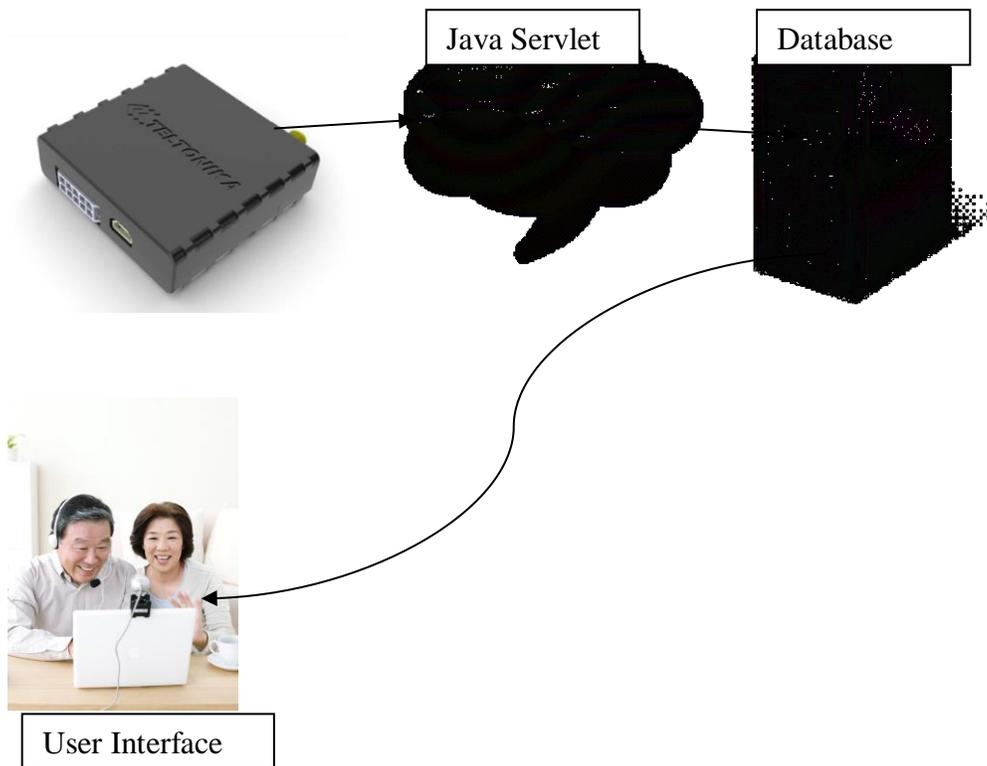


Figure 1. Basic idea of the project.

2 RELEVANT TECHNOLOGIES

Since this project incorporated an RTU sending a signal to a web service which then interprets the signal and stores the data to a database, from which a web application picks up and visualizes it to a customer, it had to involve few relevant technologies.

- FM11XX_Configurator_1.1.7.22 was used to configure the appropriate settings of the RTU. This RTU can be used for different purposes such as, car tracking, road assistance, international logistics, fleet management..etc. Therefore, the right configuration parameter should be applied to fulfill ones purpose. The RTU used for this project is called FM1100, this configurator, FM11XX_Configurator_1.1.7.22, was used to adjust the parameters such as where to send the data (IP address and port), what protocol to use (TCP, UDP), how long should the RTU save the data before it sends it to the server, which PINs should trigger the sending of data, and so on. /1/
- Eclipse was used to write the Java server programming.

The entire servlet was written in Java programming language and therefore Eclipse was used for writing the program. This software is free to use and has so many functionalities that makes a programmer's job much easier.

- Apache Ant was used to build the code written in Java and converting to a war file.

Creating any WAR file is similar to creating a JAR file since they both work in the same context as a ZIP file. We simply need some extensions specific to creating a WAR file plus all the attributes needed to create a JAR as well.

some extension attributes that are specific to the WAR task are:

- webxml: contains the web.xml file
- lib: these are libraries that go into WEB-INF\lib folder
- classes: classes that are compiled into WEB-INF\classes folder
- metainf: contains instructions that are capable of generating the MAINFEST.MF file. /2/

- Apache Tomcat Server: After getting the war file which is built by using the apache ant, we can put the war file in the apache server which will be running for as long as the server is running. The server programming is written to listen to certain port at all time, which will react if some data is sent from the RTU and act accordingly.

The Apache Tomcat Server is an open source web server which is used to run any Java based web applications, JAVA EE /7/. The war file is placed here with the same IP address as the one placed in the configurator and is listening to a port similar to the one configured in the RTU box configurator.

- MySQL Server: The database will be running on this server. If the Java server programming running on the Apache server that is listening on some port, receives a digital input state which is different from the previous state of the same digital input, the value will be stored in the database running on this server. The configuration parameters placed in the .properties file specify where this server will be placed and which port it will be open on. The same configuration should also be made in the MySQL server to make sure that these two are communicating.

3 APPLICATION DESCRIPTION

3.1 Teltonika FM 1100

The device (teltonika FM 1100) comes with the box, wires connecting all the inputs, outputs and power, GPS and GSM antennas. The device can also be configured for the intended purpose. /1/

The device has a slot for a SIM card as shown on the figure below

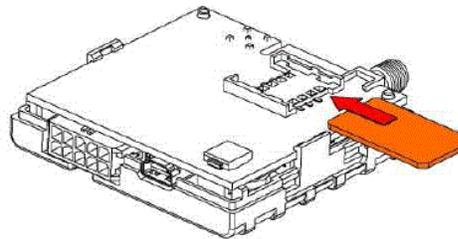


Figure 2. SIM card slot /1/

The wires that come with these devices are for the ten sockets shown bellow.

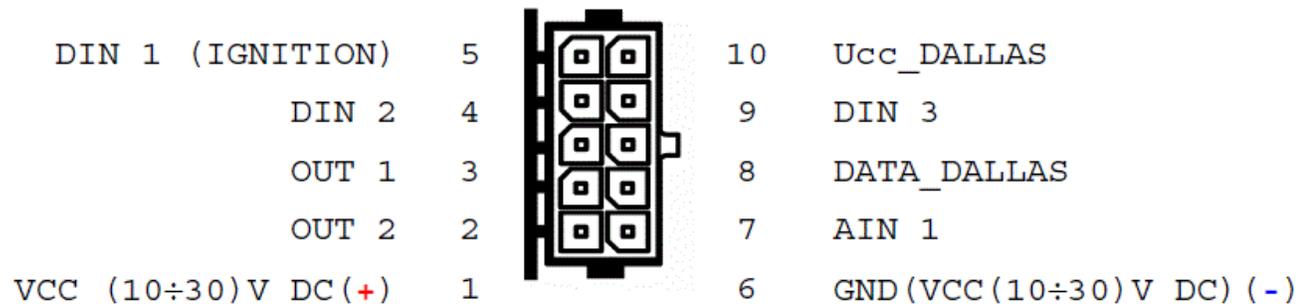


Figure 3. 2X5 Socket Pinout /1/.

The description of the sockets can be seen in the table below:

Table 1. Socket 2X5 pinout description /1/

Pin Nr.	Pin Name	Description
1	VCC (10÷30)V DC (+)	Power supply for module. Power supply range (10...30) V DC
2	OUT 2	Digital output. Channel 2. Open collector output. Max. 300mA.
3	OUT 1	Digital output. Channel 1. Open collector output. Max. 300mA.
4	DIN 2	Digital input, channel 2
5	DIN 1	Digital input, channel 1 DEDICATED FOR IGNITION INPUT
6	GND(VCC(10÷30) VDC (-))	Ground pin. (10÷30)V DC (—)

7	AIN 1	Analog input, channel 1. Input range: 0-30V/0-10V DC
8	DATA_DALLAS	Data channel for Dallas 1-Wire® devices
9	DIN 3	Digital input, channel 3
10	Ucc_DALLAS	+ 3,8 V output for Dallas 1-Wire devices. (max 20mA)

This device connects with the computer using a mini USB connector, that way the device can be configured visually. The packets can also be seen being generated using tools such as Putty.

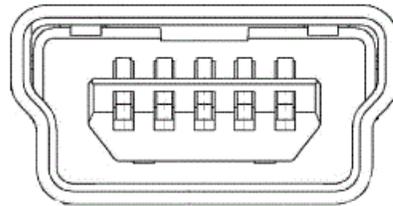


Figure 4. Mini USB type B connector. /1/

3.2 Configurator

The RTU can be connected to the configurator through a USB cable. When the IMEI and version fields which were empty are now filled, it means the connection is successful.

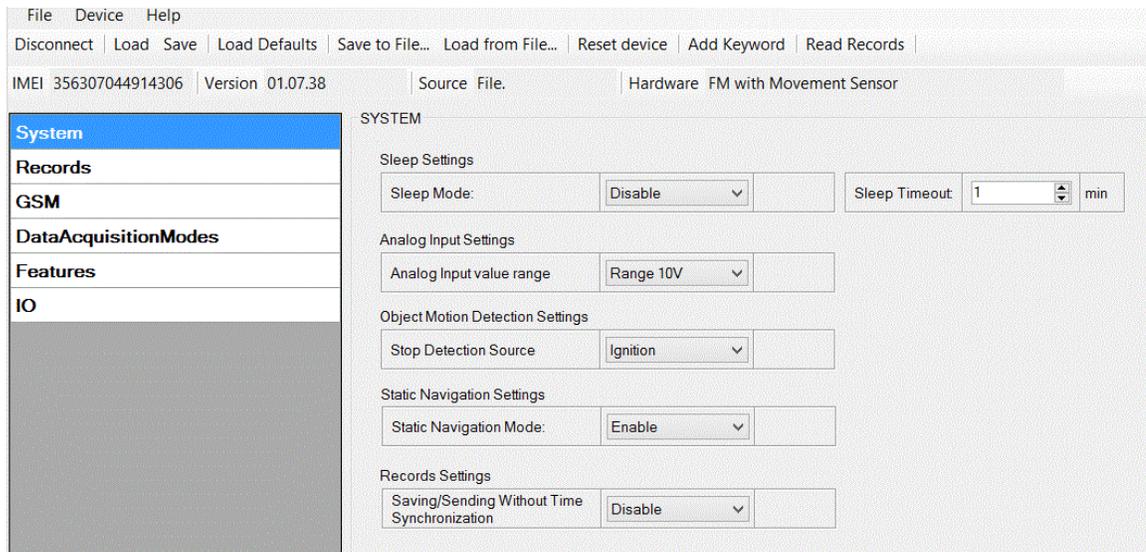


Figure 5. Teltonika FM11XX Configurator. /4/

In Figure 5 the configurator can be seen having four different areas,

- The main button
- Information area
- Setting menu and
- Parameters and values area.

In the main button we have:

- Connect/Disconnect : connects/disconnects the device.
- Load: Loads the setting of the device in the configurator.
- Save: saves the new parameters to the device.
- Load Defaults: loads the default values of FM1100 to the device
- Save to File: enables the settings made to be saved in a file with xml format.
- Load from File: loads the setting in a file to the configurator which can be saved to the device.
- Reset Device: resets the device.

In the setting menu we have:

- System
- Records
- GSM
- DataAcquisitionModes
- Features
- IO

This setting shown in Figure 6 will open up the parameters and value menu, which we will see closely. Only those settings and parameters are focused on in the thesis that are exclusive to the project, therefore the entire setting will not be covered.

3.2.1 System

This will be the first window that opens up when starting the configurator. The modes that can be altered are placed on the right hand side of the configurator and they are also very straight forward.

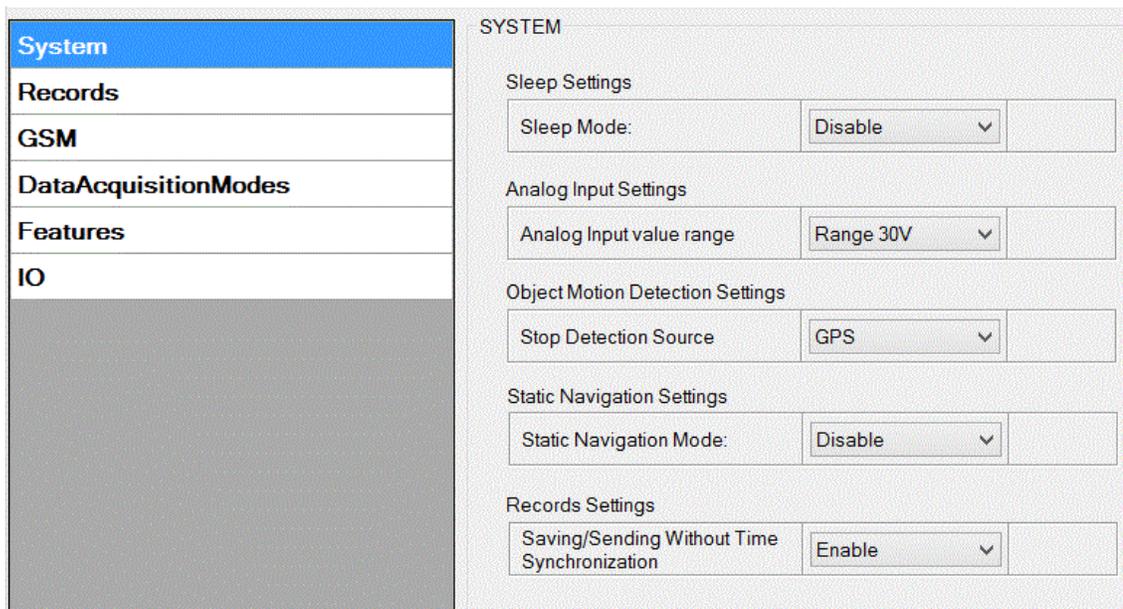


Figure 6. System Settings with parameters /4/

The main focus here is the analog input value range and stop detection source.

The analog input value range has two options: 0-10V or 0-30V, the lower range(0-10V) gives a higher accuracy of measurement. But the higher range(30V) has a much higher input resistance (167 kOhm) /1/. For that reason the higher range was chosen as can be seen in Figure 7.

This device can be configured to work as a tracking device for cars as described above, hence, the setting for stop detection source can have a different parameter which could be Ignition, Msensor or GPS. The ignition and Msensor (movement sensor) are usually used for tracking vehicles. We are using this device for tracking a status of a big machinery. Therefore, using a GPS is the appropriate setting for this purpose. /1/

3.2.2 GSM

In GSM (Global System for Mobile), we will be able to configure parameters such as GPRS, SMS, and operator list.

3.2.2.1 GPRS

In order for the device to send data over the GPRS antenna, the transport protocol has to be enabled, either Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). Because we need this data transmission to be as reliable as possible, we have chosen TCP as our protocol type.

The APN is dependent on the SIM card provider. If say, Saunalahti, is used, the APN becomes 'internet.saunalahti'. If Elisa is used, the APN will be 'internet'. Appropriate APN names are available on the Internet.

For security purposes it is advisable to set a user name and a password on the configurator as well.

The domain is the IP address of the server running on Tomcat which has the Java servlet. This device sends the data to that IP address and it can also be configured to be sending to a specific port.

System	GPRS data sending Settings	
Records	GPRS Context Activation	Enable
GSM	Protocol	TCP
> GPRS	APN:	internet
> SMS	APN user name:	
> Operator list	APN password:	
DataAcquisitionModes	Domain:	192.168.1.1
Features	Target Server Port:	8080
IO		

Figure 7. GSM/GPRS settings and values. /4/

3.2.2.2 SMS

The Teltonika FM 1100 can be configured to send the data through SMS to the server in case the GPRS is down. If SMS data sending setting is chosen to be enabled, it is advisable to set an SMS login and password on the device to have a secure connection. /1/

It is possible to then set the phone numbers that one wants to authorize and set which day and what time of the day one wants the device to use this setting, in case there is an information when the GPRS is going to stop or for any other reason.

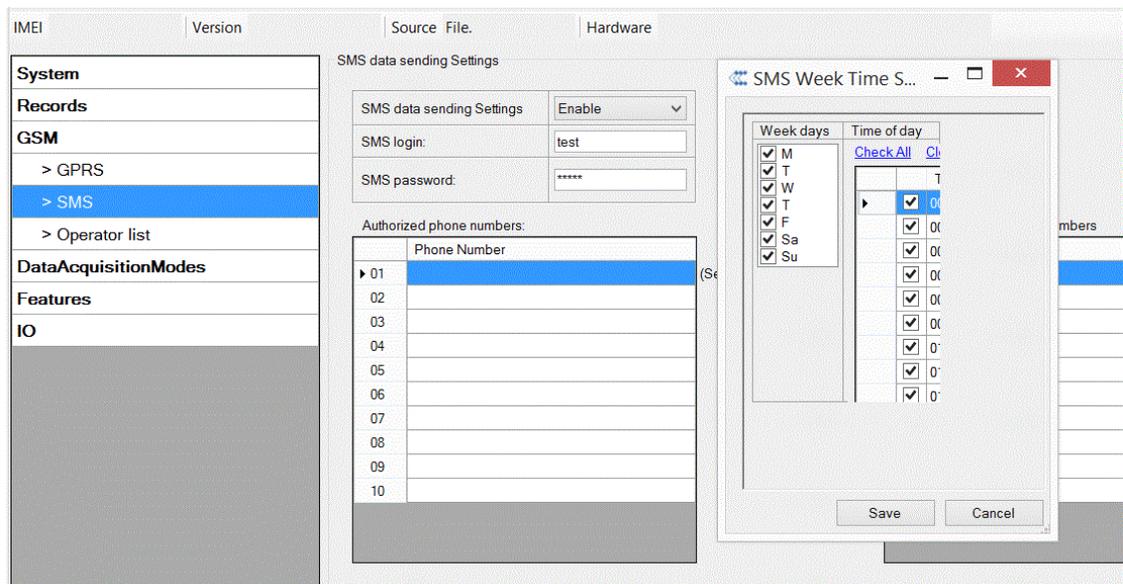


Figure 8. SMS Configurator /4/

3.2.2.3 Operator List

This setting is used for data acquisition mode switching which will be discussed latter on.

The Teltonka FM1100 always checks to see if there is a home operator code which has the highest priority. If the home operator was not set then it looks for operators on the preferred roaming operator list. The operator listed on top has the highest priority. If there are no operator code in there either, it will start using a GPRS to any GSM operator which is called unknown mode /1/.

It comes back looking for home operator list which has the highest priority and does this until one is found. FM 1100 checks for a connectivity to a home network connection every fifteen minutes.

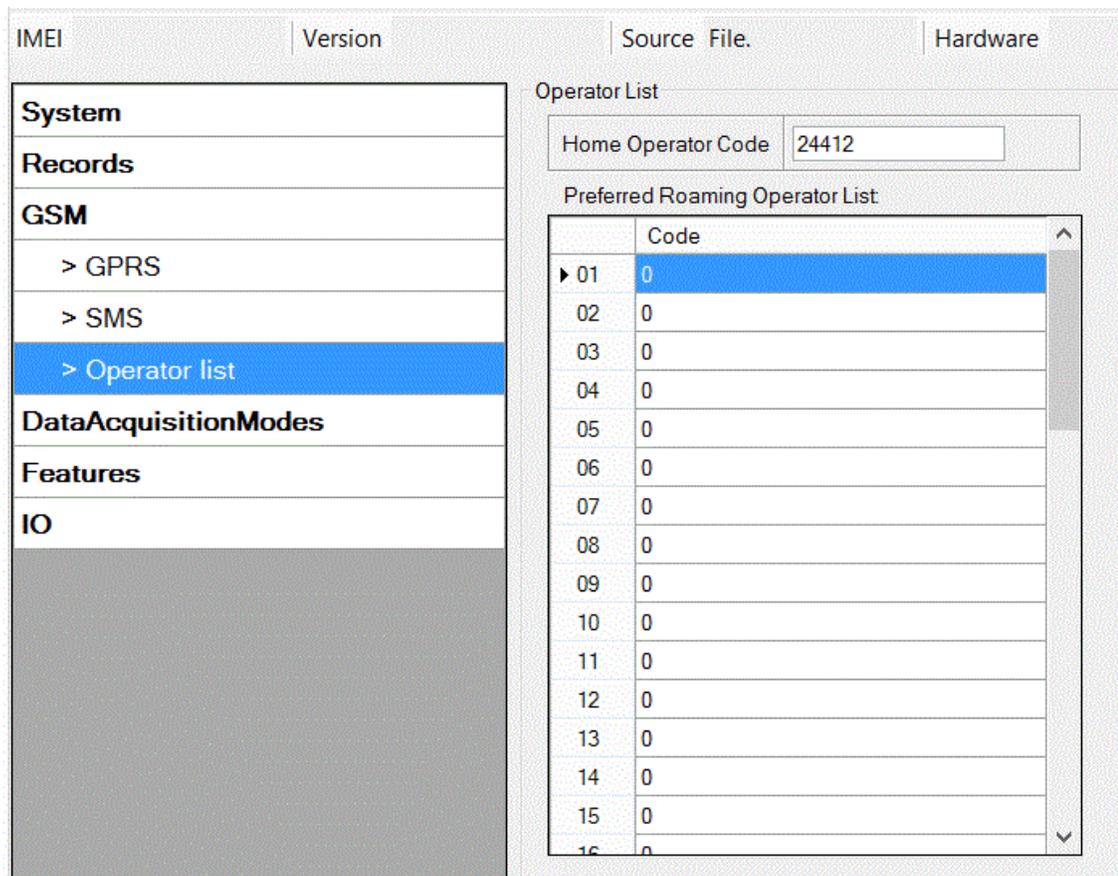


Figure 9. Operator List Setting. /4/

3.2.3 Data Acquisition Modes

This device can send data and save data as required. It can be configured how many records should be saved before sending them, the time it should wait before sending another data and so on /1/. It can be seen in the diagram that the devices can be moving or on stop. This is so because this device is widely used to track vehicles. In our case though, we are tracking machinery running on a factory floor. Therefore, we will be focusing on the ‘Vehicle on STOP’ settings only.

The device operators on HOME, ROAMING or UNKNOWN mode. These modes are set on the defined operator list described above.

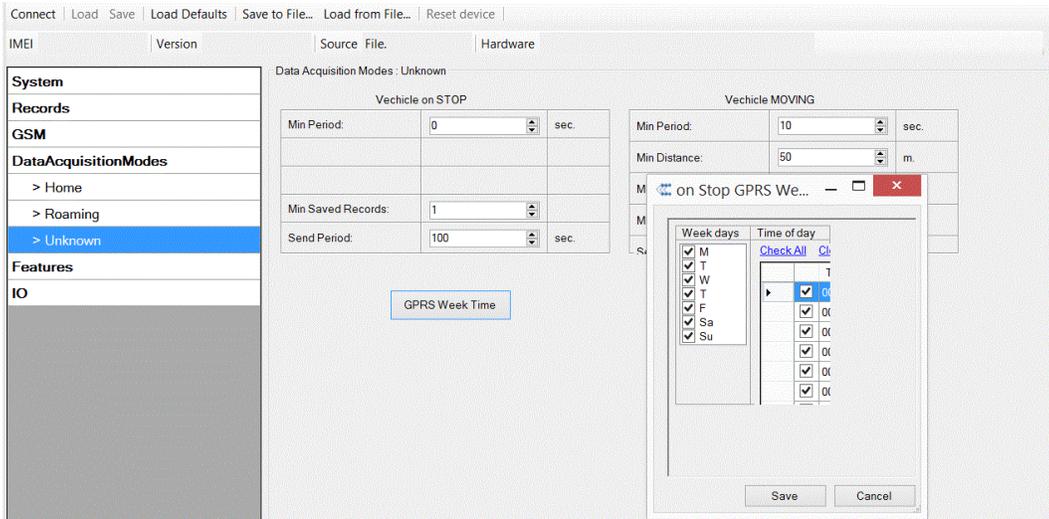


Figure 10. Data Acquisition Mode, Unknown. /4/

The data acquisition mode setting used for unknown mode looks like the one shown in Figure 10.

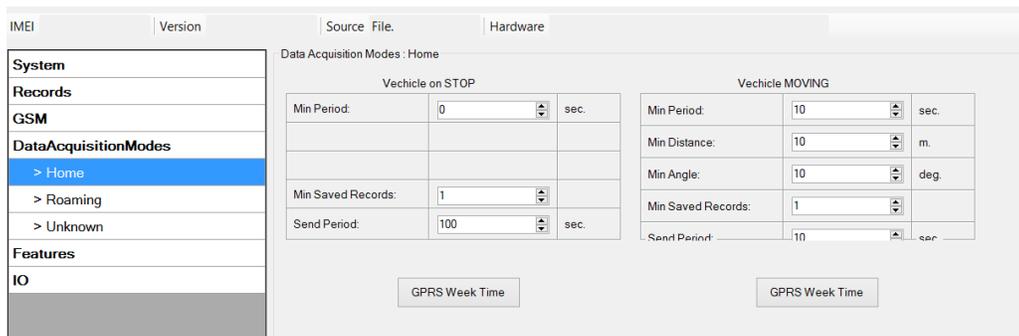


Figure 11. Data Acquisition Mode, Home /4/

The data acquisition mode setting used for unknown mode looks like the one shown in Figure 11. The setting for “Roaming” mode can be seen in Figure 12.

IMEI	Version	Source File.	Hardware
System			
Records			
GSM			
DataAcquisitionModes			
> Home			
> Roaming			
> Unknown			
Features			
IO			

Data Acquisition Modes : Roaming			
Vehicle on STOP		Vehicle MOVING	
Min Period:	0	sec.	
Min Saved Records:	1		
Send Period:	100	sec.	
Min Period:	10	sec.	
Min Distance:	10	m.	
Min Angle:	30	deg	
Min Saved Records:	1		
Send Period:	10	sec.	

Figure 12. Data Acquisition Mode, Roaming. /4/

‘Vehicle moving’ or ‘Stop’ state is defined by Stop Detection Source parameter which is on the system settings menu.

If the operators list is set to work on the home operator, the data acquisition will also work on home settings. If the operator list is set to work on the roaming, then so will the data acquisition. But if there is no setting on the operator list then the data acquisition mode will be set to unknown and it can be set what the minimum saved records and what the sending period record should be /1/.

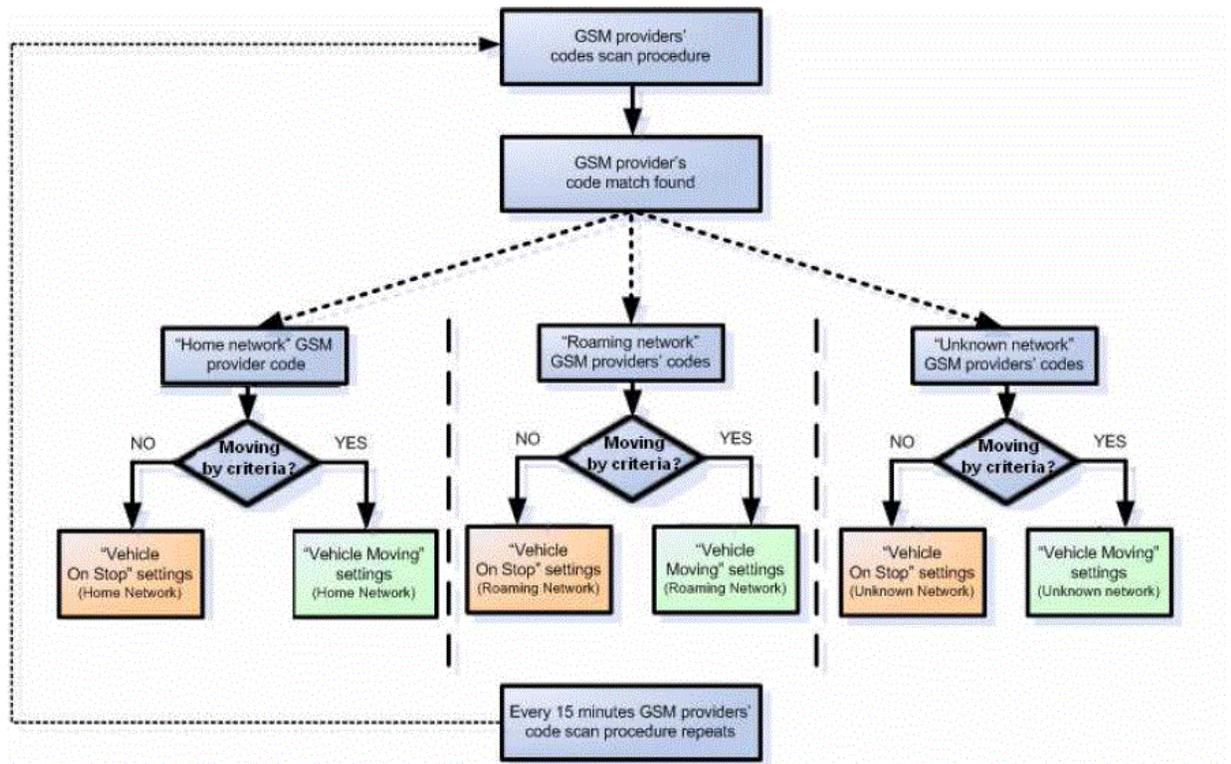


Figure 13. Data Acquisition Mode Configuration. /1/

3.2.3 IO

There are about 20 input/output signals we can enable/disable on this device. /1/ We are only interested in Digital Input 1 (DIN1), Digital Input 2 (DIN2), Digital Input 3 (DIN3) and Analog Input 1 (AI1). These digital and analog inputs have been enabled. which means our device will be sending information about those inputs that are enabled to our listener.

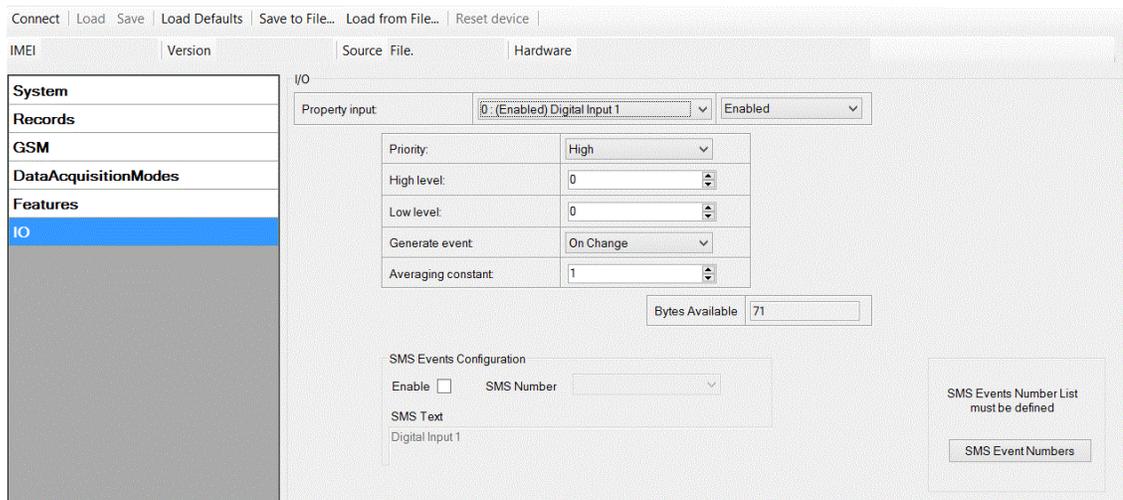


Figure 14. IO Configuration /4/

The priority has also been set to high. This means that, once the device senses a change happening on this inputs, it will stop whatever it is doing and sends the change that has occurred on these inputs to our server. There are a few options to set regarding priority.

Table 2. IO Settings. /1/

Setting	Value
Priority	low, high
High level	maximum threshold
Low level	minimum threshold
Generate event	on interval enter, on interval exit, on both enter and exit, hysteresis, onChange

Average constant	1 – 2 ₃₂ (4 Bytes)
------------------	-------------------------------

As can be seen in Figure 15, the priority is set to high, high and low level are both set to zero (0), and the averaging constant is set to 1 which is the default value.

The setting on generate event has been set to *onChange* as seen on Figure 15, this means that the device sends information if there was a change in state of the inputs. For example, if the state of Digital Input 1 (DIN1) is zero (0), the device will not send us any information for the next time unless the state of DIN1 has changed to one (1).

If the GPRS fails, the device can be configured to send signal using an SMS given that the SMS setting is enabled.

3.2 Sequence Diagram

The interaction between the RTU to the servlet, then from the servlet to the database can be demonstrated by a sequence diagram.

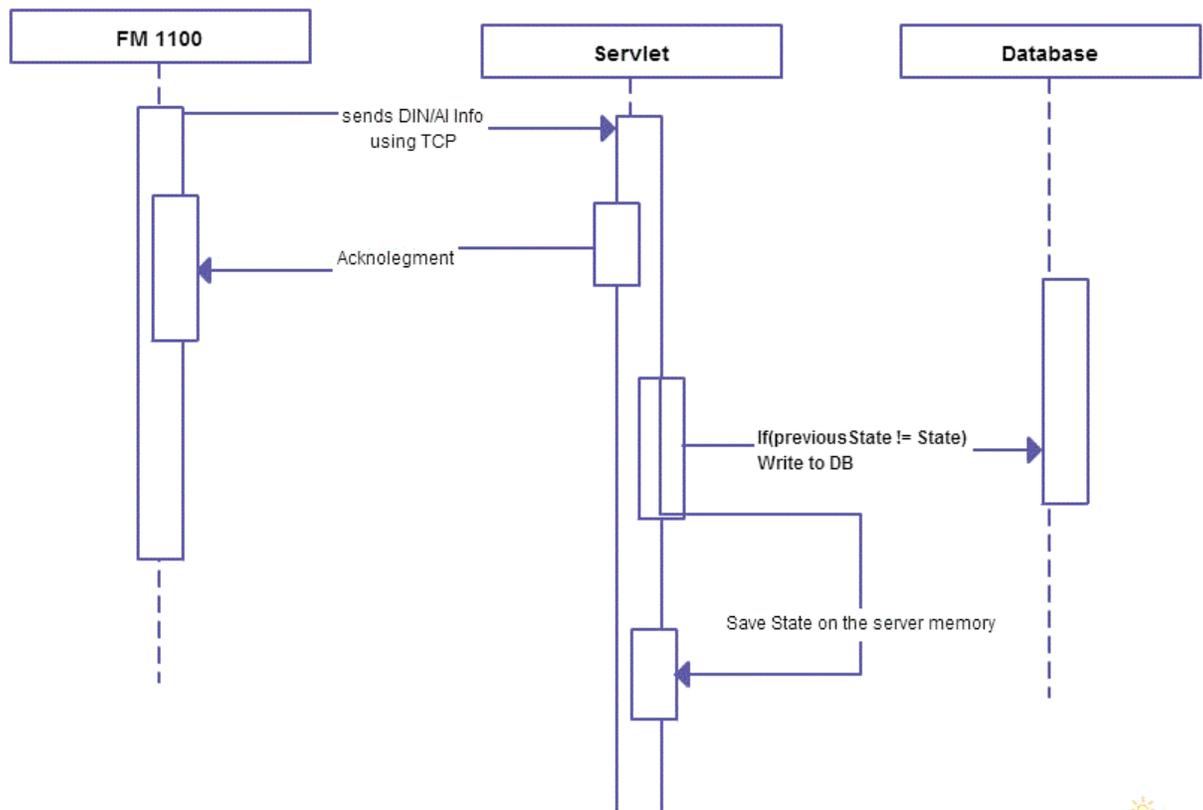


Figure 15. Interaction between components of the system.

The above diagram shows the sequence of the interaction between components of the system. The FM1100 box sends data using the TCP protocol to a machine (servlet) which has the same IP and is listening on the same port as the one set on the device. If the servlet has gotten the data sent then it sends acknowledgement of that back to the box. This goes on as long as the FM1100 box is sending the signal. On another thread, The servlet will take out the status data of each digital inputs and compares them to the previous statuses. If they are not the same, the new states will be saved on the server memory replacing the older one and will also be written to the database. If this is not the case, they will simply be discarded.

3.3 State Diagram

The bases for the RTU to send any signal is when there is any change in one of its DINs or the AI, which will force it to send the data to the servlet for further processing. This can be demonstrated using a state diagram, see Figure 16 below.

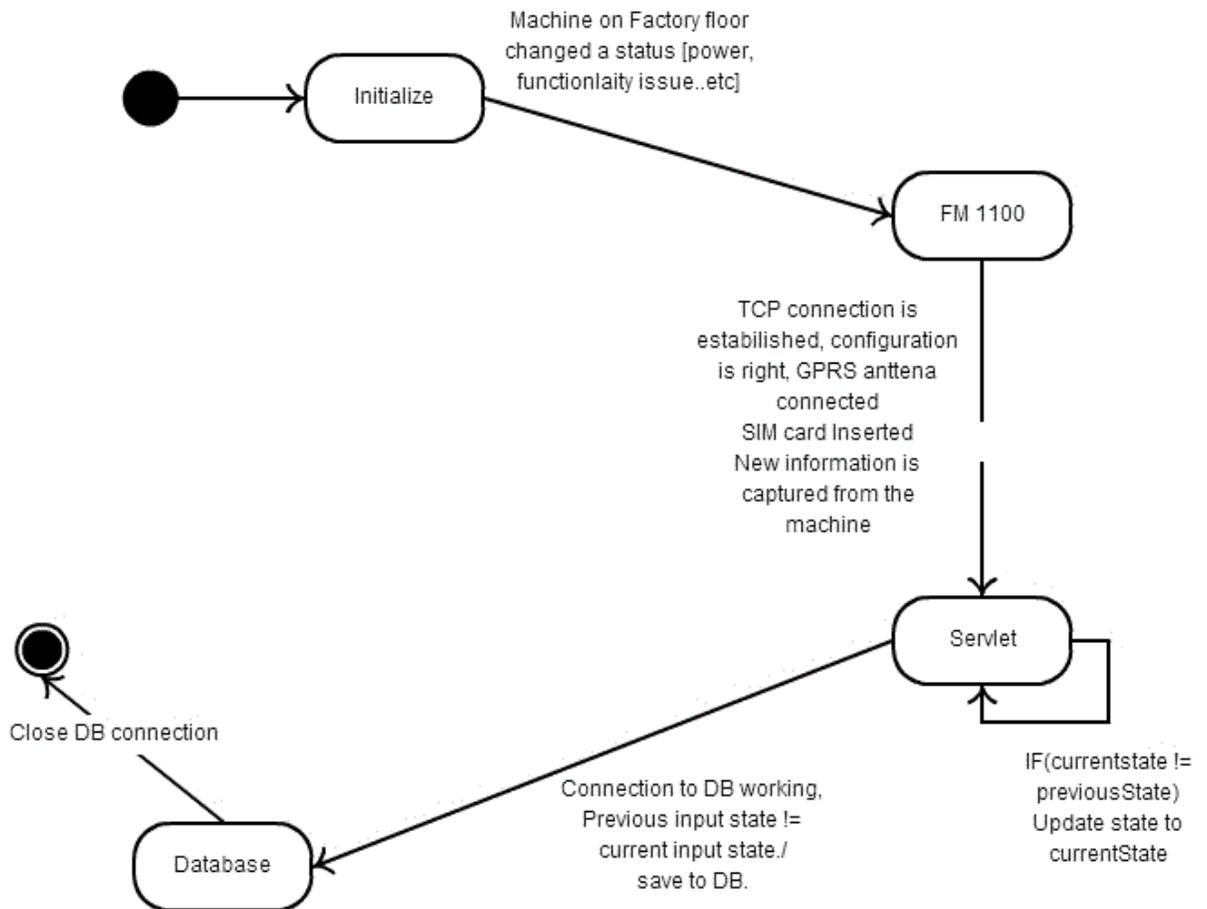


Figure 16. State Diagram.

The FM1100 can be configured to send status data only when there is a change in the digital or analog inputs. It will then look into the IP address and port number inside its configuration and tries to send the information using the GPRS antenna connected (which can only work if there is a SIM card inside) using the configuration protocol (TCP). If the servlet is running on the expected IP address and port number, it will compare all the digital and analog inputs with their respective previous states. If one or more of them has changed then it will be replacing the previous state and will also be written as a new entry to the database /1/.

4 IMPLEMENTATION

We have a servlet, which we call TLReceiver, running on a Tomcat server. This war file was build using Ant. To be able to do that there is:

- config: where all the configuration file are located including the configuration to connect to the database, setting where the location of proxool-configuration.xml file, setting for connecting with the server, and so on.
- lib: where all the Jar file are located
- build/classes: where all the Java files that are built are going to be placed
- Dist: where the final war file will be located
- ant: where the build.xml is placed and has the content of all file locations
- src: all the .java file are here

Eclipse was used to write the code and it makes building the ant file to generate a war file relatively an easy task.

We have a database running on the MySql server. The design of the database was done by another person, but for this purpose we will be using only one table called `rtu_event`, which saves the events of the digital and analog states of the device.

Here is the basic structure of the `rtu_event` table.

```
CREATE TABLE `rtu_event` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `rtu_imei` varchar(20) COLLATE utf8_unicode_ci
  NOT NULL,
  `rtu_input_id` int(11) DEFAULT NULL,
```

```

    `value` double NOT NULL,
    `evt_timestamp` datetime NOT NULL COMMENT 'time
recorded by rtu, if available',
    `created` datetime NOT NULL COMMENT 'record
creation timestamp',
    PRIMARY KEY (`id`),
    KEY `fk_rtu_event_remote_terminal_unit1_idx`
(`rtu_imei`)
) ENGINE=InnoDB AUTO_INCREMENT=205446 DEFAULT
CHARSET=utf8 COLLATE=utf8_unicode_ci;

```

Code Snippet 1. Code For Creating DB

- `rtu_imei`: is a unique number for every device purchased, two devices will never have the same `rtu_imei` number.
- `rtu_input_id`: determines which digital input or analog input was entered. 1 for DIN1, 2 for DIN 2, 3 for DIN 3 and 9 for AI1
- `value`: the state of the input entered which in the digital sense is either zero or one.
- `evt_timestamp`: the device sends a timestamp along with other information to the receiver. This timestamp is stored here on this column.
- `created`: when the input was entered into the database

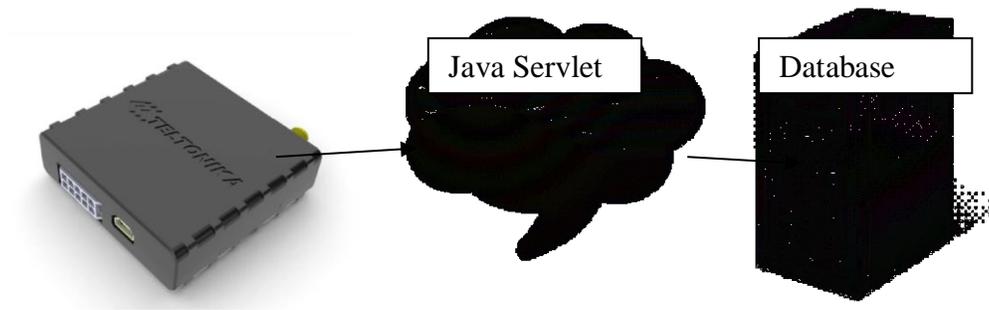


Figure 17. General Structure of the Project

To generalize how the structure of the project, we can see in Figure 16, that the Teltonika FM1100 device sends information using a GPRS antenna. This information is sent to the IP address of a certain port of a certain machine.

This machine contains a receiver we call TLReceiver, which is the Java servlet running on a Tomcat server. If this servlet hears information coming on that certain port of its IP address, then it processes this information. After analyzing whether or not the information is worth keeping it can make a decision whether or not to write it on the database.

4.1 Implementation of Different Parts of the Servlet

For security reasons and for the simplicity of programming we have separated the information involving usernames, passwords, IP addresses, port numbers, etc.. to a different file which can be read by the main program.

This makes the programming easier because, we need to sell this product for different customers. Therefore we do not have to change the entire code, but only go to the file named TLReceiver.properties and change the features that need changing. It also

makes it easier to protect the data since the configuration file is separate from the code.

4.1.1 TLReceiver Service Listener.

When the war file starts running on the Tomcat machine, it initializes all the necessary components.

```
try
{
    InputStream localInputStream =
    getClass().getClassLoader().getResourceAsStream("TL
Receiver.properties");

    this.applicationProperties.load(localInputStream);
}
catch (IOException localIOException)
{
    Log.info("Could not load configuration file:
" + localIOException.toString());

    localIOException.printStackTrace();
}
```

Code Snippet 2. loading TLReceiver.properties file

This code snippet will load the configuration file to access sensitive information such as the database username and password, port number and so on.

```

int portNumber =
Integer.parseInt(this.applicationProperties.getProp
erty("port_number"));

String proxoolConfigFile =
this.applicationProperties.getProperty("proxool_con
fig_file");

String proxoolDatabaseAlias =
this.applicationProperties.getProperty("proxool_dat
abase_alias");

String username =
this.applicationProperties.getProperty("user");

String password =
this.applicationProperties.getProperty("password");

```

Code Snippet 3. Accessing the sensitive info

It can be seen that after the file has loaded we can access the port number, configuration file, database alias, username and password for the database and so on.

```

    this.ds = new
    DatabaseServices(proxoolConfigFile,
    proxoolDatabaseAlias, username, password);

    this.tlrt = new TLReceiverThread();

    this.tlrt.init(portNumber, this.ds,
    this.applicationProperties);

    this.tlrt.start();

    Log.info("TLReceiverServiceListener
contextInitialized ready!");
}

public void contextDestroyed(ServletContextEvent
paramServletContextEvent)
{

```

```

        Log.info("START contextDestroyed");

        this.tlRt.exit();

        Log.info("END contextDestroyed");
    }

```

Code Snippet 4. Starting TLReceiver Service Listener with the info

The next thing we do as it can be seen from the code snippet above is to establish the connection with the database now that we have all the information we need from the file. Once we have that we start running another thread which is responsible for determining whether or not there is a socket open on the server, while the previous program is running. If there is an open socket it will process the data coming from there, if not, it closes the thread we started and starts from the beginning.

```

public void run()
{
    Log.info("TLReceiverThread running...");
    try
    {
        this.serverSocket = new
ServerSocket(this.portNumber);
        Log.info("TLReceiverThread Listening on TCP
port " + this.portNumber);
    }
    catch (IOException localIOException1)
    {
        localIOException1.printStackTrace();
    }
    while (this.running)
        try
        {

```

```

        Socket localSocket =
this.serverSocket.accept();
        RequestHandlerThread
localRequestHandlerThread = new
RequestHandlerThread(localSocket, this.ds,
this.applicationProperties);
        localRequestHandlerThread.start();
    }
    catch (IOException localIOException2)
    {
        Log.info("Error in TLReceiverThread: " +
localIOException2.toString());
    }
    Log.info("TLReceiverThread exited!");
}
}
}

```

Code Snippet 5. Listening on the open port

This thread will be running to check if there is an open socket on the server coming at a specific port initialized at the .properties file.

If there is an open socket, it will be calling a class called RequestHandlerThread which will take the local socket received from the open socket received, the database service connection and the application properties. That also runs on a different thread.

4.1.2 Singleton

We may have many devices running on a factory floor. These devices have a unique id called IMEI. Each device has DIN1, DIN2, DIN3 and AI1. This would mean we have to make sure we save the right input for the right device. We cannot afford to mix them, or worse, we cannot afford to skip a value that should be written to the database.

For that purpose, a class called imeiInfo was created, which contains the unique id (imei) and the all the digital and analog inputs.

The code snippet below is written to accomplish that.

```
public enum Singleton {
    INSTANCE;
    HashMap<String, Integer> updatecount = new
    HashMap<String, Integer>();
    ArrayList<imeiInfo> imeiState = new
    ArrayList<imeiInfo>();

    int i=0;

    public ArrayList<imeiInfo> getImeiState() {
        return imeiState;
    }

    public void setImeiState(ArrayList<imeiInfo>
    imeiState) {
        this.imeiState = imeiState;
    }
}
```

```
public static class imeiInfo{

    public String imei;
    public int din1;
    public int din2;
    public int din3;
    public int ai1;
    public imeiInfo(String imei, int
din1, int din2, int din3, int ai1) {
        super();
        this.imei = imei;
        this.din1 = din1;
        this.din2 = din2;
        this.din3 = din3;
        this.ai1 = ai1;
    }
    public String getImei() {
        return imei;
    }
    public void setImei(String imei) {
        this.imei = imei;
    }
    public int getDin1() {
        return din1;
    }
    public void setDin1(int din1) {
        this.din1 = din1;
    }
    public int getDin2() {
        return din2;
    }
    public void setDin2(int din2) {
        this.din2 = din2;
    }
    public int getDin3() {
        return din3;
    }
    public void setDin3(int din3) {
        this.din3 = din3;
    }
    public int getAi1() {
        return ai1;
    }
    public void setAi1(int ai1) {
```

```

        this.ai1 = ai1;
    }

}

```

Code Snippet 6. Singleton, saving DIN states

4.1.3 Request Handler Thread.

The variables that are in the class called imeiInfo which can be accessed using an arraylist of type imeiInfo,

```
private ArrayList<imeiInfo> requestImeiState;
```

A variable called ‘countimei’ was created, which will be used to count how many times a device has sent some data.

With this array list, it is possible to check whether or not it has something stored in it. If it is empty, we initialize the array list by putting the new imei of the device and a value of 2 for each inputs(DIN1, DIN2, DIN3 and AI1). We then can save this information to the singleton class using an instance called INSTANCE.

If the array list is not empty, it means we have already stored a data for some device in it. Therefore, we check if the stored imei value is the same value as the one we just received. If that is the case, we add 1 to the value of the variable countimei. That would make it impossible to get to the next if statement, which is designed to initialize everything.

```
int countimei = 0;

System.out.println("Module IMEI:" + imei);

if(requestImeiState.size()==0){

//requestListOfImeis = new ArrayList<String>();

//Singleton.INSTANCE.setListofImeis(requestListOfImeis);
    requestImeiState = new ArrayList<imeiInfo>();

    requestImeiState.add(new imeiInfo(imei, 2, 2, 2, 2));

    Singleton.INSTANCE.setImeiState(requestImeiState);

}

else{

    for(int i=0;i<requestImeiState.size();i++){

        if(requestImeiState.get(i).imei.equals(imei)){

            countimei++;

            break;

        }

    }

    if(countimei==0){

        //requestListOfImeis.add(imei);

    }

}
```

```

        Singleton.INSTANCE.setListofImeis(requestListofImeis);

        requestImeiState.add(new imeiInfo(imei, 2, 2, 2, 2));

        Singleton.INSTANCE.setImeiState(requestImeiState);
    }
}

```

Code Snippet 7. Saving the first DINs and AIs

```

New connection from module:Socket[addr=/85.76.66.223,port=4408,localport=8086]
Module IMEI:356307044914306
list of items 0: 356307044914306
Codec found: FM4 [Priority=10] [GPS element=[X=0] [Y=0] [Speed=255] [Angle=0] [Altitude=0] [Satellites=0]] [IO=] [Timestamp=]
Received records:8
Codec ID: 8
Priority: 1
Event source: 1
FM4 [Priority=1] [GPS element=[X=0] [Y=0] [Speed=0] [Angle=0] [Altitude=0] [Satellites=0]] [IO=[1=1] [2=0] [3=0] ]
Device timestamp: 2011-03-12 20:04:34
[X=0] [Y=0] [Speed=0] [Angle=0] [Altitude=0] [Satellites=0]

```

Figure 18. Tomcat Log-Event Source.

At the initial point at which the servlet is up and running we can see from the figure above that the RTU will send us all the information it has. Here it is assumed that DIN1 has changed, which is seen by the highlighted “Event source: 1”. Then we have [IO=[1=1][2=0][3=0]], this are the values of the state of DIN1 = 1, DIN2 =0 and DIN3=0. And since the values at the initial point at the servlet for all DINs is 2, these

values will be written to the database. And these states (DIN1 = 1, DIN2 =0 and DIN3=0) will be the new values of the DINs on the servlet as well.

The IMEI number can also be seen from there, which as explained earlier, is unique for every device. We also get information about the codec data , how many records we have received, the codec id, priority, and so on.

Event source is the name of the digital or analog input that did send the information to our listener (servlet). For example, in Figure 17, the information was sent to our servlet because digital input 1 has changed in state, from 0 to 1 or vice versa.

```
int eventSource = (int) ((AvLDataFM4)
LData).getTriggeredPropertyId();
```

let us consider a simple case where the event source came from analog input 1 (AI1). that would mean eventsource = 9

```
IOElement io = new IOElement();

io = avLData.getInputOutputElement();

// Analog data measurements

AnalogMeasurementSubUnit amsu = new
AnalogMeasurementSubUnit();

// Analog input 1

if(io.getProperty(9) != null) {

    int[] ana01Data = io.getProperty(9);

    if(LoggingEnabled) {

        System.out.println("Periodic analog
input 1: [" +
ana01Data[0] + " = " + ana01Data[1] + "]")
    }
}
```

```

        amsu.setValue("a9",
Integer.toString(ana01Data[1]));

        recordUnit.addSubUnit(RecordUnit.ANALOG_MEASUR
EMENT_UNIT_KEY,          amsu);
    }

    record.addRecordUnit(recordUnit);

    saveToDatabase(record);

```

Code Snippet 8. EventSource

The main idea of the code above is that, when there is an analog input value which is always going to be different from the previous one, we will capture the value of that input and save it to a place called Record unit which has a hash table and it uses the key "ANALOG_MEASUREMENT_UNIT_KEY" to save the value into that table. and the value will be Integer.toString(ana01Data[1]).

```

    public void addSubUnit(String key, Record value) {
        units.put(key, value);
    }

```

Code Snippet 9. Adding sub units in the hash table

This value will then be recorded in the database.

This is a very simple procedure compared to dealing with digital inputs, because in reality no two analog inputs can be the same, which means you don't have to save the previous value on the server and compare that to the value that you receive. You simply have to write whatever comes to the server.

In the case of digital inputs, there are only two possibilities for the states (value), one or zero. That will force us to compare the previous value with the current one.

```

        if( eventSource!= 0 ){
            for(int
                i=0;i<requestImeiState.size();i++)

                if(requestImeiState.get(i).imei.equals(imei)){

                    if(requestImeiState.get(i).imei.equals(imei)){

                        //Digital Input 1

                        System.out.println("the array value of din1: "
+
requestImeiState.get(i).din1);

                            eventSource = 1;

                            io = avldata.getInputOutputElement();

                            int[] dinIOData1 =
io.getProperty(eventSource);

                                if(loggingEnabled) {

                                    System.out.println("IO Event
[" + dinIOData1[0] + "=" +
dinIOData1[1] +"]");
                                }

```

Code Snippet 10. Comparing previous state with the current one

What we are doing here is simply, check if you can find the same device number that is sending the data as one of the devices inside the array list. if you find it, then look for the new state of digital input1 (DIN1) and save that state value in an array called *dinIOData1*

```

if(dinIOData1[1] !=
Singleton.INSTANCE.getImeiState().get(i).din1){

    DinSubUnit dsu1 = new DinSubUnit();

    dsu1.setDinInput(1);

    dsu1.setDinState(dinIOData1[1]);

    recordUnit.addSubUnit(RecordUnit.DIN_UNIT_KEY,
dsu1);

    System.out.println("Michael: previous DIN_UNIT
1: " +

    Singleton.INSTANCE.getImeiState().get(i).din1
;

    System.out.println("Michael: current DIN_UNIT
1: " + dinIOData1[1]);

    digitalinput1 = dinIOData1[1];

}

else

digitalinput1 =
Singleton.INSTANCE.getImeiState().get(i).din1;

```

Code Snippet 11. Saving the current DIN1 state in the hash table

Now that we have the state of DIN1 saved, we compare that one with the previous value of DIN1's state, which we have saved in a singleton previously. Therefore, we can get the value by calling *Singleton.INSTANCE.getImeiState().get(i).din1*.

If they turn out to be different then the state of the DIN1 will be stored in a hash table with a key *RecordUnit.DIN_UNIT_KEY*. Then the state of DIN1 will be saved in a variable called *digitalinput1*.

Now that we have checked all the things we need for DIN1, we need to do the same for DIN2 and DIN3. And the reason for that is there is a chance that more than one DIN can be pressed at the same time. If there will be more than one DIN pressed, the event source will have the value of the one that came first. For example, if all DINs (DIN1, DIN2 and DIN3) did change in state, the event source could only be either one of them. That why we have the line `eventSource = 1;` on our code. This will force the servlet to check for DIN1. Therefore, we should do the same for DIN2 and DIN3.

```

eventSource = 2;

io = avldata.getInputOutputElement();

int[] dinIOData2 = io.getProperty(eventSource);

if(loggingEnabled) {

    System.out.println("IO Event [" +
        dinIOData2[0] + "=" + dinIOData2[1] + "]");
}

if(Singleton.INSTANCE.getImeiState().get(i).din2 !=
    dinIOData2[1] ){

    DinSubUnit dsu2 = new DinSubUnit();

    dsu2.setDinInput2(2);

    dsu2.setDinState2(dinIOData2[1]);

    recordUnit.addSubUnit(RecordUnit.DIN_UNIT_KEY2
        , dsu2);
}

```

```

        System.out.println("Michael: previous DIN_UNIT
2: " +
        Singleton.INSTANCE.getImeiState().get(i).din2)
;

        System.out.println("Michael: current DIN_UNIT
2: " + dinIOData2[1]);

// Singleton.INSTANCE.setDin2(dinIOData2[1]);

        digitalinput2 = dinIOData2[1];

}

else

digitalinput2 =
Singleton.INSTANCE.getImeiState().get(i).din2;

```

Code Snippet 12. Saving DIN2 state to the hash table

```

//Digital Input 3 Event

eventSource = 3;

io = avldata.getInputOutputElement();

int[] dinIOData3 = io.getProperty(eventSource);

if(loggingEnabled) {

        system.out.println("IO Event [" +
dinIOData3[0] + "=" + dinIOData3[1] + "]");

}

        if(dinIOData3[1]!=Singleton.INSTANCE.getImeiSt
ate().get(i).getDin3()){

```

```

        DinSubUnit dsu3 = new DinSubUnit();

        dsu3.setDinInput3(3);

        dsu3.setDinState3(dinIOData3[1]);

        recordUnit.addSubUnit(RecordUnit.DIN_UNIT_KEY3
, dsu3);

        System.out.println("Michael: previous
DIN_UNIT 3: " +
        Singleton.INSTANCE.getImeiState().get(i).getDi
n3());

        System.out.println("Michael: current
DIN_UNIT 3: " + dinIOData3[1]);

        digitalinput3 = dinIOData3[1];

    }

    else

        digitalinput3 =
Singleton.INSTANCE.getImeiState().get(i).din3;

```

Code Snippet 13. Saving DIN3 state to the hash table.

The last two code snippets essentially are similar to the work we did for DIN1. It follows the same structure and procedure to save the name and value of the device input unit.

As you might have noticed, we have save the values of IMEI, DIN1, DIN2, DIN3 and AI1 in variables called imei, digitalinput1, digitalinput2, digitalinput3 and analoginput1 consecutively. Those values will be saved next to the array list which have created before that is of type class imeiInfo.

We also need to save all the imeiinfo variables into a singleton which will be later called with these values referred to as the previous values.

We also have these values saved in a hash table called recordUnit. this records will be added to another array list called record. this record will then be saved into a database.

```
requestImeiState.set(i, new imeiInfo(imei,
digitalinput1, digitalinput2, digitalinput3,
analoginput1));

Singleton.INSTANCE.setImeiState(requestImeiState);

record.addRecordUnit(recordUnit);
```

Code Snippet 14. Saving all DINs in recordUnit hash table

We have an established database connection when we started to run this application in the beginning. therefore, we simply call it here as dc.getConnection to get the database connection here. If we have a connection with the database we called the function called saveToDatabase and pass the arraylist called record as a parameter.

Once the record is saved we have to make sure the connection is closed. therefore, on the code below there is a function called, closeDbConnection().

But if for some reason, the connection is not working we need to have a way to inform the server that the connection to the database is not working and also it is a good idea to write the problem into a file. This is done with the line of code seen at the end of the code below.

```
Connection c = dc.getConnection();

// Create Database object to communicate with db.
```

```

Database d = new Database(c);

// Check database connection is available.

if(c != null) {

    // Save RecordSet to db.

    boolean dbSaving = d.saveToDatabase(record);

    // Close database Connection object.
    d.closeDbConnection();

    // Records successfully saved to db

    if(dbSaving == true) {

// Write length of received packets to Teltonika to
indicate that all records received.

        dos.writeInt(decoded.Length);

    }
else {
    System.err.println("ERROR: TLReceiver:
RequestHandlerThread: cannot get database
connection!");
    fh.saveStringToFile(saveMessageDirectory + "/"
+ "TLReceiver-" + System.currentTimeMillis() +
"_" + System.nanoTime() + ".xml", record.toXML());
}
}

```

Code Snippet 15. Database Connection and Record Writing

4.1.4 The Database

After receiving the information from the device and processing it in a way we have seen, we have to permanently save the information in to a database.

To do this we have a table called `rtu_event` on the database where all the information can be stored and latter read from as well. The structure of the `rtu_event` has been illustrated in Chapter 4.1

To illustrate a bit, the table has the following columns:

- `id`: This is a primary key, which increments itself.
- `rtu_imei`: which will be storing the sending devices IMEI number. We know this number is unique for all devices, but we still cannot use it as a primary key because one device can send multiple different values to the database.
- `rtu_input_id` which stores the name of the DIN or AI. We have assigned 1 for DIN1, 2 for DIN2, 3 for DIN3 and 9 for AI1.
- `value`: which will be storing all the digital and analog states sent from the device.
- `evt_timestamp`: which will be storing the timestamp that the device sent along with the real data
- `created`: which will be saving the time that the information was created/saved onto the database

We first need to load all the files and information to establish a reliable connection with the database. We will use this code to load the file containing user authentication information and all the database aliases.

```
public class DatabaseServices {
```

```
/**
```

```

        * Configuration file path of proxool.
        */
        private String proxoolConfigFile;

        /**
         * Database alias name in proxool
         configuration file.
         */
        private String proxoolDatabaseAlias;

        private String user;
        private String password;

        /**
         * Constructor of the DatabaseSaver
         */
        public DatabaseServices(String
        proxoolConfigFile, String
        proxoolDatabaseAlias, String user, String password)
        {
            // Set proxool configuration file.
            this.proxoolConfigFile =
        proxoolConfigFile;

            // Set proxool database alias (select
            database configuration to use).
            this.proxoolDatabaseAlias =
        proxoolDatabaseAlias;
            this.user = user;
            this.password = password;

            // Initialize proxool.
            initialize();
        }
        private void initialize() {
            try {

                JAXPConfigurator.configure(proxoolConfigFile,
                false); // The false means non-validating
            }
            catch (ProxoolException pe) {

```

```

        System.out.println("ERROR:
DatabaseServices: initialize(): ProxoolException: "
+ pe.toString());
    }
    catch (Exception e) {
        System.out.println("ERROR:
DatabaseServices: initialize(): Exception: " +
e.toString());
    }
}

```

Code Snippet 16. Establishing database connection using proxool config file

Now that we have accomplished that, we can go on and create a class that can be called from anywhere to connect with the database.

```

public Connection getConnection() {

    // Database connection object.
    Connection c = null;
    // Get database connection from pool.

    try {
        c =
        DriverManager.getConnection(proxoolDatabaseAlias,us
er,password);
    }
    catch (SQLException sqle) {
        System.err.println("ERROR:
DatabaseServices: getConnection(): SQLException: "
+ sqle.toString());
    }
    catch (Exception e) {
        System.err.println("ERROR:
DatabaseServices: getConnection(): Exception: " +
e.toString());
    }

    return c;
}

```

Code Snippet 17. More on database connection establishment

For example, in the Request Handler class we have used to save the records we have managed to collect to the database calling a function called `dc.getConnection()`.

This is a good way to write an application because, unless there is a need to read or write in the database there is no reason to have the connection open.

We have also seen that, after establishing a connection, we have had a code snippet like:

```
Connection c = dc.getConnection();  
  
// Create Database object to communicate with db.  
  
Database d = new Database(c);  
  
// Check database connection is available.  
  
if(c != null) {  
  
    // Save RecordSet to db.  
  
    boolean dbSaving = d.saveToDatabase(record);  
  
    // Close database Connection object.  
    d.closeDbConnection();  
}
```

Code Snippet 18. More on Databases

Once the record is ready to be written, a connection is established from the `databaseServices` class, and that connection is passed as argument to a constructor called `Database` in a class called `Database`.

```
public Database(Connection dbConn) {  
    this.dbConn = dbConn; }
```

Then we call the function called `saveToDatabase` and pass the record array list to it. We can see how Digital input 1 is saved to the database in the following code snippet.

```
public boolean saveToDatabase(RecordSet data) {

    int eventId;
    boolean status;

    ResultSet rs = null;
    PreparedStatement ps = null;

    String tempImei = "";
    String tempIbutton = "";

try {

    // Save all records into database
    int size = data.getSize();
    for(int i=0; i<size; i++) {

        // Get next RecordUnit from table
        RecordUnit ru = (RecordUnit)
data.getRecordUnit(i);

        // Get SystemSubUnit
        SystemSubUnit ssu = (SystemSubUnit)
ru.getSubUnit(RecordUnit.SYSTEM_UNIT_KEY);

        // Get system IMEI code for debug logging
        tempImei = ssu.getImeiCode();

        // Check IMEI code is valid (15 characters)
        if(tempImei.length() == 15) {

            // Check datetime exist
            if(!ssu.getDatetime().equals("null")) {
```

```

// SAVING DIN 1 TO THE DATABASE

    DinSubUnit dsu = (DinSubUnit)
ru.getSubUnit(RecordUnit.DIN_UNIT_KEY);
    if(dsu != null) {
        System.out.println("Michael: DIN_1
inside DB: " + dsu.getDinInput());

        System.out.println("Michael: Din State
of DIN1: " + dsu.getDinState());

        String sqlInsertStatement = "insert into
rtu_event
(rtu_imei,rtu_input_id,value,evt_timestamp,cre
ated) values (?, ?, ?, ?, ?)";

        // SQL statement for getting a new id
for the event based on the device IMEI code

        ps =
dbConn.prepareStatement(sqlInsertStatement);
        ps.setString(1, ssu.getImeiCode());
        ps.setInt(2, dsu.getDinInput());
        ps.setInt(3, dsu.getDinState());

        ps.setTimestamp(4,
Timestamp.valueOf(ssu.getDatetime()));
        ps.setTimestamp(5,
Timestamp.valueOf(ssu.getDatetime()));

        // Execute query
        ps.executeUpdate();

    }
else
    System.out.println("Michael: DIN 1 is null");

```

Code Snippet 19. Writing the records

From the code above, we can see that the data was passed as an argument of type RecordSet, which we have been saving as in the RequestHandler class. From there we can access the hash table we have stored in RecordUnit and SystemSubUnit. The SystemSubUnit hash table key contains the IMEI number of the device and the hash table in the RecordUnit contains the digital input names and their values.

A legitimate IMEI number is 15 characters long, if it is shorter or longer than that, we can discard it because it is not a real IMEI number. We can do this using:

```
SystemSubUnit ssu = (SystemSubUnit)
ru.getSubUnit(RecordUnit.SYSTEM_UNIT_KEY);
```

Then we can access the content of the ssu using:

```
tempImei = ssu.getImeiCode();
```

Then we check if we have anything inside Record unit:

```
DinSubUnit dsu = (DinSubUnit)
ru.getSubUnit(RecordUnit.DIN_UNIT_KEY);
```

If there is, we can access the content:

```
dsu.getDinInput(): Gives us which digital input it is (1, 2 or 3)
```

```
dsu.getDinState(): Gives us the value of the digital input (0 or 1)
```

Then we have the sql statement as seen above:

```
String sqlInsertStatement = "insert into rtu_event
(rtu_imei,rtu_input_id,value,evt_timestamp,created)
values (?, ?, ?, ?, ?)";
```

Code Snippet 20. Insert statement

and insert all the needed value:

```

ps.setString(1, ssu.getImeiCode()); //IMEI number
of the device sending the information
ps.setInt(2, dsu.getDinInput()); //What digital
input has changed in state to send the information
ps.setInt(3, dsu.getDinState()); //The current state
of the digital input (0 or 1)
ps.setTimestamp(4,
Timestamp.valueOf(ssu.getDatetime()));
//timestamp sent from the device
ps.setTimestamp(5,
Timestamp.valueOf(ssu.getDatetime())); //timestamp
will be updated to
now()in db

```

Code Snippet 21. Inserting the values to the DB

Similarly for Digital input 2:

```

DinSubUnit dsu2 = (DinSubUnit)
ru.getSubUnit(RecordUnit.DIN_UNIT_KEY2);
if(dsu2 != null) {

    System.out.println("Michael: DIN_UNIT2 inside
DB: " + dsu2.getDinInput2());
    System.out.println("Michael: Din State of
DIN2: " + dsu2.getDinState2());

    String sqlInsertStatement = "insert into
rtu_event
(rtu_imei,rtu_input_id,value,evt_timestamp,cre
ated) values (?, ?, ?, ?, ?)";

    // SQL statement for getting a new id for the
event based on the device IMEI code
    ps =
dbConn.prepareStatement(sqlInsertStatement);
    ps.setString(1, ssu.getImeiCode());
    ps.setInt(2, dsu2.getDinInput2());

```

```

        ps.setInt(3, dsu2.getDinState2());

        ps.setTimestamp(4,
Timestamp.valueOf(ssu.getDatetime()));
        ps.setTimestamp(5,
Timestamp.valueOf(ssu.getDatetime()));

        // Execute query
        ps.executeUpdate();

    }
    else

        System.out.println("Michael: DIN 2 is null");

```

Code Snippet 22. DIN2 database entry

And for Digital Input 3

```

DinSubUnit dsu3 = (DinSubUnit)
ru.getSubUnit(RecordUnit.DIN_UNIT_KEY3);

if(dsu3 != null) {

    System.out.println("Michael: DIN_UNIT3 inside
DB: " + dsu3.getDinInput3());

    System.out.println("Michael: Din State of
DIN3: " + dsu3.getDinState3());

    String sqlInsertStatement = "insert into
rtu_event
(rtu_imei,rtu_input_id,value,evt_timestamp,cre
ated) values (?, ?, ?, ?, ?)";

    // SQL statement for getting a new id for the
event based on the device IMEI code

```

```

        ps =
        dbConn.prepareStatement(sqlInsertStatement);
        ps.setString(1, ssu.getImeiCode());
        ps.setInt(2, dsu3.getDinInput3());
        ps.setInt(3, dsu3.getDinState3());
        ps.setTimestamp(4,
Timestamp.valueOf(ssu.getDatetime()));
        ps.setTimestamp(5,
Timestamp.valueOf(ssu.getDatetime()));

        // Execute query
        ps.executeUpdate();

    }

else
    System.out.println("Michael: DIN 3 is null");

```

Code Snippet 23. DIN3 database entry

To save analog input1 a different method is used. This is because the hash table we have for it is different than the digital inputs. And it is also stored in a different class called AnalogMeasurementSubUnit. The key used to store the name of the digital input has a value of "a9":

```

AnalogMeasurementSubUnit amsu =
(AnalogMeasurementSubUnit)ru.getSubUnit(RecordUnit.
ANALOG_MEASUREMENT_UNIT_KEY)

```

Now we can retrieve the name and the value using the amsu variable in a similar way as we did with the Digital inputs.

```

value = amsu.getValue("a9");
AnalogMeasurementSubUnit amsu =
(AnalogMeasurementSubUnit)

```

```

ru.getSubUnit(RecordUnit.ANALOG_MEASUREMENT_UNIT_KEY);

if(amsu != null) {

    String value = null;
    value = amsu.getValue("a9");
    if(value != null) {
        System.out.println("Michael: AI1 in DB:
" + 9);
        System.out.println("Michael: AI1 value:
" + amsu.getValue("a9"));
        String sqlInsertStatement = "insert into
rtu_event
(rtu_imei,rtu_input_id,value,evt_timestamp,cre
ated) values (?, ?, ?, ?, ?)";

        // SQL statement for getting a new id
for the event based on the device IMEI code
        ps =
dbConn.prepareStatement(sqlInsertStatement);
        ps.setString(1, ssu.getImeiCode());
        ps.setInt(2, 9);
//
        ps.setDouble(3,
Double.parseDouble(amsu.getValue("a9")));
        ps.setTimestamp(4,
Timestamp.valueOf(ssu.getDatetime()));
        ps.setTimestamp(5,
Timestamp.valueOf(ssu.getDatetime()));

ps.executeUpdate();

    }

    else
        System.out.println("Analog Input 1 is
NULL");
}

```

Code Snippet 24. AI database entry

5 TESTING

This application was developed for a company called Devatus Oy. The testing on the other hand has been done by another company called Tähtipiste Oy. Therefore, a device was used to simulate what would happen on a factory floor by connecting different wires into the digital inputs and using a switch. When that was done, the project would be sent to Tähtipiste for a test. This process went on until they were satisfied with the product.

5.1 One DIN

At the beginning of the project, there was only one RTU (FM 1100) that we were considering to work on. And that device will send one digital input at a time. Therefore, in the RequestHandlerThread class we considered using a code that will check only for an event source. If there is an event source, then the program checks which digital input matches the number and proceed from there.

```
switch (eventSource) {
    // Digital Input 1 Event
    case 1: {
        code goes here
        break;
    }
    // Digital Input 2 Event
    case 2: {
        code goes here
        break;
    }
}
```

```

    }

    // Digital Input 3 Event
    case 3: {
        code goes here
        break;
    }
}

```

Code Snippet 25. Test case for a single DIN trigger

Since the switch was connected on the wires of the RTU (Digital Input) , It was possible to change the state of a DIN from 1 to 0 or from 0 to 1. This meant that, no matter what happened there would not be a change for two DINs to be switched on or off simultaneously. But in the real machine, it was possible for that to happen.

```

FM4 [Priority=1] [GPS element=[X=0] [Y=0] [Speed=0] [Angle=0] [Altitude=0] [Satellites=0]] [IO=[1=1] [2=0] [3=0] ]
Device timestamp: 2011-03-11 11:04:23
Event source: 3
FM4 [Priority=1] [GPS element=[X=0] [Y=0] [Speed=0] [Angle=0] [Altitude=0] [Satellites=0]] [IO=[1=0] [2=1] [3=1] ]
Device timestamp: 2011-03-05 11:58:31

```

Figure 19. Testing one digital input at a time.

Consider the log captured from the server when all the digital inputs changed their states. The one above is the original status (previous state), the one below is the state we newly received from the device (current state).

It can be seen from the above figure that the DINs have changed a state.

But since our event source shown in Figure 18 is three(3), then the listener will think that it was only DIN 3 that has changed its state (from 0 to 1), when in reality all of the DINs have changed.

After getting the comment on that problem, the device was rewired by connecting two wires (say DIN2 and DIN3) together and connected them both to one switch. That means when a switch is pressed both DINs will change state simultaneously. After that there were some changes that were made, especially with replacing the switch statement seen above on the code snippet to if statement.

```
if( eventSource!= 0 ){  
  
    //Digital Input 1  
    eventSource = 1;  
    Code goes here  
    // Digital Input 2 Event  
    eventSource = 2;  
    Code goes here  
    // Digital Input 3 Event  
    eventSource = 3;  
    Code goes here  
}
```

Code Snippet 26. Check for change of state for all the DINs

As it can be seen from the code snippet above, we have replaced the switch statement with if statement. The condition being if event source is different from zero, which is never the case if there would be a change in state that the device sent.

Then we force the code to check every DIN state by setting the event source to the value of the DIN.

```

IO Event [1=1]
Michael: previous DIN_UNIT 1: 0
Michael: current DIN_UNIT 1: 1
IO Event [2=0]
Michael: previous DIN_UNIT 2: 1
Michael: current DIN_UNIT 2: 0
IO Event [3=0]

```

Figure 20. Each DIN being Checked.

It can be seen in Figure 19, that DIN 1 changed state from 0 to 1 and DIN 2 changed state from 1 to 0. The program forced all the DINs to be checked. Then it writes all of those into a database, see Figure 20.

```

Michael: DIN_1 inside DB: 1
Michael: Din State of DIN1: 1
Michael: DIN_UNIT2 inside DB: 2
Michael: Din State of DIN2: 0

```

Figure 21. Writing all the changed states to the DB.

5.2 Two RTUs

There was only one device (teltonika FM1100) at the beginning of this project. Which means there is only DIN and AI for one device to worry about, when in reality there could be many of these devices connected on a factory floor to many machines. And they all will of course have to send their data to on listener (servlet).

When Tähtipiste did the test and said that they were missing some states in the database, it took a while to figure out that it was because the code did not see that the information were coming from another machine. Instead it compared the state of one machine with the state of another machine.

For example: Device A has DIN1 = 0, DIN2 = 0 and DIN3 = 0

Device B has DIN1 = 1, DIN2 = 1 and DIN3 = 1

Now if Device B sent data saying the state of DIN 1 has changed from 1 to 0 when the listener(servlet) was expecting machine A to send information. It will check the new state which is 0 to the state of Device A's DIN 1 previous state, which is 0. Since they are the same, that value will not be written to the database. That will mean we have lost one state that was supposed to be written to the database from Device B's DIN1.

In our Singleton file we had:

```
int din1;
int din2;
int din3;
double a1;
public int getDin1() {
    return din1;
}

public void setDin1(int din1) {
    this.din1 = din1;
}

public int getDin2() {
    return din2;
}

public void setDin2(int din2) {
    this.din2 = din2;
}

public int getDin3() {
    return din3;
}

public void setDin3(int din3) {
    this.din3 = din3;
}
```

```
    }  
  
    public double getAi1() {  
        return ai1;  
    }  
  
    public void setAi1(double ai1) {  
        this.ai1 = ai1;  
    }  
}
```

Code Snippet 27. sigletone working for one DIN change only

As it can be seen there is no checking for whom these DINs and AIs belong to. That was then changed to the code snippet (see in Chapter 4.2.2) by creating a class called imeiInfo. It is populated by creating an array list of class imeiInfo, and saved the state on to that array list. That made it possible to save DINs and AIs separately by grouping them with their respective origin of devices which can be easily identified by a unique IMEI number.

5.3 Saving State on the Server

There was a problem in checking the last state (previous state) of the devices input by going though the database and checking the last state that was written there by that device with that particular input number. This took more processing time since it involved going back and forth between the servlet and the database, and there were some input values that were lost because of that. It did not help with the growth in the number of devices inside one factory floor either. The next code snippet shows how it was done

```

sqlSelect = "SELECT value FROM rtu_event where
rtu_imei = ? and rtu_input_id = ? ORDER BY id
DESC LIMIT 1";

ps = dbConn.prepareStatement(sqlSelect);
ps.setString(1, ssu.getImeiCode());
ps.setInt(2, dsu.getDinInput());

rs = ps.executeQuery();

if(rs.first())
lastDinState = rs.getInt("value");
System.out.println("Michael: Last State for
DIN1 was: " + lastDinState);

if(lastDinState != dsu.getDinState()){

// A similar code to save the digital
state goes here
}

```

Code Snippet 28. Check previous state from DB

This code combined with a simple getter and setter for the DINs and AI would only check for the last state of the DINs input inside the database.

It was proven to be inconsistent in saving all the states as intended, so there was a need to come up with a solution to save the DIN states inside the servlet, inside the Singleton class, and checking if the last state and current state are different before we even got to the Database class.

```

if(dinIOData1[1] !=
Singleton.INSTANCE.getImeiState().get(i).din1){

    DinSubUnit dsu1 = new DinSubUnit();
    dsu1.setDinInput(1);
    dsu1.setDinState(dinIOData1[1]);

    recordUnit.addSubUnit(RecordUnit.DIN_UNIT_KEY,
dsu1);

    System.out.println("Michael: previous DIN_UNIT
1: " +
Singleton.INSTANCE.getImeiState().get(i).din1)
;

    System.out.println("Michael: current DIN_UNIT
1: " + dinIOData1[1]);

    digitalinput1 = dinIOData1[1];

}

```

Code Snippet 29. Saving DIN state in the server memory

Now it can be seen that it checks the current, for example, DIN1 state with the state saved inside the Singleton class, if it does it will save the current state as the Singleton state (previous state) and send that data to be written to the database. The database class does not need to undergo any checking functionality but saves whatever is sent from here (RequestHandlerThread class).

That proved to be efficient in saving the states and proved to be consistent in saving all the state to the database.

6 CONCLUSION

The main idea of this project was to get the status information of a big machinery working on a factory floor. This information can tell a person if the machine is on or off, when it has been turned on or turned off, if it is working properly, if it has been broken and so on.

Based on these information a person can then make informed decision about breakdown preventive repair maintenance which could save a lot of money to the company while keeping customers happy.

With web based tools or mobile applications, it is now made possible to visualize the outcomes of the status sent from the device. This can show for example, which part of the machinery is not working, and what personnel to assign to fix it, how many hours it takes, what was the cause of failure for future reference, what spare part is used to fix and which storage room is it found, and how much was the total cost of these, and so on.

By using a device like FM1100, we can utilize all the three digital inputs. DIN1 is reserved to indicate if the machine is turned on or off. But we can do as we wish with the other two digital inputs. We can basically attach these digital inputs to the machine and configure it to give us the information we need based on the customers need and the machines capability.

FM1100 also makes it easier to communicate with a servlet with GPRS(TCP/IP and UDP/IP protocols) to send information and if that fails to deliver, we can use SMS as a backup delivery method.

This project took almost half a year to implement. It has been tested for quite some time by Tähtipiste Oy extensively, and is now fully functional and available for sale

on Maintbox's main page. The web service was implemented by a company called Mainscope which is also a share holder to Maintbox.

There has been also a requirement for a mobile application integration with the system. That has also been fully done on all mobile phone platforms and is being tested now. We used Cordova to implement the functionality.

The mobile application would not offer the entire functionalities as the web application would, but very important functionalities are integrated

REFERENCES

/1/ FM1100 User Manual

v1.32. Accessed 13.08.2013.

<http://www.teltonika.lt/uploads/docs/FM4100%20user%20manual%20V1.4.pdf>

/2/ Apache Ant Build . Accessed 13.08.2013.

<http://ant.apache.org/>

http://wiki.eclipse.org/FAQ_What_is_Ant%3F

/3/ Singleton usage. Accessed 17.02.2013.

<http://c2.com/cgi/wiki?JavaSingleton>.

/4/ FM1100 Configurator. Accessed 13.01.2013.

<http://www.gps-vehicle.com/downloads>

/5/ Computerized Maintenance Management Systems (CMMS). Accessed 10.01.2013

http://www1.eere.energy.gov/femp/pdfs/omguide_complete.pdf

/6/ Devatus Accessed 9.01.2013.

<http://devatus.fi/>

/7/ Apache Tomcat Server Accessed 19.05.2013.

<http://tomcat.apache.org/>