Nowadays, Wireless Sensor Networks are employed in many different fields. This project presents the implementation of this type of network on trains, with the purpose of monitoring the equipments condition at any time while trains are in motion. The Wireless Sensor Network is in charge of measuring and transmitting temperature and vibration parameters. The Gateway is the device that processes and transfers the gathered data to a remote server connected to Internet providing real-time measurements. This project contains the guidelines to configure and develop all applications related to the Gateway.

**Keywords**: Gateway, Wireless Sensor Network, Wi-Fi, GPRS, UMTS.
# Table of Contents

Table of Contents ................................................................. ii
Table of Figures ................................................................. v
List of Tables ................................................................. vii
Introduction ........................................................................... viii

Chapter I. Problem Statement ...................................................... 1
  I.1 Problem Statement .......................................................... 1
  I.2 General Aim ................................................................. 3
  I.3 Specific Aims ................................................................. 3

Chapter II. Review of the State of the Art ......................................... 5
  II.1 Gateway Definition ....................................................... 5
  II.2 Wireless Sensor Networks (WSN) ........................................ 5
    II.2.1 IEEE 802.15.4 ......................................................... 7
      II.2.1.1 Physical Layer (PHY) ........................................... 7
      II.2.1.2 MAC sublayer ................................................... 8
  II.3 IEEE 802.11 (Wi-Fi) ...................................................... 9
    II.3.1 IEEE 802.11 Physical Layer (PHY) ............................ 10
    II.3.2 IEEE 802.11 MAC Layer ......................................... 11
      II.3.2.1 IEEE 802.11 MAC Layer Functions ....................... 12
      II.3.2.2 Security ......................................................... 13
  II.4 Mobile Technologies ....................................................... 16
    II.4.1 General Packet Radio Service (GPRS) .......................... 17
    II.4.2 Universal Mobile Telecommunication System (UMTS) ....... 18
  II.5 Protocols ................................................................. 20
    II.5.1 Point to Point Protocol (PPP) ................................. 20
    II.5.2 File Transfer Protocol (FTP) ................................. 21
Chapter III. Methodology ................................................................. 24

III.1 Hardware Description ................................................................. 24

III.1.1 GW2358-4 Network Processor .................................................. 24

III.1.1.1 Processor ........................................................................... 25

III.1.1.2 SDRAM DDRII ................................................................. 25

III.1.1.3 Flash ................................................................................. 25

III.1.1.4 Compact Flash ................................................................ 25

III.1.1.5 Mini-PCI ........................................................................... 25

III.1.1.6 Ethernet ............................................................................ 26

III.1.1.7 Host USB .......................................................................... 26

III.1.1.8 Optional GPS ................................................................. 26

III.1.2 Copernicus® GPS Receiver .................................................... 26

III.1.3 Ovation MC950D USB Modem ............................................ 27

III.1.4 Atheros 4G Wireless Mini-PCI Adapter for IEEE 802.11 .......... 28

III.1.5 Crossbow’s TelosB Motes .................................................. 29

III.2 Network Architecture ................................................................. 30

III.3 Software Description ................................................................. 32

III.3.1 System Software ................................................................. 32

III.3.1.1 GNU/Linux .................................................................... 32

III.3.1.2 OpenWRT firmware ......................................................... 32

III.3.1.3 Drivers and Packages ....................................................... 33

III.3.1.4 Servers ........................................................................... 34

III.3.2 Application Software ................................................................. 34

III.3.2.1 Data Collection and Management .................................... 34

III.3.2.2 Alarms ............................................................................ 46
III.3.2.3 GPS Data Management ................................................................. 51
III.3.2.4 Data Treatment and Transmission .............................................. 52
III.3.2.5 Internet Access ............................................................................ 56
III.3.2.6 Web Application ........................................................................... 60
III.3.2.7 Vibration Files Transmission Protocol ......................................... 72

Chapter IV. Results .................................................................................. 74
IV.1 Temperature Packs Received ............................................................... 76
IV.2 Reply Ratio ........................................................................................ 78
IV.3 Average Reply Times ......................................................................... 80
IV.4 File Reception Ratio ........................................................................... 82
IV.5 Average Time of Vibration Files Transmission .................................... 83
IV.6 Average Vibration File Transmission Rate .......................................... 85
IV.7 Average Transmitted Packets for Correct received packets ............... 87

Chapter V. Conclusions and Recommendations ........................................ 89
V.I Conclusions ....................................................................................... 89
V.II Recommendations ............................................................................ 90

References ................................................................................................. 91

Acronyms .................................................................................................. 93

Annexes ..................................................................................................... 95
## Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Block Diagram</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2</td>
<td>System Layout</td>
<td>2</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Typical sensor network arrangement.(4 p. 16)</td>
<td>6</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Communication between TE and MT.(8)</td>
<td>20</td>
</tr>
<tr>
<td>Figure 5</td>
<td>GW2358-4(11)</td>
<td>24</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Copernicus GPS Receiver.(12)</td>
<td>27</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Compact Magnetic-Mount Antenna.(12)</td>
<td>27</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Ovation MC950D USB Modem.(13)</td>
<td>28</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Atheros 5354MP PLUS ARIES2 wireless Mini-PCI adapter.(14)</td>
<td>29</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Crossbow's TelosB Mote.(15)</td>
<td>29</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Network Architecture</td>
<td>31</td>
</tr>
<tr>
<td>Figure 12</td>
<td>MAC frame format</td>
<td>36</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Serial Frame format</td>
<td>37</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Temperature Frame Format</td>
<td>38</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Control Frame Format</td>
<td>39</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Vibration Frame Format</td>
<td>42</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Web Index</td>
<td>64</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Parameters Configuration Part I</td>
<td>65</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Parameters Configuration Part II</td>
<td>65</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Sensor Nodes Status</td>
<td>66</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Wi-Fi Configuration. List of Accepted Networks</td>
<td>66</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Wi-Fi Configuration. Add New Network Parameters Part I</td>
<td>67</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Wi-Fi Configuration. Add New Network Parameters Part II</td>
<td>67</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Wi-Fi Configuration. Modify Network Parameters Part I</td>
<td>67</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Wi-Fi Configuration. Modify Network Parameters Part II</td>
<td>68</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Wi-Fi Configuration. Delete Networks</td>
<td>69</td>
</tr>
<tr>
<td>Figure 27</td>
<td>3G/UMTS Configuration Part I</td>
<td>69</td>
</tr>
<tr>
<td>Figure 28</td>
<td>3G/UMTS Configuration Part II</td>
<td>70</td>
</tr>
</tbody>
</table>
Figure 29. History of Sent Files ................................................................. 70
Figure 30. Run Commands ........................................................................ 71
Figure 31. Results Showed After Executing a Command .......................... 71
Figure 32. Network deployed for tests ....................................................... 75
Figure 33. Test 1 Temperature Frames Received (W=6) ......................... 77
Figure 34. Test 2 Temperature Frames Received (W=10) ....................... 77
Figure 35. Test 3 Temperature Frames Received (W=14) ....................... 77
Figure 36. Test 1 Reply Ratio (W=6) ......................................................... 79
Figure 37. Test 2 Reply Ratio (W=10) ....................................................... 79
Figure 38. Test 3 Reply Ratio (W=14) ....................................................... 79
Figure 39. Test 1 Average Reply Times (W=6) ......................................... 80
Figure 40. Test 2 Average Reply Times (W=10) ....................................... 81
Figure 41. Test 3 Average Reply Times (W=14) ....................................... 81
Figure 42. Test 1 File Reception Ratio (W=6) ........................................... 82
Figure 43. Test 2 File Reception Ratio (W=10) ......................................... 82
Figure 44. Test 3 Files Reception Ratio (W=14) ....................................... 83
Figure 45. Test 1 Average Time of Vibration Files Transmission (W=6) .... 84
Figure 46. Test 2 Average Time of Vibration Files Transmission (W=10) .. 84
Figure 47. Test 3 Average Time of Vibration Files Transmission (W=14) ... 84
Figure 48. Test 1 Average Transmission Rate of Vibration Files (W=6) .... 86
Figure 49. Test 2 Average Transmission Rate of Vibration Files (W=10) .. 86
Figure 50. Test 3 Average Transmission Rate of Vibration Files (W=14) ... 86
Figure 51. Test 1 Average Transmitted Packet for Correct Received Packets (W=6) 87
Figure 52. Test 2 Average Transmitted Packet for Correct Received Packets (W=10) 87
Figure 53. Test 3 Average Transmitted Packet for Correct Received Packets (W=14) 88
List of Tables

Table 1. Average Time, Standard Deviation and Coefficient of Variation for processes of each test. .......................................................... 76
Introduction

The continuous research and development in Wireless Sensor Networks has led to an increase in the implementation of such networks under different scenarios.

Wireless Sensor Networks are composed of several sensor nodes for monitoring the environment where they are deployed. Usually, those nodes are in charge of the data collection, analysis and transmission over the network. The specific functions of sensor nodes are defined according to the applications that employ this type of networks. In applications that involve accessing the information collected by sensor nodes, it is required to connect the wireless sensor network with other types of networks such as Internet or intranets. The device capable of bridging different types of networks is a Gateway.

In this project it is presented the implementation of Wireless Sensor Networks on trains to measure different parameters related to the correct functioning of the rail network. The application is based on continuous monitoring of temperature and vibration measurements in the wheels of trains to avoid potential system failures and to know trains conditions in real-time by transmitting the collected data to a server connected to Internet while they are travelling. For this latter requirement, it is necessary to connect the itinerant wireless sensor network with Internet to transfer data to the server. Hence, it is required to employ a Gateway to control the sensor network.

The Gateway receives all data gathered and transmitted within the wireless sensor network, afterwards it processes the information in order to detect alarms in the system and finally it periodically transfers the processed data to the remote server. When an alarm is detected, the gateway immediately reports it to the server.

This project comprises all the steps for developing and configuring the Gateway that will operate according to the previous conditions. It is divided in five chapters. The Chapter I is composed of the problem statement which leads to the proposal of
general and specific aims. Chapter II presents the review of the state of the art, including the technologies employed to develop and achieve the stated aims.

Chapter III contains the methodology, which comprises the description of the hardware used, as well as the network architecture, the system software required and the application software developed.

Chapter IV contemplates the results of three tests executed with different parameters of transmission to check the performance of the network in each case.

Chapter V consists of the project conclusions and recommendations for future improvements in the system. Straight afterwards, the consulted references are presented followed by a list of the acronyms used and the Annexes containing the source code developed.
Chapter I. Problem Statement

I.1 Problem Statement

The equipment maintenance is a very important issue in transport companies and is directly related with the quality of service offered to the clients. A continuous inspection and monitoring of the equipment could detect a faulty element and inform that an action has to be taken to avoid the possible consequences.

The Condition-Based Maintenance (CBM) is intended to detect and diagnose out of order equipment. As the name indicates, it is based on the conditions that a certain object presents in a particular moment to determine if it needs maintenance or reparation. This type of maintenance permits the company to act when is actually necessary, reduces the response times when a fault occurs and shrinks corrective actions (1).

To implement the CBM is necessary to gather and process information about the equipment behavior in real time. The purpose of this project is based on applying this type of maintenance in the transport sector, specifically for railways equipments, using a Wireless Sensor Network (WSN), located along a train, controlled by a gateway that processes and sends information to a remote computer server with a database, in this way the maintenance team is kept abreast of what is happening on the train. Then, the implementation can be divided into three different blocks, the first one is referred to the WSN, the second block corresponds to the gateway and its interfaces and the last block is composed by the server which includes a database, an FTP server and a Web server. This distribution can be seen in Figure 1.
The WSN is responsible for the data acquisition, measuring temperature and vibration. The gateway gathers the measures from all sensors, detects alarms in the system, creates files and sends them to the FTP (File Transfer Protocol) server via 3G Mobile Networks, such as UMTS (Universal Mobile Telecommunications System) and GPRS (General Packet Radio Service), or Wi-Fi. The Server receives and saves the information in a database, which can be accessed through a web application. A graphic representation is depicted in figure 2.
This project presents the development and implementation of the Gateway, as the device that manages the sensor network, collects and processes the information looking for failures and finally transmits the data to a remote server to be analyzed by the maintenance team.

I.2 General Aim

Develop and configure a Gateway platform to control and connect an itinerant WSN with a central system via Internet using Wi-Fi and 3G mobile technologies, including UMTS and GPRS.

I.3 Specific Aims

There are several steps to develop and accomplish the main objective of this project. These steps are considered the specific aims and are described below.

- Manage and control the gateway board and its interfaces.
- Collect and store in files the data sent through the sensor network, including temperature and vibration measures.
- Acquire and save data from the integrated GPS (Global Positioning System) to know the exact position and speed of the train in any time.
- Periodically transmit the data and the GPS information to the remote server via GPRS, UMTS or Wi-Fi.
- Check the data collected to detect alarms.
- Trigger an alarm when the temperature reach certain threshold, the difference of temperature between wheels of the same axle is higher than a pre-established maximum value, the Discrete Fourier Transform (DFT) of the vibration data generates a tone which amplitude exceeds the limit.
- Send the data collected to the server when an alarm occurs with additional information about what caused the alarm.

- Implement a web application to configure the gateway and sensor parameters as well as the Wi-Fi and 3G connections.
Chapter II. Review of the State of the Art

II.1 Gateway Definition

A gateway is a device that interconnects two networks, and whose presence is usually visible to network users. It may be required to manage one or more differences between the networks it connects:

- Change of addressing domain, when the networks have addressing domains managed by separate groups a gateway may be used to handle address transformation.

- Control of charging, a gateway may be used to handle user authorization and usage accounting.

- Change of protocol, when the networks use different protocols a gateway could be used to carry out protocol conversion.

II.2 Wireless Sensor Networks (WSN)

A Wireless Sensor Network is an infrastructure formed by small devices using sensors to monitor physical or environmental conditions, such as temperature, sound or vibration. Each device is in charge of sensing (measuring), computing and transmitting its information. “Individually, each node is autonomous and has short range; collectively, they are cooperative and effective over a large area” (3 p. 7).

Therefore, a sensor node has four key components. A sensing unit composed by sensors and analog-to-digital converters, a power unit (usually batteries), a processing and storage unit and a transceiver unit for network connection.

The general functions of each sensor node are the data collection, analysis and transmission, normally using wireless links and multi-hop routing, to a sink point (destination device). A typical sensor network, with sensor nodes, clustering nodes
and the final destination node, is depicted in Figure 3 (4). The sensor nodes gather the sensor’s information, perform some processing and send the data to the clustering node; the clustering nodes receive, carry out further processing and forward the data to the next clustering node (multi-hop) or to the base station; the final processing node, also called base station, is in charge of receiving and processing the information sent through the sensor network; hence this latter node must have enough memory, power and computational capacity to correctly perform those functions. This node is usually responsible of connecting the sensor network with other networks, in order to inform an administrator about the data collected. The sensor nodes are organized in clusters to reduce the power consumption, because the energy used to transmit a message from a sensor to the clustering node is lower than the one that would be necessary to transmit the same message from the sensor nodes to the base station in a single-hop.

![Figure 3. Typical sensor network arrangement. (4 p. 16).](image)

The sensors network has some important limitations due to the sensors nodes’ small sizes. The principal constraints are related to power supply issues, computational and memory capacity and the environmental conditions. Batteries are usually the power supply used by the nodes; therefore the lifetime of the nodes directly depends on the duration of the batteries. The basic functionalities of the nodes, such as managing the
sensors, processing the data collected and transmitting the messages, require power supply. That is why each node normally has two operational modes, slept or idle if it is not needed and awake when it is required. The protocols used to guarantee the communication in the network must take into account these limitations to make the best use of the resources and assure the correct functioning of the network as long as possible.

II.2.1 IEEE 802.15.4

The IEEE 802.15.4 standard defines the physical layer (PHY) and the Medium Access Control (MAC) sublayer for low-rate Wireless Personal Area Networks (LR-WPAN). The Wireless Personal Area Networks are intended to transmit information, using wireless links, over short distances. The last version of the standard is the IEEE 802.15.4-2006 developed by the IEEE 802.15 group.

An IEEE 802.15.4 LR-WPAN could operate in a star or a peer-to-peer topology formed by full function devices and reduced function devices, with one of the full function devices acting as the PAN coordinator, which can start, finish or route communication through the network. All the devices have unique 64-bit addresses.

II.2.1.1 Physical Layer (PHY)

The PHY is in charge of the activation and deactivation of the radio transceiver, the energy detection (ED) within the current working channel, the link quality indicator (LQI) for received packets, the channel frequency selection and the data transmission and reception (5).

The standard defines four PHYs operating in three different frequency bands, the 868 MHz band (868 – 868.6 MHz) for Europe, the 915 MHz band (902 – 928 MHz) available in USA and the 2400 MHz band (2400 – 2483.5 MHz) available worldwide.
The 2400 MHz band provides data rates of up to 250 Kbps using Direct Sequence Spread Spectrum (DSSS) and Offset Quadrature Phase Shift Keying (O-QPSK) modulation.

There are three different modulation techniques specified for the 868/915 MHz bands. The first one employs DSSS and Binary Phase Shift Keying (BPSK) modulation, which permits data rates of up to 20 Kbps in the 868 MHz band and 40 Kbps when operating in the 915 MHz band. The second one is an optional PHY defined to use a multi-code modulation technique called Parallel Sequence Spread Spectrum (PSSS) or Orthogonal Code Division Multiplexing (OCDM) and Amplitude Shift Keying (ASK) modulation; the BPSK modulation is also supported. The data rate of this PHY is up to 250 Kbps for both, the 868 MHz and the 915 MHz bands. The third, also optional, PHY uses DSSS and O-QPSK modulation to achieve data rates of up to 100 Kbps in the 868 MHz band and 250 Kbps in the 915 MHz band (5).

The channel assignment is based in a combination of channel numbers and channel pages; there are 32 channel pages and 27 (0 - 26) channel numbers per channel page. The channel page 0 is defined for the 868/915 MHz bands using BPSK modulation and for the 2400 MHz band using O-QPSK modulation, within this channel page the channel numbers assigned are 0 for the 868 MHz band, from 1 to 10 for the 915 MHz band and 11 to 26 for the 2400 MHz band. From channel pages 1 and 2, 11 channels are used by the 868/915 MHz bands using ASK and O-QPSK modulation respectively, with channels numbered from 1 to 10 for the 915 MHz band and the channel 0 for the 868 MHz band. The rest of channel numbers and channel pages are reserved for future use (5).

**II.2.1.2 MAC sublayer**

The MAC sublayer is in charge of the access to the physical radio channel using CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) mechanism, the
beacon management, the frame validation, the nodes’ association and disassociation. It provides reliable links between two peer MAC entities (5).

The standard defines four types of frames: data, acknowledgement, beacon and MAC command. The acknowledgment frames are optionally sent to confirm the reception of a data frame or a MAC command frame. The MAC command frames are used to control the network, specifying with the command that an action must be performed within the devices that form the network. The data frames contain the information exchanged within the network. The beacon frames are used for synchronization, to identify the PAN, and to inform about the structure of the superframes. The superframes are delimited by network beacons sent by the coordinator and are divided into 16 slots. The time between the first and the last beacon is the Contention Access Period (CAP); the devices can transmit during this time using a slotted CSMA/CA mechanism. For applications that require specific data bandwidth, the coordinator can assign a period of time called Contention Free Period (CFP) containing Guaranteed Time Slots (GTS) at the end of the CAP (5).

There are two ways of channel access, with beacons and using superframes or without beacons and using unslotted CSMA/CA channel access mechanism.

**II.3 IEEE 802.11 (Wi-Fi)**

The IEEE 802.11 standard specifies the physical and MAC layers for Wireless Local Area Networks (WLAN). The release of the original standard was in 1999 and since then there have been many revisions and amendments to improve the communication within WLANs.

The basic components of the 802.11 networks are the stations, the access points, the wireless medium and the distribution system. The stations are devices with wireless network interfaces which can communicate with each other directly or through access points. The main function of the access points (APs) is bridging the wireless network with wired networks. A distribution system is used for the communication between
access points when these are connected to form a large coverage area. The basic building block of these networks is the Basic Service Set (BSS), which can be independent or with infrastructure. The first one is formed only by interconnected stations and the second one combines stations and access points.

II.3.1 IEEE 802.11 Physical Layer (PHY)

The original IEEE 802.11 standard defines two physical layers based on radio waves, which are the Frequency Hopping Spread Spectrum (FHSS) and the Direct Sequence Spread Spectrum (DSSS) allowing data rates of 1 Mbps and 2 Mbps while working in the 2.4 GHz band. The following published standards specified three more physical layers.

The IEEE 802.11a standard introduces the Orthogonal Frequency Division Multiplexing (OFDM); it operates in the 5 GHz band with a maximum theoretical data rate of 54 Mbps, which in practice is around 20 Mbps.

The IEEE 802.11b standard presents the High Rate Direct Sequence Spread Spectrum (HR/DSSS) using DSSS with several modulation schemes such as Complimentary Code Keying (CCK) and Packet Binary Convolutional Coding (PBCC); it provides a maximum theoretical data rate of 11 Mbps which results in real data rates between 5 and 7 Mbps working in the 2.4 GHz band.

The IEEE 802.11g standard defines an Extended Rate PHY (ERP) employing DSSS, FHSS and OFDM; it operates in the 2.4 GHz band and offers a maximum raw data rate of 54 Mbps with an average throughput of 22 Mbps. The 802.11g hardware is compatible with the 802.11b hardware.

The 2.4 GHz band and the 5 GHz band are divided into channels. The availability of channels is regulated by each country and is limited by the manner in which radio spectrum is allocated for miscellaneous services.
II.3.2 IEEE 802.11 MAC Layer

The functions of the IEEE 802.11 MAC layer are intended to manage and coordinate access to the transmission channel and are responsible for authentication and other security and management tasks.

The mechanism to access the medium is Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) along with Request to Send (RTS) and Clear to Send (CTS) signals to avoid collisions and to solve the problem of the hidden node.

The access to the wireless medium is controlled by coordination functions. The Distributed Coordination Function (DCF) manages the access during the contention period; the Point Coordination Function (PCF) provides contention-free services, it is implemented by the access points to organize the transmissions using polling mechanisms, the Hybrid Coordination Function (HCF) is a combination of the DCF and PCF intended to provide better service quality coordinating the access depending on the applications’ priorities (6).

The basis of the standard is the DCF. When a station wants to transmit, it first listens to the medium (carrier sense), if the medium is busy (other station is transmitting) it waits a random time (backoff timer) to try to transmit again. The station will transmit only when the medium is idle to avoid collisions. When the station finally transmits, the receiving station must confirm the reception by sending an ACK packet to the sender. If the sender does not receive an ACK packet after transmitting, it will retransmit the information until the confirmation is received or it will give up after a given number of retransmissions. The transmission is considered successful when the sender receives the confirmation packet (ACK).

There are two types of carrier-sensing functions, the physical and the virtual. The physical carrier-sensing functions are provided by the physical layer and depend on the medium and the modulation used. The virtual carrier-sensing is implemented using the Network Allocation Vector (NAV), which represents the time that the
stations need to use the medium. When a station desires to transmit and the medium is idle, it sends the time required for completing the transaction in a duration field sent within the frame; the rest of stations set their NAV timer and do not attempt to use the medium until the timer expires.

### II.3.2.1 IEEE 802.11 MAC Layer Functions

The principal MAC functions specified in the standard are scanning, authentication, association, power saving, fragmentation and security.

The standard defines the mechanisms for scanning the medium, which can be passive or active. The passive scanning consists in checking all channels to find the best AP signal; the APs periodically broadcast beacon frames with information about the BSS parameters. If the active scanning is implemented, the station broadcasts a probe frame and all the APs within the range respond with a probe response frame; with this mechanism is not necessary to wait for the beacon frames to find the best AP, however it introduces overhead on the network.

The authentication process starts when the station finds the desired AP. It is the process of providing identity. There are two forms of authentication, open system and shared key. The open system authentication is a two steps process; the station sends an authentication request frame to the AP and the AP replies sending an authentication response frame. The shared key authentication is a four steps process that uses Wired Equivalent Privacy (WEP); the station sends the authentication request frame to the AP, the AP replies adding a 128 bits challenge text in the response frame, then the station must encrypt the text with its WEP key and send a third frame with the encrypted text, finally the AP decrypts the challenge text and compares it with the original one, if they are equivalent the AP assumes that the sender has the correct key and replies with an authentication frame approving the sender’s identity.
The association with an AP must be performed after authentication and before sending data frames. The association permits the exchange of basic information between the station and the AP. The station sends an association request frame to the AP and the AP replies with an association response frame containing an association ID to identify the station for further data transmission.

The stations in a wireless network are normally mobile devices powered by batteries; the 802.11 standard defines a mechanism for power conservation, an optional power save mode that could be on or off depending on data transmission requirements. The stations inform their status to the AP in a bit within the frame header. The AP buffers the data corresponding to the sleeping stations and starts sending regular beacon frames to notify about the buffered packets to the stations. When the stations wake up and receive one of those beacon frames they should request the AP to send their corresponding data.

Another important function of the MAC layer is the fragmentation of data packets to reduce the effects of the interference in the medium, the smallest the packet the less overhead it causes to retransmit it in case of packet losses.

The information transmitted in wireless networks is susceptible to attacks and interceptions; therefore it is essential to implement security in this type of networks. The mechanisms and algorithms used for security are described in the following section.

**II.3.2.2 Security**

The first security mechanism standardized for WLANs was the Wired Equivalent Privacy (WEP). WEP uses a RC4 cipher algorithm that implements a key-stream (stream of bits) combined with the message using the exclusive OR (XOR) operation to create a cipher-text that is going to be sent instead of the plain message; the receiver has the same key-stream to recover the original message. WEP only protects the 802.11 MAC payloads; it calculates an integrity check value (ICV) based on the
information contained in the frames payloads. The WEP keys are formed concatenating the secret key and the 24-bits initialization vector (IV) to produce different stream ciphers for each frame; those keys are used by the RC4 algorithm to encrypt the MAC frames’ payload. The final frames are formed by the MAC header, the WEP header with the IV and the key number used for the encryption (WEP allows four keys), the encrypted data and the ICV. The length of the WEP keys can be 40 or 104 bits. There are two types of key, the key-mapping key shared between the station and the AP to protect unicast frames and the default key shared by all stations to protect multicast and broadcast frames (6).

In addition to the 802.11 authentication types described in the previous section, the IEEE 802.1X standard can be used for authentication using the Extensible Authentication Protocol (EAP) based in users’ authentication rather than machines’ authentication. The 802.1X authentication is initiated after the station has been associated with the 802.11 network and consists in the exchange of frames between the station (supplicant), the AP (authenticator) and a back-end authentication server like RADIUS (Remote Authentication Dial-In User Server); the authenticator sends an EAP-Request/Identity frame to the supplicant, the supplicant replies with an EAP-Response/Identity frame that is forwarded by the authenticator to the RADIUS server which sends an EAP-Request frame asking for the EAP method to authenticate the corresponding user’s identity, the supplicant responds with the method used and the user’s information required for the authentication. The authenticator transmits an EAP-Success frame to the supplicant when the RADIUS server accepts the user’s authentication. The AP sends the keys to the supplicant which can initiate the transmission of data frames after processing those keys. The supplicant transmits a logoff message when it desires to finish the connection (6).

The WEP encryption presented many weaknesses and it could be easily cracked. Therefore, the IEEE 802.11i standard was released to enhance the security in wireless networks. The 802.11i standard comprises the 802.1X for authentication and two new link-layer encryption protocols, the Temporal Key Integrity Protocol (TKIP)
endorsed under the name Wi-Fi Protected Access (WPA) and the Counter Mode with Cipher Block Chaining Message Authentication Code Protocol (CCMP). The 802.11i standard is known as WPA2 or RSN (Robust Security Network).

TKIP was designed to be a WEP upgrade and was introduced by the Wi-Fi Alliance before the ratification of the IEEE 802.11i standard; it implements a key mixing function combining master keys to create a unique encryption key for each frame and also uses a 48-bits IV, the master keys are refreshed using key management operations; TKIP uses RC4 cipher algorithm to make use of the WEP hardware. It employs a sequence counter to avoid replay attacks, so frames received out of order are discarded. In comparison with WEP, TKIP implements a more robust message integrity check (MIC) algorithm called Michael but it can also be attacked thus TKIP uses countermeasures to control the harm that an active attack could cause to the network.

TKIP is better than WEP, but remains vulnerable to attacks by using a stream cipher. The Advanced Encryption Standard (AES) is a cipher that works with many key and block sizes, the 802.11i standard specifies the use of 128-bits keys and blocks. The Counter Mode with CBC-MAC Protocol (CCMP) defined in the 802.11i standard is based on AES for encryption. Like WEP and TKIP, CCMP only protects the 802.11 MAC payloads and uses temporal keys with a key identifier to know which key is going to be used for encryption and recovery of the data; it assigns a 48-bit packet number to each frame, which is incremented by one with every transmitted frame and it is used to avoid replay attacks because the numbers are never re-used for the same temporal key. CCMP creates an Additional Authentication Data (AAD) field which contains the fields in the frame header used for authentication that are transported without encryption; the receiver uses the AAD field to check that the authenticated fields, such as the address fields, were not corrupted during the transmission. CCMP constructs a nonce (string of bits) using the packet’s number and the sender’s address. The nonce, the AAD and the 128-bit temporal key are employed to encrypt the frames’ payload and the 8-bytes MIC using CCM. The final encrypted frame contains
the MAC header, the CCMP header that includes the packet’s number and the key identifier, the MAC payload with its MIC both encrypted and the FCS (Frame Check Sequence).

The IEEE 802.11i standard also defines the types of keys needed to implement the security, which are the pairwise keys that protect the transmissions between a station and the AP and the group keys for broadcast or multicast traffic. TKIP and CCMP take a single master key and turn it into the different keys used to protect the frames. The pairwise master key (PMK) is formed by 256 bits and could be a Pre-Shared Key (PSK) or could be generated by an authentication server like RADIUS. The group master key (GMK) is maintained by the authenticator (AP) and has a 128-bit length.

II.4 Mobile Technologies

The evolution of mobile telecommunications is divided into generations. The primary service of the first and the second generations of cellular mobile systems, 1G and 2G respectively, is voice communication using circuit-switched data. The main difference between both systems is that 1G systems transport analog signals while 2G systems use digital signals. Therefore, 2G systems were developed to enhance the voice traffic throughput; 2G introduced data services with the Short Message Service (SMS). The third generation (3G) provides high-speed data and voice services offering multimedia capabilities in the mobile wireless systems using packet data. There is a transitional generation between 2G and 3G and is called 2.5G; 2.5G introduces digital packet services combining circuit-switched and packet data. The next expected evolution is the fourth generation (4G) which is an enhancement of 3G technologies combined with WiMAX technologies to provide high data rates. The technologies used to develop this project were GPRS and UMTS, belonging to 2.5G and 3G respectively, which are explained below.
II.4.1 General Packet Radio Service (GPRS)

GPRS is a packet-switching technology which provides high data rates for data services. In real networks, it provides packet speeds of up to 40 or 53 Kbps depending on the number of time slots assigned for data transmissions. The RF resources are only used when sending or receiving data and are shared between multiple users. GPRS employs TDMA (Time Division Multiple Access) with FDD (Frequency Division Duplex) to make use of the frequency band. FDD means that the downlink is established in different frequencies than the uplink, for example, if working in the 900 MHz band, frequencies from 890 to 915 MHz are used for the uplink (transmissions from mobile station to the network) and from 935 to 960 MHz for the downlink (transmissions from the network to the mobile station). Each of the downlink and the uplink bands is separated into channels or carriers of 200 KHz and each carrier is divided into eight time slots transmitted in a frame structure that lasts around 4.62 ms, each time slot can carry user traffic or control signaling (7).

GPRS is a packet data network overlaid on the 2G GSM circuit-switched network. A GPRS network is formed by the Base Station Controller (BSC), the Packet Control Unit (PCU), the Serving GPRS Support Node (SGSN), the Gateway GPRS Support Node (GGSN) and the Charging Gateway Function (CGF). The PCU is in charge of logical functions such as the air-interface access control, packet scheduling and packet assembly and reassembly; it is integrated with the BSC. The SGSN is responsible for packet routing and transfer, mobility management, security and authentication and charging functions. The GGSN is the interface with external packet-data networks, like the Internet. A SGSN may serve multiple BSCs; the interface between the SGCN and the BSC is based in Frame Relay and uses the Base Station System GPRS Protocol (BSSGP) and is intended to exchange signaling and control information aside from user data traffic; both elements, BSC and SGSN, have relay functions to transmit the packets between different protocol stacks (7).
There is another interface between the SGSN and the GGSN; it is based in Internet Protocol (IP) and is used to carry signaling and user data. It uses the GPRS Tunneling Protocol (GTP) which tunnels user data through the IP backbone network and allows mobility in the network while providing a continuous connection.

The Mobile Station (MS) protocol stack comprises the RF interface as the first layer; above this are the Radio Link Control (RLC) and the Medium Access Control (MAC) functions. The next layer is the Logical Link Control (LLC) which provides a framing structure to communicate with the SGSN. The Sub-Network Dependent Convergence Protocol (SNDCP) is between the LLC and the network layer and provides support for many network protocols maintaining the lower layers unchanged. The MS may be divided into the Terminal Equipment (TE) and the Mobile Termination (MT), the first is the computing part of the MS and the latter supports the wireless access capabilities.

### II.4.2 Universal Mobile Telecommunication System (UMTS)

UMTS is a third generation technology that provides high speed data transmissions over mobile wireless networks. It uses Direct Sequence Wideband Code Division Multiple Access (DS-WCDMA) with both, Frequency Division Duplex (FDD) and Time Division Duplex (TDD), being FDD the most used solution. It provides theoretical data rates of up to 384 Kbps in the downlink and up to 64 Kbps in the uplink. Higher data rates may be reached using High Speed Downlink Packet Data (HSDPA) and High Speed Uplink Packet Data (HSUPA), which are modulation schemes used to improve WCDMA throughput to a theoretical 14.4 Mbps in the downlink and up to 7.2 Mbps in the uplink.

UMTS can be split into domains, the user equipment (UE) domain and the infrastructure domain; the latter is formed by the access network and the core network. The UMTS access network is the UMTS Terrestrial Radio Access Network (UTRAN) and the core network is based on the core network used for GSM/GPRS.
There are two types of nodes in the UTRAN, the Radio Network Controller (RNC) and Node B (base station). All the interfaces in the UTRAN and between UTRAN and the core network use Asynchronous Transfer Mode (ATM) for transport.

The UE is divided into the Mobile Equipment (ME) domain and the User Services Identity Module (USIM) domain. The ME may be sub-divided, as the MS in GPRS, in the Mobile Termination (MT) and the Terminal Equipment (TE). The TE and MT may be physically separated but can be connected by multiple technologies (serial, Bluetooth, infrared). The more common link layer protocol used at the TE to transport IP frames is the Point to Point Protocol (PPP) because it is supported by many operating systems. The TE uses the Hayes AT command set to control the MT.

The TE issues AT commands to set up parameters and enter PPP mode, it waits for the MT responses. The PPP protocol in the TE sends a LCP Configure-Request, which is used to establish a PPP link between the TE and the MT. The MT replies with a LCP Configure-Ack to confirm the establishment of the PPP link and sends a LCP Configure-Request to negotiate the authentication protocol used for authentication of the host TE; the MT shall initially negotiate for CHAP, and if this is unsuccessful, for PAP. The TE returns a LCP Configure-Ack to the MT to confirm the use of the specified authentication protocol and authenticates itself using the chosen protocol. The MT stores the necessary authentication data and sends an ACK of the authentication to the TE. If none of the protocols is supported by the host TE no authentication shall be performed. The PPP protocol in the TE sends to the MT a NCP Configure-Request to activate the IP protocol. If the MS is not yet attached to the Packet Switched (PS) network, the MT performs the PS Attach procedure. The MT sends a NCP Configure-Ack command, which is used to confirm that the IP protocol is activated and also may carry IP protocol related parameters such as dynamic IP address. The MT shall also pass name server information to the TE if the TE has requested for it and if this information is provided by the GGSN (8). This process is depicted in Figure 4.
II.5 Protocols

Protocols implemented are briefly described in this section.

II.5.1 Point to Point Protocol (PPP)

Point to Point Protocol is used to establish the communication with the MT. PPP is a data link protocol that provides multi-protocol encapsulation, a Link Control Protocol (LCP) to activate, test, configure and deactivate data links and a Network Control Protocol (NCP) for each network protocol supported. To establish a point-to-point communication, there must be an exchange of LCP packets between peers (each end of the link) to establish the link, after this it is required to choose and configure the network protocol via NCP packets. The link remains established until LCP or NCP packets specify that must be terminated.

There are three types of LCP packets:
− Link configuration packets: Configure-Request, Configure-Ack, Configure-Nak and Configure-Reject.
− Link termination packets: Terminate-Request, Terminate-Ack. They are used to close the link connection.
− Link maintenance packets: Code-Reject, Protocol-Reject, Echo-Request, Echo-Reply and Discard-Request. They are used to manage and debug the link.

When opening a connection, a Configure-Request must be sent specifying the configuring options such as maximum receive unit (MRU), authentication protocol, quality protocol, magic number, protocol field compression and address and control field compression. If all configuration requests are accepted, a Configure-Ack is sent, otherwise, a Configure-Nak with the options that were not accepted, and the negotiation continues. A Configure-Reject is sent when a configuration option is not recognized (9).

Once the link is established, each network protocol must be configured with its corresponding NCP. When using TCP/IP protocol, the NCP packets carry the IP address requests and replies. When the network layer configuration finishes, PPP will transfer the corresponding network protocol packets.

**II.5.2 File Transfer Protocol (FTP)**

File Transfer Protocol is employed to transmit files to the remote server. It is an application layer protocol intended to share and manipulate files over TCP/IP networks. It is based on client-server architecture. Control and data connections are separated. The FTP server listens on the control port, usually port 21, for users’ control connection. The FTP commands, which define the functions to be performed, are generated by the user and are transmitted to the server through the control connection that is also used to transfer server’s responses; those commands specify the parameters for the data connection such as data port, transfer mode, representation
type and structure and also define file operations. There are two modes for defining the data port, the active mode in which the FTP client opens and starts listening on a port and sends the port number to the FTP server, and the passive mode in which the FTP client requests the port number opened at the FTP server to establish the data connection.

Users may specify the type of data transferred. Those types are: ASCII (default type), EBCDIC (Extended Binary Coded Decimal Interchange Code), Image type and Local type. ASCII and EBCDIC may have an additional option to indicate the format depending on the operation that should be performed over that type of data like printing, storage and later retrieval or processing; there are three formats: non print (default), telnet format controls or carriage control. The structure of the file may also be specified which could be: file-structure that is a continuous sequence of bytes (default), record-structure that is made up of sequential records and page-structure that comprises independent indexed pages. (10).

There are three transmission modes: stream mode in which the data is transmitted as a stream of 8-bit bytes, block mode for transmitting data in series of blocks using header bytes for block’s specification and the compressed mode.

The FTP commands used to transfer files to the remote server are:

- **USER**: the argument is a string with user’s identification. It is the first command transmitted after the control connection is established. The user must be recognized by the server to access its file system.
- **PASS**: this command must follow the USER command and its argument is the user’s password.
- **CWD**: change working directory. It allows users to specify the directory where they want to store or retrieve files. The argument is the directory’s pathname.
- **PASV**: enter in passive mode. The FTP responds with the data port number.
− **STOR**: starts a data transfer from FTP client to server and stores the transferred data in a file at the server. The file name is send as the argument. If the file specified already exists at the server, it is overwritten, otherwise the file is created.

− **QUIT**: it is used for log-out and to close the control connection.

The FTP servers’ replies are 3 digit codes followed by one line of text describing the response and a code for end-of-line. There are five possible values for the first digit and six for the second digit of the reply code; the third digit provides a finer specification depending on the second digits. The code format is:

− 1yz: positive preliminary reply. The action is being initiated but other reply must be sent to confirm its correct execution.

− 2yz: positive completion reply. The action was successfully performed.

− 3yz: positive intermediate reply. It is used in command sequence groups to accept each command before execution of the sequence.

− 4yz: transient negative completion reply. The command was not accepted, but the error is temporary and the command may be sent again.

− 5yz: permanent negative completion reply. The command was not accepted and the user must not request this action again.

− x0z: syntax error.

− x1z: reply to a request for information like status or help.

− x2z: replies related to the control and data connections.

− x3z: replies to authentication and accounting.

− x4z: unspecified.

− x5z: replies to indicate the status of the server file system when a file transfer or other file system action is requested.(10).
Chapter III. Methodology

III.1 Hardware Description

The hardware used to achieve the goals of this project is described in this section.

III.1.1 GW2358-4 Network Processor

The GW2358-4, shown in Figure 5, was the network processor board used to develop the gateway platform. It is produced by Gateworks Corporation and it’s a member of the Cambria Network Processor family. This network processor consists of an Intel® IXP435 XScale® operating at 667MHz, 128Mbytes of DDRII-400 SDRAM, and 32Mbytes of Flash. Peripherals include four Type III Mini-PCI sockets, two 10/100 Base-TX Ethernet ports, two USB Host ports, and Compact Flash socket. Additional features include digital I/O, serial EEPROM with 8 Kbits of storage, real time clock with battery backup, system monitor to track operating temperature and input voltage, RS232 serial port for management and debug. The GW2358 also supports GPS and RS485 serial port. Power is applied through a dedicated connector. The board is able to support operating temperatures from -40° C to +85° C; the input voltage is in the range of 8 to 48 V DC with a typical operating power of 6 W. Its dimensions are 101.6 mm length x 152.4 mm width. This network processor has software support for Linux operating system (11).

Figure 5. GW2358-4(11).
III.1.1 Processor

The Intel® IXP435 XScale® network processor presents the following features: 32-bit SDRAM DDRII interface operating at 400 MHz, 32-bit PCI interface operating at 33 MHz for Mini-PCI support, 16-bit Expansion interface for Flash and peripheral support, two 802.3 MII/RMII interfaces for Ethernet PHY support, two USB v2.0 host controllers support low, full, and high speed operation (11).

III.1.1.2 SDRAM DDRII

The DRAM (Dynamic Random Access Memory) resides in two synchronous DDRII-400 devices. The 32-bit DDRII interface operates at 400 MHz with a memory capacity of 128 Mbytes (11).

III.1.1.3 Flash

It resides in a single Intel StrataFlash® device. The Flash memory capacity is 32 Mbytes (11).

III.1.1.4 Compact Flash

A Compact Flash socket located on the bottom side of the board extends the local Flash storage capacity. This socket accepts small removable mass storage cards with storage capacities up to 64 Gbytes (11).

III.1.1.5 Mini-PCI

It is a small form factor PCI (Peripheral Component Interconnect) card. The GW2358-4 has four Mini-PCI sockets, two located on the top side of the board and the other two on the bottom. A total operating power of 20 W is distributed between the four sockets, providing each socket with an average of 5 W for supporting high power radios. The Mini-PCI sockets also support a low-ripple 5V supply capable of providing up to 8W of power to the Mini-PCI sockets, which makes them ideal for present and future generations of WiMAX cards (11).
III.1.6 Ethernet

The Ethernet ports operate in a 100BASE-TX or 10BASE-T, they include full-duplex operation for both 10 Mbps and 100 Mbps configurations as well as support for auto-negotiation. The Ethernet ports are available through standard RJ45 connectors (11).

III.1.7 Host USB

The GW2358-4 supports two v2.0 host USB ports directly from the IXP processor. The USB ports are Enhanced Host Controller Interface (EHCI). The ports support all three standard data transfer rates of low speed (1.5 Mbps), full speed (12 Mbps), and high speed (480 Mbps). The power switch for both ports includes over current protection, thermal protection, in-rush current limiting, and hot-plug noise filters (11).

III.1.8 Optional GPS

The GW2358-4 implements the Trimble Copernicus module as an optional feature, which is described in the following section (11).

III.1.2 Copernicus® GPS Receiver

One scope of this project was to know the position and speed of the trains in a particular moment, in this way it was necessary the integration of a GPS receiver in the board previously described. The device used was the Copernicus® GPS Receiver fabricated by Trimble. The Copernicus module is a complete 12-channel GPS receiver in a 19 mm × 19 mm × 2.54 mm thumbnail-sized module, as can be seen in Figure 6, its prime power ranges from +2.7 to +3.3 V DC, supports operating temperatures from -40 to +85º C and it works at a 1 Hz update rate (12).

The Copernicus GPS receiver can autonomously acquire GPS satellite signals and quickly generate reliable position fixes in extremely challenging environments and under poor signal conditions. The module consumes less than 94 mW typically at full power with continuous tracking (12).
The Copernicus GPS module is a receiver that provides position, speed, and time data in a user’s choice of three protocols. Trimble’s powerful TSIP (Trimble Standard Interface Protocol) protocol offers complete control over receiver operation and provides detailed satellite information. The TAIP (Trimble ASCII Interface Protocol) protocol is an easy-to-use ASCII protocol designed specifically for track and trace applications. The bi-directional NMEA 0183 v3.0 protocol offers industry-standard data messages and a command set for easy interface to mapping software (12).

The Copernicus GPS receiver is used in combination with a 3V active micropatch antenna with magnetic mount and dimensions 30.4 mm width x 35.5 mm length x 11.7 mm height. This antenna is shown in Figure 7.

III.1.3 Ovation MC950D USB Modem

The gateway should be connected to the Internet at anytime to send the data collected from the WSN to the remote server. The measures are taken within moving trains, thus it was necessary to implement mobile technologies to access the Internet. The device used to provide mobile broadband Internet access was the Ovation MC950D USB modem.
The Ovation MC950D USB modem, shown in Figure 8, is an HSPA Mobile Broadband device using the Qualcomm MSM7200 chipset. This tri-band UMTS (850/1900/2100 MHz) and quad-band EDGE/GPRS (850/900/1800/1900 MHz) modem supports download rates of 7.2 Mbps and upload speeds of up to 2.1 Mbps with an integrated 5 bands antenna and Equalizer support. Its dimensions are 70 x 25 x 12 mm, it has a SIM/USIM card slot for 25 mm x 15 mm card sizes (13).

III.1.4 Atheros 4G Wireless Mini-PCI Adapter for IEEE 802.11

Another gateway’s feature is the detection of specific wireless networks. When within range of one of them, the gateway should connect to the Internet through that network and continue the transmissions to the remote server. Therefore, a wireless adapter was needed.

The Atheros 4G wireless Mini-PCI adapter is a Type III Mini-PCI dual band wireless adapter for IEEE 802.11a/b/g networks. It operates in the 2.4 GHz band for IEEE 802.11b/g and in the 5 GHz band for IEEE 802.11a using the Dynamic Frequency Selection (DFS) process and the Transmission Power Control (TPC) mechanism to find the best channel. The latter mechanism is used to control the output power to avoid interferences. The data rates supported are the ones defined in the three standards, up to 11 Mbps for IEEE 802.11b and up to 54 Mbps for IEEE 802.11a/g theoretically. The security mechanisms implemented are 64/128/152-bit WEP, WPA, TKIP and AES. The wireless card supports multi-country roaming, automatically adjusts regulatory domain to operate in different countries (14).

The operating voltage is 3.3 V and the operating temperature ranges from 0º C to 55º C. The wireless Mini-PCI adapter has two U.FL connectors (miniature coaxial RF
connectors) to plug in the Wi-Fi antenna. The card’s dimensions are 50.9 mm length x 59.6 mm width x 4.8 mm height. It is shown in Figure 9. (14)

![Figure 9. Atheros 5354MP PLUS ARIES2 wireless Mini-PCI adapter.](image)

### III.1.5 Crossbow’s TelosB Motes

The Wireless Sensor Network was implemented using TelosB motes. The gateway needed an interface to communicate with the WSN, for that reason a TelosB mote must be connected to one of the board’s USB ports.

Crossbow’s TelosB mote, shown in Figure 10, is an open source platform designed to enable experimentation for the research community. It includes USB programming capability, an IEEE 802.15.4 radio with integrated antenna, a globally compatible ISM band with a frequency range of 2.4 to 2.4835 GHz, a 250 Kbps data rate, a 8 MHz TI MSP430 microcontroller with 10 Kbyte RAM, a low current consumption (1.8 mA in active mode and 5.1 µA in sleep mode) and 1 MByte external flash for data logging. It runs TinyOS 1.1.10 or higher. Its size is 65 mm length x 31 mm width x 6 mm height excluding the battery pack. (15)

![Figure 10. Crossbow's TelosB Mote.](image)

The TelosB platform was developed and published to the research community by the University of California, Berkeley. It is powered by two AA batteries with long
lifetime due to its low power consumption. If it is plugged into the USB port, the power is provided by the host computer.(15).

III.2 Network Architecture

This project is intended to be implemented in trains. A train is composed by series of vehicles connected and attached to bogies. The bogie is a complex structure or chassis comprising two axles and four wheels (two wheels per axle); it also carries suspension and brake systems.

There are three types of nodes in the network:

- **Sensor node**: it is a battery powered TelosB mote connected to a temperature sensor and an accelerometer sensor; the latter is used to measure shock and vibration along three axes or dimensions (x, y, z) generating an analog signal which is sampled by the node at a specific frequency. Sensor nodes are organized in groups or clusters of four nodes.

- **Relay node**: it is the TelosB mote that manages the clusters. It is in charge of searching the best route to transmit the sensor nodes’ information through the network to reach the gateway node.

- **Gateway node**: it is a TelosB mote connected to the network processor board. It is the interface between the WSN and the board. This node receives the WSN traffic, changes the frames’ format and transmits it to the gateway through the USB port; it also makes the inverse process when transmitting frames from the board to the WSN.

The network processor board (GW2358-4), from now on the gateway, is the last component of the network. It receives frames from the gateway node. The gateway is responsible for processing and storing the data sensed, identifying alarms in the system and transmitting the information to a remote server.

Hence, the network deployed along the train is formed by one sensor node per wheel which means four sensor nodes per bogie, one relay node per bogie, one gateway
node and the gateway; this is illustrated in Figure 11. The sensor nodes are located at the wheels while the other components are inside the train, the number of sensor nodes will depend on the number of wheels. Relay nodes are responsible of multihop routing, thus the amount of relay nodes inside the train depends on the coverage of their antennas to reach other relay nodes. In the first place, it would be one relay node per bogie, but this will depend on the real scenario inside the train. The WSN operates in the 2.4 GHz band.

The network may also be divided having one gateway to control one half of the train and a second gateway for the other half; this is intended to speed up the data transmission when reducing the amount of sensor nodes managed by one single gateway; in this case each WSN operates in different frequency channels.

The address assignment is implemented as follows:

− Gateway Nodes: 0x 00 00 – 0x 00 0F.
− Relay Nodes: 0x 10 00 – 0x 10 0F.
− Sensor Nodes: 0x0 coach/car (4 bits) bogie (4 bits) wheel (4 bits). Left side wheels have an odd value and right side wheels have an even value. For example:
  − 0x 01 11: first car, first bogie, first wheel (first row/axle, left side).
  − 0x 03 24: third car, second bogie, fourth wheel (second row/axle, right side).

Figure 11. Network Architecture
III.3 Software Description

The software used to handle the hardware, described in section III.1, is explained in this section. The software may be divided into the system software and the application software.

III.3.1 System Software

System software comprises the files and programs responsible of interacting with the hardware. It includes the operating system, the firmware, drivers, servers, assemblers, compilers, interpreters and other system utilities.

III.3.1.1 GNU/Linux

The operating system implemented is formed by the Linux Kernel and the OpenWrt firmware. The Kernel is the basic component of an operating system and is in charge of resource allocation, low-level hardware interfaces and file system, process, memory and device management. The version of the Linux Kernel used was the 2.6.29.1; however this may vary due to the constant release of new stable versions.

III.3.1.2 OpenWRT firmware

OpenWrt is defined as a Linux distribution for embedded devices like gateways. It provides a fully writable file system with package management. These features allow configuring the gateway only with the packages that are required by the peripherals and the applications developed.

OpenWrt employs Linux embedded tools like uClibc, busybox and shell interpreter. The uClibc is a C library, much smaller than the GNU C library, for developing embedded Linux systems. The busybox is a small executable software containing many standard Unix utilities and commands.
III.3.1.3 Drivers and Packages

The packages and drivers installed in the gateway are described in this sub-section. These are required for the proper functioning of the system.

- **bash**: it is a UNIX shell. It is required by several scripts to execute their commands.

- **fftw3**: it is a C library for computing the Discrete Fourier Transform (DFT) of an input signal.

- **perl**: it is the interpreter of Perl scripts. The web application is programmed in Perl, therefore this package is required. The following Perl modules must also be installed: CGI, essential, file, getopt and xsloader.

- **ppp**: kernel driver to support connections using point-to-point protocol.

- **pppd**: point-to-point protocol daemon. It works with the kernel PPP driver to establish and maintain a PPP link with the remote server. It is used to negotiate IP addresses and for authentication.

- **wpa-suppliant**: it is an implementation of an IEEE 802.11i supplicant for Linux. It also supports WEP and WPA. It is required to establish the Wi-Fi connection.

- **wpa-cli**: it is used to control the wpa_supplicant.

- **ath5k**: it is the driver needed for the wireless mini-PCI adapter. It requires mac80211 and cfg80211 modules.

- **ftdi-sio**: it is Future Technology Devices International (FTDI) USB serial converter driver to detect the TelosB when connected to the USB port.
• **usbserial**: it is used to bind the USB modem. It must be loaded with the information about the modem’s vendor and product codes when the device is not supported by the driver.

### III.3.1.4 Servers

The server installed is the lighttpd; it is a secure, fast and flexible web server. It consumes less CPU and RAM than other web servers. Lighttpd supports virtual hosting, authentication, SSL cipher, CGI, SCGI and FastCGI. Therefore, lighttpd can communicate with external programs written in almost any programming language.

The authentication and the CGI modules must also be installed; the first one is used to restrict the access only for those who know the user and password; the latter module permits a web client to get information from programs executed at the web server.

### III.3.2 Application Software

The programming languages used to develop the applications were basically C, Perl and Shell scripts. The C programs were compiled using the gcc cross-compiler that is installed with the OpenWrt; only the executables were copied to the gateway board.

#### III.3.2.1 Data Collection and Management

The main program was designed to receive, process and store the data sensed. This program is described step by step along this section.

The interface with the WSN is the gateway node. This mote receives the data from the WSN and transmits it to the gateway through the serial port. The gateway must be able to receive and understand the frames sent by the mote.

This application uses other applications provided with the TinyOS distribution. TinyOS is an open-source operating system designed for wireless embedded sensor networks; it includes network protocols, sensor drivers, distributed services and data
management tools. TinyOS also includes some applications written in C to handle the serial communication system for mote-to-PC data exchange.

When the gateway node is connected to the USB port, it is recognized by the USB FTDI driver and can be found as a file in the /dev directory with name /dev/ttyUSBX, where the X depends on the port used to connect the mote. One of the TinyOS’ applications is the *serialsource* which consist in opening the serial device port at a specified baud rate, creating a file descriptor which is used for any I/O operations, reading the frames received from the mote and writing the frames sent to the WSN using commands like open, read and write. Other application provided by TinyOS is *message* that defines the frames’ header fields to read and write using the correct frame format and provides a function to read and write the frames’ payload based on the offset and the length of each field.

Frames transmitted through the network are formed by the MAC header, the AODV header, the AM header and the payload which includes the data sensed, as shown in Figure 12. Each frame is composed by the following fields:

- **MAC Header:** it is a 9 bytes header defined for 802.15.4 systems and it is divided into the following fields:
  - Frame Control: 2 bytes.
  - Sequence Number: 1 byte.
  - Destination PAN Address: 2 bytes.
  - Destination Address: 2 bytes.
  - Source Address: 2 bytes.

- **AODV Header:** it is the header added by the routing protocol (AODV). It is formed by 9 bytes distributed in the following fields:
  - Single Hop Sequence Number: 1 byte.
  - Multi Hop Source Address: 2 bytes.
  - Multi Hop Destination Address: 2 bytes.
  - Application Field: 1 byte.
Multi Hop Sequence Number: 1 byte.

TTL field: 1 byte.

Flag field (optional): 1 byte.

AM Header: it is a 1 byte field with the information about the type of data being carried within the frame.

<table>
<thead>
<tr>
<th>MAC Header</th>
<th>AODV Header</th>
<th>AM Header</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B</td>
<td>9B</td>
<td>1B</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12. MAC frame format.

MAC and the AODV headers are transparent to the gateway board. When the gateway node receives a frame, it only takes the information that must be sent to the gateway board through the serial interface which is the AM header and the payload. It adds the Serial Protocol Header that is composed of 7 bytes distributed in the following fields:

- Serial Active Message (AM) ID: it is a 1 byte field set to 0 and identifies the frame as a serial protocol frame.
- Destination Address: 2 bytes.
- Source Address: 2 bytes.
- Payload Length: 1 byte. It contains the number of bytes being carried within the payload field.
- Group ID: 1 byte.

The value of the AM header depends on the type of data exchanged within the network. There are three types of frames defined to accomplish the specifications of this project. The types of frames together with their AM value (hexadecimal) are:

- Temperature Data Frames: 0xD0. These frames are sent continuously with the temperature data as part of the payload field.
- Vibration Data Frames: 0xE0. The vibration data is transmitted within these frames’ payload.
Control Frames: 0xC0. These frames are sent from the gateway to the sensor nodes. Nodes should send a confirmation to the gateway when receiving one of these frames; the AM value for confirmations is also 0xC0.

The final frame sent is composed of the Serial Header, the AM Header and the payload. The payload is defined depending on the information transmitted. TinyOS application adds 3 bytes at each end of the frame before sending it through the serial port. The frame format is shown in Figure 13. The added bytes are:

- A framing byte to indicate start of frame (1 byte).
- Type of protocol: ACK, NACK or Unknown (1 byte).
- Sequence number (1 byte).
- CRC from the field of sequence number until the end of the payload, this field is added after the payload (2 bytes).
- A framing byte to indicate the end of the frame (1 byte).

![Figure 13. Serial Frame format.](image)

There are two types of data acquired from the WSN, temperature data and vibration data. The temperature measures are constantly sent (e.g. every 60 seconds) by each node of the WSN. The vibration data is only sent on request. The program was designed to always read from the file descriptor, opened by the TinyOS application, to process all the frames sent by the gateway node.

**Temperature Frames Reception**

The payload of temperature frames (shown in Figure 14) is composed of 6 bytes distributed in the following fields:
− Source Sensor Node Address: it is a 2 bytes field containing the address of the node that sends the temperature sensed.
− Source Relay Node Address: this field holds the 2 bytes address of the relay to which the source sensor node is associated.
− Temperature: it is 2 bytes of temperature data sensed.

![Temperature Frame Format](image)

When receiving one frame, the AM field is checked. If it carries temperature information, then the values of the payload fields are extracted and the information is stored in files. For each temperature frame received, the data stored in the temperature files is:

− Two bytes for the address of the source sensor node.
− Two bytes for the temperature data.
− Eight bytes for the timestamp when the frame was received.

The size of the temperature files is less than 1 Kbyte, storing around three temperature measures of each sensor.

The communication with the sensor nodes is limited to the reception of temperature frames. In order to save energy, the sensor nodes are in sleep mode between each temperature frame sent, however they remain awake some time (e.g. 3 seconds) after sending a temperature frame waiting for instructions from the gateway. If they don’t receive any message in that short time, then they change to sleep mode until the next temperature frame must be sent.
Therefore the only chance to interact with the sensor nodes is when receiving a temperature frame. The next step after storing the temperature information in files is to check if any request must be sent to the sensor node. There are three types of requests:

- Capture vibration data.
- Send vibration data.
- Change a parameter.

Frames sent from the gateway to the sensor nodes are Control Frames, shown in Figure 15; it means that the AM value is 0xC0. The control frame’s payload is composed of the following fields:

- Source Sensor Node Address: 2 bytes.
- Source Relay Node Address: 2 bytes.
- Control Type: it is 1 byte to identify the type of request.
- Control value: its length depends on the type of request.

The hexadecimal values of the Control Type field are:

- **0xB4**: requests sent by the gateway to sensor nodes to capture vibration data.

The length of the control value field is up to 3 bytes specifying the dimensions or axes (0, 1, and/or 2) along which the vibration must be measured. The dimensions required are defined by the user and may be modified. The sensor nodes must confirm the capture.
− **0x01**: requests sent by the gateway to sensor nodes to order the transmission of vibration data. This type of frames does not need a control value field. Confirmation is not needed, instead sensor nodes must initiate the vibration data transmission on receiving one of this frames.

− There are three parameters involving the gateway and the sensor nodes that may be modified by users. If one of these parameters varies, the gateway will send requests to all sensor nodes in order to change the specified parameter. The values of the control type field for each request frame are described below. On receiving one of these frames, sensor nodes must adjust the requested parameter and send a confirmation frame to the gateway:

  − **0xB1**: change the period of time between temperature data frames. The new period is sent in 1 byte as the control value field. Thus, possible values are between 0 and 255 seconds.

  − **0xB2**: change the frequency used to sample the vibration signal. The frequency is a 1 byte value between 0 and 255 Hz, when receiving this frame, the control value field (sampling frequency) is multiplied by 100, then sampling frequency values are ranged from 0 to 25500 Hz.

  − **0xB3**: change the number of samples. The value is sent in 1 byte and represents a power of two. For instance, if the value is 12 then the number of samples will be 2 raised to the power of 12, it means 4096 samples. Then the possible number of samples is between 2 and 2 raised to the power of 255; however the more common values are 9, 10, 11, 12 and 13 which represent 512, 1024, 2048, 4096 and 8092 samples respectively.

  − **0xE1**: confirmation (ACK) sent by the sensor nodes to acknowledge the reception and execution of a request. The control value field is the control type field of the request being acknowledged. It means that the control value field may be 0xB1, 0xB2, 0xB3 or 0xB4.
Only one type of request frame is sent at a time, for instance, while requesting captures no other requests may be sent (the same is applied to all request types). Capture and transmission of vibration data are two different processes that must be done in that order, finishing the capture process leads to start the transmission process. There is a maximum time to complete each process within all sensor nodes in the network, that time is measured by two different timers. Both processes are explained below. Timers are described later in this section.

**Capture Process**

Vibration data must be captured only when the train reaches some specific speeds, then if the train speed has reached a pre-defined value when a temperature frame is received, a capture flag is activated and a request of capture vibration data must be sent to the sensor node. If the sensor node does not confirm the capture, the request will be sent again to that sensor when it sends the next temperature data frame.

Capture process ends when: all nodes have confirmed their captures, train speed changes or the capture timer expires. If the train speed changes and it is not required to capture vibration data anymore, then the capture flag is deactivated.

A sensor node performs a series of operations upon receiving a capture request. The accelerometer generates an analog signal representing the acceleration. The values of the signal vary between 225g and -225g. The signal is converted to digital using an analog-to-digital converter with a resolution of 12 bits (4096 quantization levels) and a sampling frequency specified by the user which must be higher than twice the highest frequency of the signal. The node takes the number of samples required and stores them. Sensor nodes send those samples in vibration data frames when they receive the control frame requesting the vibration data.

**Transmission Process**

The process of requesting vibration data starts when the capture process is finished. There is a list with all the active sensor nodes in the network and their information
about whether they have confirmed or not the captures. When a sensor node, which has previously confirmed the capture, sends a temperature frame it is requested to send its vibration data. The sensor node must start the transmission on receiving this request. No other requests can be sent while awaiting the response of the sensor node in order to avoid losing vibration data frames due to collisions in the serial bus. If the sensor does not respond in a given time, that request will be considered as lost and a request will be sent to another sensor.

As vibration data corresponds to all the signal’s samples, they have to be transmitted in several frames. The payload of vibration data frames (shown in Figure 16) is composed of 102 bytes distributed in the following fields:

- Source Sensor Node Address: 2 bytes.
- Source Relay Node Address: 2 bytes.
- Sequence Number: 1 byte. The order of the data is essential to recover the original signal from the samples.
- Dimension: 1 byte. It specifies the axis along which data was sensed.
- Vibration Data: 96 bytes. Each sample is represented by 12 bits; therefore each vibration data field carries 64 samples.

![Figure 16. Vibration Frame Format.](image)

Vibration data is stored in an array whose size is equal to the number of samples. This array is used to detect alarms and create the vibration file (further explanation in section III.3.2.2). Each vibration file is particular to one sensor node and its size depends on the number of samples. The information stored is:
− Source Sensor Node Address: 2 bytes.
− Dimension: 1 byte.
− Sampling Frequency: 4 bytes. This is a network’s parameter that may change. It is necessary to store the frequency used to sample the data contained in the file to the correctly data processing at the server.
− Number of Samples: 4 bytes. It is required for the same reasons as the previous value.
− Capture Timestamp: 8 bytes. This timestamp is generated when the sensor node confirms its capture. It is possible to know the position of the train when the capture was done comparing this value with the GPS information.
− Train Speed: 4 bytes. The data pattern acquired depends on the train speed. This information is required to properly analyze the data at the user end.
− Vibration Data: each sample is stored in 2 bytes. Then the size of the data stored is the number of samples multiplied by 2 bytes.

During the reception of a vibration file the gateway will not send any requests to avoid collisions in the serial bus and guarantee the complete data delivery. The gateway will wait a given time for the total transmission of the vibration file; if this time expires and the transmission have not finished due to errors in the communication, the file is discarded and other request may be send to other sensor nodes. If a sensor node fails in sending its vibration data, it will be asked again to send its data. Each sensor node has three opportunities to send its vibration data if the capture flag is up; after the third failure the sensor node is requested to capture new data when the transmission process ends. This condition is required to speed up the transmission process when successive data captures must be done.

The process of requesting vibration data finishes when: all sensor nodes have sent their captured data or the timer expires. If the capture flag is up, the capture process starts again.
Changing Parameters

There is a configuration file with certain parameters that users could modify. Three of these parameters are also related to the applications running at the sensor node. Therefore, when those values are modified, the gateway must send a control frame to the sensor node indicating the parameter and its new value.

One of the parameters is the **duration of the sleep mode**, which is also the time between temperature frames transmissions. The other parameters are the **sampling frequency** and **number of samples** that are used in applications related to the vibration data in both sensor nodes and the gateway.

The configuration file is constantly checked. When a modification is detected, requests to change parameters are sent to sensor nodes if the capture or transmission process is not being executed or if none of the sensor nodes have a pending capture to be transmitted.

The sampling frequency and number of samples are two essential parameters for vibration data treatment, thus data misinterpretation may occur if modifications are applied during the transmission process or the capture process.

Therefore, frames to request the capture or transmission of vibration data will not be sent to sensor nodes that have not sent the confirmation frame to acknowledge the parameter’s modification.

Requests to change parameters are sent to sensor nodes until they acknowledge the changes.

Timers

Four timers are used in this application. They are mentioned and explained below with the names used within the program.
− **request_vib_timer**: it is set after sending a vibration data request with the maximum time that will wait until the sensor node starts the transmission. The value of the time interval depends on the response time of the sensor nodes. If this timer expires and the sensor node has not replied, the request may be sent to another sensor node and the timer is set again.

− **vibration_timer**: it is activated when the transmission process starts. The time measured is the result of the sum between average time of vibration file transmission and the time among temperature’s frames multiplied by the number of active sensor nodes in the network. The transmission process finishes if this timer expires before receiving all vibration data files.

− **capture_timer**: it is set when the capture process begins. The time interval is 180 seconds. The capture process will be stopped if it does not conclude within this time. The value may be changed in the header file.

− **vib_file_error**: it is started when the first vibration frame is received. It is set to 180 seconds, which may be modified in the header file. If this timer expires before receiving all vibration data frames, the vibration file will be discarded.

There are two main functions related to timers. The timer set function consists in storing the time when this function is called (initial time) and the interval of time to wait, the unit used is milliseconds. The timer expired function compares the elapsed time and the interval of time defined when the timer is set; the elapsed time is the difference between the time when this function is called and the initial time; the timer expires if this difference is equal or higher than the time interval defined.

**Header File**

This file (config.h) includes libraries used, constants and data types definitions. The main application creates four threads for data treatment. The header file is included in those five programs.
III.3.2.2 Alarms

All temperature and vibration data received from the WSN must be checked in order to report irregularities in the system.

Besides storing data in files, it is also temporarily stored for real-time processing. Useful information from each sensor is maintained in different fields of a list (array). The content of this list is described below.

- Last temperature data received.
- Timestamp generated when receiving the last temperature data.
- Flag to indicate whether the sensor node has confirmed the vibration data capture or not.
- Flag to indicate if a request to change parameters should be sent to the sensor node; it is modified upon receiving the confirmation frame.

Addresses of all active sensor nodes are stored in other list (array) when the first temperature data frame is received from each node. This array is used as a reference to place information in other lists in the same order and in this way link the information with its corresponding source sensor node. For instance, the third field within the addresses list must be checked to know the address of the sensor node that has generated the third temperature data entry in the information list.

Vibration data is also temporarily saved in an array linked to the addresses array. Each field of the vibration data array comprises:

- Three pointers to uint16_t (2 bytes unsigned integer), one for each possible dimension of vibration measures (x, y, z). The length of each vibration sample extracted from vibration data frames is 12 bits, but they are stored in 2 bytes (16 bits) to facilitate data handling.
- Flag to indicate if the data is ready to be processed (1 byte).
- Timestamp when vibration data was captured (8 bytes).
- Train’s speed when the request to capture vibration data was sent (4 bytes).
When the first frame of the vibration file is received, memory is allocated to one pointer depending on the dimension field value. The assigned memory size is the number of samples multiplied by size of uint16_t (2 bytes) where all vibration samples are stored for processing. After processing vibration data, it is required to free the allocated memory. When all vibration data frames are received, the flag is activated to indicate that data is ready to be processed. After processing, the flag is set to 0.

Four threads are created by the main program to manage information and detect alarms. These threads are:

- **check_config**: it is continuously checking the configuration file in searching of any modification. All variables that users can change are shared between this thread and the main program. Thus, changes of variables are done immediately after detecting them, excepting those variables related to vibration data. To apply these latter modifications, sensor nodes must not have vibration data captures pending to be sent and the transmission process should not be running. Those variables are: sampling frequency, number of samples and dimensions (axes) in which sensor nodes must capture vibration data.

- **read_fifo**: a FIFO is a special file that can be opened by multiple processes for reading or writing. FIFOs are created with *mkfifo* command and it must exist before any readings or writings. A FIFO (*gps_vel*) is shared between the GPS data management application and this thread. The GPS application writes train’s speed in *gps_vel* which is read by *read_fifo*. FIFOs must be opened in both ends. This thread reads the train’s speed and compares it with certain pre-defined speeds; if it matches, the capture flag is activated and the values to detect vibration data alarms are extracted from the configuration file.

- **registrar**: it may trigger three types of alarms. It constantly checks each field of the information list taking the last temperature received and its
corresponding timestamp to do further processing. This thread detects inactive sensor nodes and temperature alarms. A sensor node is considered inactive when it has not sent a temperature data frame within a given time; this situation generates an alarm of type 3. Temperature alarms are divided in two types:

- **Type 1.** Temperature measures registered by each sensor attached to each train’s wheel are higher than a given maximum temperature value.
- **Type 2.** The difference between temperatures measured by sensors located in wheels of the same axle is greater than a pre-defined limit.

**do_fft:** it processes vibration data looking for alarms. It checks the flags within vibration data array; it starts the processing when finding a flag set to 1. To detect vibration alarms it is required to analyze the amplitude spectrum of the signal. The Discrete Fourier Transform (DFT) is computed using vibration samples. Vibration pattern changes depending on the train’s speed, thus users shall specify the velocities with their related frequencies and maximum amplitudes in order to verify the vibration alarms. Those values are stored in the configuration file and are used when the speed of the train is equal to any required speed. If any of the frequencies checked exceeds its maximum amplitude, a vibration alarm is triggered and the information that has generated the alarm is sent to the remote server.

**Temperature Alarms**

Three types of alarms may be detected with temperature data frames. The continuous reception of these frames indicates that sensor nodes are active. Alarm files are created upon detecting any failures, the corresponding information is written and then they are immediately sent to the remote server. Those files contain 1 byte indicating the type of alarm detected, which could be 0x01, 0x02 or 0x03. The information that follows varies regarding the type of alarm. For type 0x01 that information is:
− The address of the sensor node that has generated the alarm (2 bytes).
− Timestamp of the frame which carried the temperature that caused the alarm (8 bytes).
− Temperature value (2 bytes).

For type 0x02:
− The address of the sensor node located at the wheel in one axle end (2 bytes).
− Timestamp corresponding to the last temperature data received from previous sensor node (8 bytes).
− The difference of temperature that triggered the alarm (2 bytes).
− The address of the sensor node situated at the wheel in the other axle end (2 bytes).
− Timestamp corresponding to the last temperature data received from previous sensor node (8 bytes).

For type 0x03:
− The address of the sensor node that has generated the alarm (2 bytes).
− Timestamp when the sensor node is declared inactive (8 bytes).

This information is used to bind the alarms with the temperature information of each sensor saved in the database at the remote server.

**Vibration Alarms**

The vibration samples received in vibration data frames are expressed in bits from 0 to 4096 (quantization levels of the ADC) and present a DC component. To recover the original signal it is required to remove the DC component and convert the bit codes into acceleration units which may be m/s\(^2\) or g (referenced to the gravitational acceleration: 9.8 m/s\(^2\) approximately). The mean value of the samples is subtracted from each sample for the DC component suppression. The result is multiplied by the overall range of the sampled signal (450g) and divided by the quantization levels (4096), thus the final value of the sample is expressed in g.
The resulting samples are stored in a different array of double (8 bytes variable in C) which is used to perform the DFT to obtain the amplitude spectrum. Samples are also saved in a vibration file (described in section III.3.2.1).

The DFT is computed using functions from the *FFTW* library, which must be installed in the gateway. FFTW employs O(N log N) algorithms to calculate the DFT of real or complex input data; the computation of the transform is divided into two phases, the first one is the planner that generates a data structure called a plan that contain information about the fastest way to compute the transform and the second phase is the execution of the given plan to transform the input data.

FFTW defines several routines to transform different types of data. The routine used to analyze the vibration samples was one-dimensional DFT of real data. The input data (samples) are real numbers gathered in an array of double with ‘n’ elements. The output data is an array with ‘n/2 + 1’ elements of type *fftw_complex*, which is a double[2] to store the real and the imaginary parts of the complex number. The arguments passed to the planner are the number of samples (n), input array, output array and unsigned flags. Flags instruct the planner to whether find the optimal way to compute the transform or to estimate a reasonable plan. The DFT of the samples is computed using the created plan, which is most efficiently transformed using number of samples (n) that are products of small factors. The plan is destroyed when the execution is done.

The transform of purely real data satisfies the Hermitian symmetry, it means that the component at frequency f_k is the complex conjugate of the component at –f_k, thus half of the output is redundant; that is why the size of the output array is n/2 +1. The amplitude of each component is calculated as the magnitude of its complex value divided by the number of samples. The magnitude of a complex number is the square root of the number multiplied by its conjugate.

A two-sided power spectrum shows half the energy at the positive frequencies and the other half at the negative frequencies. It is possible to convert it to a single-sided
spectrum by discarding the negative frequencies and multiplying each component except DC by 2. For that reason, each element of the output array, excluding output[0], is multiplied by 2.

The obtained amplitudes are compared to the maximum values specified by users; if any component exceeds the limit an alarm is triggered and a file is created to store the information that generates the alarm. Alarm files are sent immediately to the remote server to report the failure. That file contains:

- The address of the sensor node that sent the vibration data (2 bytes).
- Timestamp when the sensor node captured the vibration data (8 bytes).
- Frequency at which the alarm was detected (4 bytes).
- The amplitude value that caused the alarm (8 bytes).

### III.3.2.3 GPS Data Management

The GPS receiver integrated within the gateway board provides positioning information each second. It is recognized as a serial port device, thus the application, used to read data from the receiver, opens the port ‘/dev/ttyS1’ and creates a file descriptor required to read from serial port. The information obtained from the GPS receiver is:

- Current time UTC.
- Position: latitude, longitude and altitude.
- Speed at the given time and position.

The application is constantly reading the file descriptor searching for data. It gets the information and generates entries with the following fields:

- Timestamp when data was retrieved from serial port (8 bytes).
- Latitude (4 bytes).
- Longitude (4 bytes).
- Speed (4 bytes).
– Packet ID (4 bytes).

Entries are stored in files containing the number of entries (2 bytes) followed by the previous information. The number of entries per file may be modified in the header file of this application. When a given number of GPS data files are created, they are sent to the remote server; the number of files can also be modified. The remote server stores those files at the database and uses the information to locate trains’ position at any time and to relate temperature and vibration measures with it.

**III.3.2.4 Data Treatment and Transmission**

All executables and files generated are stored in ‘/home’ directory. Shell scripts are located in ‘/etc/alstom’.

Temperature and vibration data files are created in ‘/home/listen’ directory. GPS files are generated in ‘/home/gpsdata’. Alarms files are placed in ‘/home/alarmas’, this directory has two sub-directories: ‘talarm’ and ‘valarm’ for storing temperature and vibration alarms respectively. File names have the following nomenclature:

– ‘T_Gateway ID_Timestamp’ for temperature files. Each gateway must have a unique ID to distinguish between all gateways within the rail network. Timestamp is expressed in seconds and indicates when the file was created.

– ‘V_Gateway Id_Sensor Node Address_Timestamp’ for vibration data files. Each sensor node generates a vibration file; the address of each node is specified within the file’s name to easily identify (without looking inside the file) what sensor node have sent the data. Timestamp is expressed in seconds and represents the time when the sensor node confirmed the capture of the vibration data.

– ‘GPS_Timestamp’ for GPS data files. Timestamp corresponds to the time, in second, when the first entry of the file was read.
– ‘V_Timestamp’ and ‘T_Timestamp’ for vibration and temperature alarms respectively. Where Timestamp refers to the detection of the alarm in seconds. Each alarm file (temperature and vibration) may have information of several sensor nodes.

When temperature and vibration data files are completed and closed, they are moved to ‘/home/datos/temperatura’ and ‘/home/datos/vibracion’ respectively. The ‘/home/datos’ directory is periodically compressed and sent to the remote server. Thus, if files were created in the ‘/home/datos’ directory, they might be compressed before being completed or closed. That is why they are created in ‘/home/listen’ and afterwards moved to ‘/home/datos’. But there could be errors if files are moved during the directory compression. Therefore, a file lock function must be used. The idea is to lock the directory while being compressed; this is done using the `fcntl` function. That function perform locking operations over open files, thus the moving and compression processes must have access to the same file that acts as a semaphore. The process of moving files executes a shared lock while the compression process executes an exclusive lock. The file that is going to be locked must exist, it is located in ‘/home/lock’ and its name is ‘locking_datos’, this is defined within the program; if the name or location of the file is changed then the program must be modified and recompiled.

The lock program is called with the first argument indicating if the lock is shared (r) or exclusive (w). For shared locks, the name of the file being moved must be passed as the second argument. The file (locking_datos) is opened for read and write. When a shared lock is set on a file, other processes are able to set shared locks on that file, but they prevent any other process from setting an exclusive lock on the protected file. An exclusive lock prevents any other process from setting shared or exclusive locks on the same file. There are two ways of setting the locks; the first one is `F_SETLK` that returns immediately if the lock cannot be set and `F_SETLKW` which waits until the request is satisfied.
When a temperature or vibration file is completed, the lock function is called. If the shared lock can be set, the moving function is called; otherwise, the file is kept in ‘/home/listen’ and should be moved with the next attempt of moving a file. The lock function is also called when a data compression is required. It first tries to set an exclusive lock on ‘locking_datos’, if the file is already locked, it will wait until the file is released to set the exclusive lock and execute the compression.

The GPS application counts the completed files, when that count reaches a given number; it moves those files to ‘/home/gpsdata_tx’ and afterwards executes the compression of that directory.

When files of alarms are created they are immediately compressed to be sent to the remote server as soon as possible.

Compression is implemented using shell scripts. The script used for temperature and vibration files is executed by ‘cron’, which is a task scheduler used to execute processes at a specific time or periodically; the desired time of execution may be modified at anytime. Compressed files are created in different directories, where they are kept in case they cannot be sent to the remote server. Those directories are:

- ‘/home/ftp’ where the compressed ‘/home/datos’ is created with a name given by: ‘datos.Gateway ID.Date and Time.tar.gz’. Date and time are expressed in ‘day-month-year_hours-minutes-seconds’ and represent the moment when the compression was done.
- ‘/home/datos_gps’ where the compressed ‘/home/gpsdata_tx’ is created with a name given by: ‘GPS.Gateway ID. Date and Time.tar.gz’.
- ‘/home/alarmas_tx/temperatura’ and ‘/home/alarmas_tx/vibracion’ where the compressed ‘/home/alarmas/alarma’ and ‘/home/alarmas/valarma’ are created respectively. The name format for both directories is: ‘ALARMA.Gateway ID.Date and Time.tar.gz’.
The compressed files must be sent to the remote server. The transmission is done using the File Transfer Protocol (explained in section II.5.2).

An FTP client library was used to transfer files to the server, which must have an FTP server running. Routines implemented in the library are based in sockets to send FTP commands and receive the corresponding FTP server’s responses. The functions employed were:

- **ftp_connect**: it opens a connection-oriented socket on AF_INET domain and connects to the remote server using its IP address and the number of the FTP control port (usually 21). The socket is used to send FTP commands and collect the server’s responses.

- **ftp_auth**: it is used to logging in the remote FTP server, sending the user (USER) and password (PASS) through the created socket and waiting for authentication.

- **ftp_pasv**: it sends the PASV command to the FTP server to request the number of the data port on which it would be listening for file transmissions.

- **ftp_cwd**: it is used to change the working directory at the remote server; each type of file will be stored in different directories at the server.

- **ftp_put**: it uploads compressed files in the remote FTP server.

- **cliente**: it opens other socket on a pre-established port (e.g. 7345) with the remote server to indicate the type of file already sent, thus the server notifies the reception of files, decompreses them and stores the information in the database. Each type of file is handled according to their formats.

- **ftp_quit**: it is used to log-off from the remote FTP server when finishing all transfers.

A number is assigned to each type of file, so the remote server can distinguish and process the data received. The numbers can be changed or more types of files may be specified in the header file. The types of files and numbers defined are:
− GPS data: 2.
− Temperature and vibration data: 3.
− Temperature alarms: 4.
− Vibration alarms: 5.

The main program to send files, which includes the previously described functions, must be called specifying some arguments:

− Server IP address.
− Name of the FTP user, required to log-in.
− Password of the FTP user.
− Type of file.
− Absolute name of files that are going to be sent.

### III.3.2.5 Internet Access

Technologies used to access the Internet and connect to the remote server are Wi-Fi, UMTS and GPRS.

The PPP daemon is used to connect to the remote server via UMTS/GPRS. It requires a series of options (man pppd) to establish the communication. Those options are stored in a file named ‘isp’ which is located in ‘/etc/ppp/peers’. The content of the ‘isp’ file is:

− connect “/usr/sbin/chat -V -f /etc/ppp/chatscript-isp”: it specifies the command (chat) that is passed to a shell for execution prior to start PPP negotiation. The chat program defines a conversational exchange between the gateway and the modem. The file specified after the ‘-f’ option contains the script with the AT commands that allow chat to interact with the modem.
− /dev/ttyUSB*: this device is used to communicate with the peer. The modem is attached to this device.
− 115200: is the desired baud rate for the serial device.
− ctscts: serial port should use hardware flow control using RTS and CTS signals.
− modem: use the modem control lines.
− noccp: disables Compression Control Protocol (CCP) negotiation.
− nobsdcomp: disables BSD-Compress compression scheme.
− novj: disables Van Jacobson TCP/IP header compression.
− defaultroute: adds a default route to the system routing tables.
− noipdefault: the peer should provide the IP address during IPCP negotiation.
− usepeerdns: requests the peer up to two DNS server addresses.
− user name: it is required during the authentication phase.
− hide-password: password for logging is excluded from the log information.
− noauth: do not require that the remote server authenticates itself.

In the authentication phase, the gateway must identify itself with a user’s name and a secret (password). The authentication protocols used are Password Authentication Protocol (PAP) and Challenge Handshake Authentication Protocol (CHAP). PAP consists in sending the user’s name and a cleartext password, while CHAP involves the other peer sending a challenge with its name to the gateway, the gateway responds with its name and a hash value resulting from the password and the challenge. Pppd stores secrets in files, ‘/etc/ppp/pap-secrets’ for PAP and ‘/etc/ppp/chap-secrets’ for CHAP. Each line of both files contains the user’s name, the peer’s name and the secret. The peer’s name may be identified with ‘*’ to allow authentication with any peer.

The chat script file (/etc/ppp/chatscript-isp) comprises the AT commands to control and communicate with the modem. It presents a collection of commands and expected responses for each command. If the response is the same as the expected one, then the following command is executed. If there is an error, it exits and the connection is canceled. The lines within ‘chatscript-isp’ are:
− ‘ ’ AT&F: there is nothing expected before the command. It requests the modem to restore factory settings.
− OK AT+CPIN=XXXX: if the previous command was successfully executed, then the expected response is OK. This command specifies the pin.
− OK AT+CGDCONT=1, “IP”, “movistar.es”, “ “, 0, 0: defines the Packet Data Protocol (PDP) context, which represents the data session and contains the subscriber session information along with the parameters required to establish an end-to-end connection. The first parameter of this command is the ‘cid’ which is a numeric value that identifies a particular PDP context definition. The second parameter is a string with the type of packet data protocol used. The third parameter is the Access Point Name (APN). The fourth parameter specifies the PDP address, if the value is omitted a dynamic address will be requested. The two final parameters are for PDP data compression and PDP header compression respectively, 0 means that those options are not implemented. (16).
− OK AT+CGMI: after the OK response from previous step, it requests manufacturer identification. This command is only for additional information; it is not required to establish the communication.
− OK AT+CSQ: this informative command returns the signal quality.
− OK ATDT*99#: it is a tone dial command; it dials the sequence specified after the command. The expected response is CONNECT.
− CONNECT \d\c: the communication fails if CONNECT is not received.

To start the PPP daemon, it is executed as: ‘pppd call isp’. The call option specifies that additional options must be read from the file named ‘isp’ which is located in ‘/etc/ppp/peers’. The establishment of the PPP link was explained in section II.4.2.

The other technology used to access the Internet was Wi-Fi (IEEE 802.11). The program employed to access wireless local area networks was wpa_supplicant, which is designed to be a daemon program that runs in background and controls the wireless
connection. This program uses a configuration file with a list of all accepted wireless networks and their specifications. The basic supported options for each network are:

- Network name (SSID).
- Broadcast or send specific Probe Request when scanning for the specified SSID.
- Priority level of each defined SSID. Defines in which order networks are matched against the scan results.
- Security mechanism: WPA, WPA2 or WEP.
- Key management protocols: PSK, EAP (with several EAP methods) and IEEE 802.1X with EAP.
- Pairwise cipher for WPA: CCMP and TKIP.
- Broadcast cipher for WPA: CCMP, TKIP, WEP104 and WEP40.

A script located in ‘/etc/init.d’ executes wpa_supplicant when the system is started; wpa_supplicant requests the kernel driver (ath5k) to scan BSSs within its range, it selects a BSS according to specifications in the configuration file and requests the kernel driver to associate with the selected BSS. After association, wpa_supplicant handles the authentication with the security conditions described in the configuration file. It configures encryption keys for unicast and broadcast and the connection is established.

It is required a front-end program to interact with wpa_supplicant, that program is wpa_cli which may request information about the current connection as well as change the configuration file when is executed with specific options. It is also used in daemon mode executing an action file (shell script) depending on the events of wpa_supplicant. The action file (wpa_action.sh) handles ‘CONNECT’ and ‘DISCONNECT’ events triggered by wpa_supplicant when connecting or disconnecting from a wireless network.

The action file contains commands that must be executed on connection or disconnection; the IP address must be requested and the PPP daemon should be killed
when ‘CONNECT’ event is triggered. In contrast, when ‘DISCONNECT’ event is noticed the IP address of the wireless interface is set to 0 and the PPP daemon is started. The IP address is requested using ‘udhcpc’ which is a DHCP (Dynamic Host Configuration Protocol) client. PPP daemon must be killed when connecting through Wi-Fi to avoid any confusion about the route used to send files to the remote server. WLAN connection has priority over mobile connection.

The basic options used when executing wpa_cli (man wpa_cli) on the command line are:

- list_networks: provides the list of networks specified in the wpa_supplicant configuration file.
- add_network: each network has a unique identifier. When executing this option, wpa_cli adds a new network and returns the identifier.
- set_network <id> <parameter> <value>: assigns value to parameter of the network identified with id.
- get_network <id> <parameter>: returns the value of parameter of the network specified with id, passwords are not showed with this command.
- select_network <id>: selects the network identified with id and disables the rest of networks.
- enable_network <id>: enables the network if this was disabled or recently created.
- disable_network <id>: disables the specified network.
- remove_network <id>: deletes the network and its specifications from the configuration file.
- save_config: saves all changes in the configuration file.

### III.3.2.6 Web Application

A series of web applications were developed to allow users to change and configure parameters implemented in the gateway. They are a combination of server-side
scripts and client-side scripts, written in Perl and html respectively. Client-side scripts are in charge of presenting the information and server-side scripts are responsible of storing, processing and retrieving the information.

The offered functionalities are:

- **Parameters configuration**: all parameters stored in the configuration file are shown. Users may whether consult their values or modify them. If there is any modification, it is stored in the configuration file.

- **Sensor nodes status**: presents a list with all sensor nodes that have sent at least one temperature data frame, with their status and the time of the last temperature data frame received from each sensor. The status of the nodes could be ‘ACTIVE’ or ‘INACTIVE’, depending on how long they have been without transmitting temperature data frames.

- **Wi-Fi configuration**: presents the list of networks specified in the wpa_supplicant configuration file. It permits to add, modify and delete networks implementing wpa_cli commands and saving changes into the wpa_supplicant configuration file.

- **3G/UMTS configuration**: shows the status of the connection whether it is connected via GPRS/UMTS or Wi-Fi. It allows connecting or disconnecting from mobile network just by pushing a button. It also permits changing the PPP parameters by reading and writing from ‘/etc/ppp/chatscript-isp’, ‘/etc/ppp/peers/isp’, ‘/etc/ppp/pap-secrets’ and ‘/etc/ppp/chap-secrets’. Those parameters are: authentication protocol (PAP, CHAP or AUTO), APN, user’s name and password, pin code, selection of DNS server (manual or auto).

- **History of files sent**: presents a list of files’ names sent to the remote server with date and time when the transmission was done. Files are sent periodically to the remote server and their names are stored in a file located in ‘/home/datos_enviados’ together with the date and time of the transmission. This application provides an option to send files immediately when clicking on a button.
− **Command execution**: allows users to type and execute UNIX/Linux commands. It prints the command output after execution.

**Configuration File**

The file containing the parameters that users can modify is the configuration file; it is named ‘parametros’ and is located in ‘/etc/alstom’. Those parameters are:

− **MUESTRAS**: number of samples represented as a power of two. If this parameter is changed it must be notified to all sensor nodes along the network.

− **LOGIN_FTP**: user’s name for FTP log-in at the remote server.

− **PASS_FTP**: password of the user stored in LOGIN_FTP to log-in at the remote server.

− **TIME_BTW_TEMP**: maximum time to consider a sensor node inactive. It is the limit of time elapsed between temperature data frames received from each sensor node.

− **DIF_MAX**: maximum temperature, in degrees centigrade, for the difference between measurements taken from each wheel comprising an axle.

− **TEMP_MAX**: maximum temperature, in degrees centigrade, of each individual measurement.

− **T_DATOS_SENSOR**: it is related to sensor nodes; it is the period of time (in seconds) for sending temperature data frames or the sleep mode duration. If this parameter is modified, it must be informed to sensor nodes.

− **DIMENSION**: dimensions (x, y and z) of vibration data captures. The format is: X, Y, Z. The value related to each dimension is 0, 1 and 2 respectively. A value of 7 means that the dimension is not required. For instance, if the value stored in the configuration file is: 7, 7, 2 it means that only the Z dimension is required.

− **SERVER_IP**: IP address of the remote server.

− **GW_ID**: string containing the gateway identification.
− **FREQ_MUESTREO**: sampling frequency. It must be multiplied by 100 Hz to obtain the real sampling frequency. It is stored in this format so it can be sent to sensor nodes, in case of modifications, in 1 byte of the control frame payload.

Besides the configuration file, there are two more files used for configuration. One of them stores parameters to detect vibration alarms and the other file contains parameters for GPRS/UMTS connection.

**GPRS/UMTS Configuration File**

It is named ‘parametros_ppp’ and is located in ‘/etc/ppp’. It is only composed of two parameters:

− **RESET**: is used to indicate if the GPRS/UMTS modem shall be reset after a given number of failed connections attempts. A value of 0 enables the reset and 1 disables the reset.

− **NUMERO**: number of failed connection attempts permitted.

**Vibration Alarms Configuration File**

The detection of vibration alarms depend on the train’s speed. Users may specify more than one speed at which they are interested in measuring the vibration pattern of wheels. With each speed range, users must indicate the frequencies and maximum amplitudes that are compared with the measurements of sensor nodes after computing the DFT. Speeds, frequencies and amplitudes may be introduced through the web application in parameter configuration section. The file format is the following:

− **NUM_VEL**: indicates the number of speeds defined in the file. The maximum is 10.

− **VELOCIDADES=0_max, 0_min, 1_max, 1_min, 2_max, 2_min…**: stores all speed ranges, specifying maximum and minimum speed values of each range separated with commas.
− *FREQ=f0, f1, f2, f3, f4*: ‘*’ identifies the speed range related to this set of frequencies (from 0 to 9). For instance, 0_FREQ corresponds to the first speed range specified. There may be five frequencies defined to be checked after implementing the DFT to vibration data.

− *AMP=a0, a1, a2, a3, a4*: ‘*’ identifies the speed range related to this set of amplitudes (from 0 to 9). They are checked at the frequencies specified by ‘*_FREQ’; for instance, the component at frequency f0 must have a maximum amplitude of a0.

For example: NUM_VEL=1; VELOCIDADES=180, 170; 0_FREQ=50, 100, 150, 200, 250 and 0_AMP=6, 8, 10, 15, 20. When the train reaches a speed between 170 and 180 Km/h, capture and transmission processes are implemented. After computing the DFT of vibration data, the amplitude of the component at frequency 50 Hz is compared with 6g and so on until checking all components at frequencies specified in 0_FREQ.

The screen captures of the web page are shown in the following figures, from Figure 17 to Figure 31.

![Web Index](image_url)
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

Figure 18. Parameters Configuration Part I

Figure 19. Parameters Configuration Part II
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

Figure 20. Sensor Nodes Status

Figure 21. Wi-Fi Configuration. List of Accepted Networks
Figure 22. Wi-Fi Configuration. Add New Network Parameters Part I

Figure 23. Wi-Fi Configuration. Add New Network Parameters Part II
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

Figure 24. Wi-Fi Configuration. Modify Network Parameters Part I

Figure 25. Wi-Fi Configuration. Modify Network Parameters Part II
Figure 26. Wi-Fi Configuration. Delete Networks

Figure 27. 3G/UMTS Configuration Part 1
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

Figure 28. 3G/UMTS Configuration Part II

Figure 29. History of Sent Files
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

Figure 30. Run Commands

Figure 31. Results Showed After Executing a Command
III.3.2.7 Vibration Files Transmission Protocol

Vibration Data is transmitted in several frames. It is required to employ a protocol to guarantee the correct reception of all frames. The protocol is implemented in the gateway node, because the TinyOS Serial Protocol defines that writes on the serial port device are blocking, then while writing (sending frames to the sensor nodes) it is not possible to read from the serial port device. Although the writing function is fast, some frames sent in bursts (vibration data is sent in bursts) may be lost. Therefore, when receiving a vibration data file, the gateway cannot send any frame. But during the file reception there should be some feedback to confirm or to request retransmissions of frames, then the protocol must be implemented in the gateway node.

A window mechanism is used to control the transmission; the sensor node sends vibration data frames in blocks of window's size. The window's size (N) depends on the network throughput. When receiving half of the window's size (N/2) frames, the gateway node sends an acknowledgement (ACK) frame and the sequence number of the last frame correctly received, confirming in this way that all previous packets were received, to assure continuous transmission and channel occupation. The gateway node saves vibration data frames in a buffer of size N, every time it has half of the window completed and in order (based on frames’ sequence number) it sends those frames to the gateway (board) through the serial port.

A data frame loss is detected by the gateway node when the frame is out of sequence, i.e. the sequence number received is different than the expected one. When there is a loss, the gateway node should send a negative acknowledgement (NACK) frame specifying the sequence number of the lost frame. If the sensor node receives the NACK frame, it should retransmit the requested frame and then continue the transmission from the last frame sent. The gateway node will not send and ACK frame until receiving the lost frame. When it finally receives the frame, it sends an ACK if the window or half of it is completed.
When a NACK frame is sent, the gateway node starts a timer. If the data vibration frame lost is not received before the timer ends, the gateway node will send the NACK frame again.

If the sensor node has not received an ACK frame after sending the Nth frame, it retransmits the last frame sent until receiving any frame from gateway node or until a timer expires.

The gateway node starts a timer when it sends an ACK frame; if the timer ends before receiving a data frame, it will send the ACK frame again.

The gateway node knows in advance how many frames it should receive based on the number of samples defined, thus the transmission ends when it correctly receives the expected number of frames.
Chapter IV. Results

Three tests were implemented to describe the network performance. The first test was done setting the window size of the vibration data files transmission protocol to 6 packets, in the second test 10 packets were used and in the third test were employed 14 packets. The period for temperature frames transmission used was 60 seconds.

The number of samples for each vibration file was 4096. Each sample is represented in 12 bits, which results in vibration files of 6144 bytes being transferred along the WSN.

Basically, all tests consisted on the constant execution of the capture and transmission processes. Timers (defined in III.3.2.1) were configured with the following values:

- request_vib_timer: 60 seconds.
- vibration_timer: variable time dependent on the number of active sensors in the network and the average time of the file transmission. The average used was 45 seconds (based in other tests) this was added to the time between temperature frames (60 seconds) and finally the result was multiplied with the number of actives sensor nodes.
- capture_timer: 180 seconds.
- vib_file_error: 180 seconds.

The tests were implemented in a laboratory with limited space, therefore all nodes were programmed to work at low power in order to assure multi-hopping. The network deployed, shown in Figure 32, was composed of:

- 7 relay nodes numbered from 1 to 7 linearly arranged, being 1 the closest to the gateway. 4 sensor nodes were associated to each relay node, except for the first test in which the relay number 6 worked with 3 sensor nodes.
− 28 sensor nodes numbered with two digits, the first one indicates the relay node at which it is associated and the second varies from 0 to 3. For instance, sensor nodes attached to relay node number 1 were: 10, 11, 12 and 13.
− 1 gateway node.
− The gateway board.

The first test, with window size equals to 6 packets, was executed during 18 hours; the second test, using a window size of 10 packets, lasts 9 hours and the third test, with a window size of 14 packets, was implemented in 21 hours. The average time of the capture and transmission processes along with the standard deviation were calculated for each test. Results are shown in Table 1.
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average Time</td>
<td>Standard Deviation</td>
<td>Coefficient of Variation</td>
</tr>
<tr>
<td></td>
<td>[sec]</td>
<td>[sec]</td>
<td>[%]</td>
</tr>
<tr>
<td>Capture Process</td>
<td>127.50</td>
<td>41.78</td>
<td>32.77</td>
</tr>
<tr>
<td></td>
<td>120.67</td>
<td>36.34</td>
<td>30.12</td>
</tr>
<tr>
<td></td>
<td>132.79</td>
<td>35.38</td>
<td>26.64</td>
</tr>
<tr>
<td>Transmission Process</td>
<td>1257.42</td>
<td>191.47</td>
<td>15.23</td>
</tr>
<tr>
<td></td>
<td>1142.10</td>
<td>226.43</td>
<td>19.83</td>
</tr>
<tr>
<td></td>
<td>1263.15</td>
<td>336.46</td>
<td>26.64</td>
</tr>
<tr>
<td>Total Average Time [sec]</td>
<td>1384.92</td>
<td>1262.77</td>
<td>1395.94</td>
</tr>
</tbody>
</table>

Table 1. Average Time, Standard Deviation and Coefficient of Variation for processes of each test.

Considering the average time and the standard deviation of the capture process, results are similar in all tests, because this process does not depend on the window size. However, the duration of transmission process varies depending on the window size; the highest average time and dispersion was obtained with a window of 14 packets while the lowest average time was achieved using a window size of 10 packets. When using a window size of 6 packets, it seems that sensor nodes send all packets that form one window before receiving a confirmation frame and then the expected behavior related to the vibration files transmission protocol is that sensor nodes will wait until receiving the acknowledgment. That time waited traduces in longer times for transmitting one file.

Relay nodes have limited storage capacity; their buffer size is for storing 3 packets, if there are many packets being transferred in the network it could lead to overflow and retransmissions due to packet losses causing the transmission process to take longer time to finish; this may be the case when using a window size of 14 packets. Therefore, the results point to the window size of 5 packets as the best option. The rest of measures taken from each test are shown and described in the following subsections.

**IV.1 Temperature Packets Received**

All requests sent to sensor nodes depend on the temperature packets received from each one. Receiving a temperature packet means that the sensor node is still active.
and awaked. Figure 33, Figure 34 and Figure 35 present the temperature packets received in test 1, test 2 and test 3 respectively.

Figure 33. Test 1 Temperature Frames Received (W=6)

Figure 34. Test 2 Temperature Frames Received (W=10)

Figure 35. Test 3 Temperature Frames Received (W=14)
The range of packets in the second graphic is different due to the duration of the second test. The first graphic, related to results of test 1, illustrates a similar number of temperature packets received from each sensor network, which is the expected because all sensor nodes are supposed to send temperature packets every 60 seconds, but there is an odd behavior in sensor nodes associated to relay node 5 which seems to be related with the link between sensor nodes and the relay node rather than a routing problem, because the rest of sensor nodes present a similar pattern.

Variations in the second and third graphics are related to the increase of packets being transferred in the network. There are some odd cases that must be related with the path from sensor nodes to their relay node.

When a relay node sends a packet to the next relay node (next hop), it maintains the packet in the buffer until receiving a confirmation packet. The packet is retransmitted up to ten times if the confirmation packet is not received. Relay nodes do not accept packets if their buffer is full. This could explain temperature frames loss.

The maximum packet loss ratio is between 10% and 8% related to the higher number of packets received.

**IV.2 Reply Ratio**

When a request frame is sent from the gateway to sensor nodes, a response frame is expected to be received, but either the request or the reply frame may not reach its destination. The percentage of reply frames received based on the number of request frames sent to each sensor node are shown in Figure 36, Figure 37 and Figure 38 for each window size used: 6 (Test 1), 10 (Test 2) and 14 (Test 3) packets respectively.
The average of reply frames received in each test is around 90%, results in the first and second tests vary between 70% and more than 90%; the same results are
extracted from the third graphic, excepting for an odd behavior of one sensor node, from which the reply frames received were less than 60% of the request frames sent.

Results obtained from test 2, with a window size of 10 packets, are globally closer to 90% while the test 3, with a window size of 14 packets, presents the lowest global results.

There are some sensor nodes with low reply ratio in each test; this may be the result of the bad orientation of those sensor nodes respects to their relay node. As the transmission power is very low, it was necessary to put the sensor nodes really close to the relay nodes, therefore slight differences of sensor nodes position may lead to significant differences in the reply ratio. This situation is similar than the one expected in trains where propagation paths will not be equal for each sensor node.

IV.3 Average Reply Times

It is interesting to know how long it takes a sensor node to respond after sending it a request frame. The average reply times for each test are presented in Figure 39, Figure 40 and Figure 41.

![Figure 39. Test 1 Average Reply Times (W=6)](image-url)
The expected pattern for these graphics is that sensor nodes located farther from the gateway take longer time to reply. This behavior can be seen in all previous graphics. The average reply time in the first graphic increases from 1.2 to 2.5 seconds with dispersion values between 30% and 60% of the average. Results from test 2 show similar reply times along the network excepting for some sensor nodes, when there is an odd value corresponding to one sensor node but the rest of sensor nodes within the same cluster behave as expected, it leads to assume that failures are specific to the transmission between that sensor node and its relay node. All average reply times are below 2 seconds with a standard deviation that varies from 30% to 70%. The third graphic, corresponding to a window size of 14 packets, shows maximum reply times
above 2.5 seconds and standard deviation from 30% to 80%; the increase of packets being transferred in the network affects the response times.

**IV.4 File Reception Ratio**

The proportion of files received from each sensor node compared with the number of files requested by the gateway is shown in Figure 42, Figure 43 and Figure 44. The results presented correspond to test 1, test 2 and test 3 respectively.

![Figure 42. Test 1 File Reception Ratio (W=6)](image)

![Figure 43. Test 2 File Reception Ratio (W=10)](image)
The average of received files for the three tests is more than 90%. It can be seen that while the traffic increases along the network, the number of received files per sensor node decreases. A vibration file being sent is discarded if the transmission takes longer time than the defined timers of applications involved in the transfer. The average times for file transmission obtained from these tests are presented in the next sub-section.

**IV.5 Average Time of Vibration Files Transmission**

It is important to know the time employed for transmitting a vibration file, in order to configure timers related to the vibration file transmission protocol and the application at the gateway that receives the files. The results obtained when using a window size of 6 packets are shown in Figure 45, for window sizes equal to 10 and 14 packets see Figure 46 and Figure 47 respectively.
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

Figure 45. Test 1 Average Time of Vibration Files Transmission (W=6)

Figure 46. Test 2 Average Time of Vibration Files Transmission (W=10)

Figure 47. Test 3 Average Time of Vibration Files Transmission (W=14)
When using a window size of 6 packets, the minimum average time for transmitting vibration files is around 15 seconds while the maximum is around 45 seconds and the standard deviation varies from 10% to 30%, then the average time for this window is around 29 seconds with 34% of dispersion. For a window size of 10 packets, the minimum average time is less than 15 seconds and the maximum is around 35 seconds with standard deviation between 9% and 66%; the average time of one vibration file transmitted using this window is 23 seconds with a standard deviation of 28%. Finally, for a window size of 14 packets, the minimum average time is 16 seconds and the maximum is 30 seconds; the standard deviation varies from 14% to 40%; the average time for transmitting one vibration file employing this window is 24 seconds and a standard deviation of 18%; which is a good result however it is the one that introduces more packets in the network.

The overall results show that the shortest times are obtained with a window size of 10 packets, while the longest times are the result of using a window size of 6 packets. The latter is related to the time waited by a sensor between sending all packets of the window and receiving a confirmation from the gateway node.

**IV.6 Average Vibration File Transmission Rate**

Using the previous results and based on the data used to develop all tests, the average transmission rate of vibration files can be calculated as the number of bits per file (6144 bytes * 8 bits) divided by the transmission time. Results for test 1, test 2 and test 3 are shown in Figure 48, Figure 49 and Figure 50 respectively.
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

Figure 48. Test 1 Average Transmission Rate of Vibration Files (W=6)

Figure 49. Test 2 Average Transmission Rate of Vibration Files (W=10)

Figure 50. Test 3 Average Transmission Rate of Vibration Files (W=14)
The sensor nodes that are closer to the gateway have higher file transmission rates, with maximum values of 3232, 3430 and 3074 bps for widow sizes of 6, 10 and 14 packets respectively; the minimum values registered were 1054, 1431 and 1630 bps.

**IV.7 Average Transmitted Packets for Correct received packets**

A vibration data file is composed of 64 packets (in these tests), when these packets are transmitted along the network there may be losses, in those cases the error correction protocol working at the application layer should retransmit those packets, so the final number of packets transmitted to complete one file is higher than 64 packets. The idea is to see how many packets are sent by the sensor nodes to receive one correct packet at the gateway. This value is calculated with the total number of packets sent by sensor nodes divided by 64 packets. Results for test 1, test 2 and test 3 are shown in Figure 51, Figure 52 and Figure 53.

![Figure 51. Test 1 Average Transmitted Packet for Correct Received Packets (W=6)](image1)

![Figure 52. Test 2 Average Transmitted Packet for Correct Received Packets (W=10)](image2)
The first graphic, corresponding to a window size of 6 packets, shows the expected behavior of the network, sensor nodes located farther from the gateway (measured by the number of hops) presents a higher number of packets lost than those located closest to the gateway. The minimum average of packet lost is 5% and the maximum is 47% respect to 64 packets. The third graphic related to test 3, with a window size of 14 packets, also illustrates the same pattern, excepting for sensor nodes associated to the relay number 3, this must be related to the cluster itself, because if it were routing problems, results from the rest of hops (from relay 4 to 7) should be affected. For instance, sensor node number 73 presents an average packet loss of 36% which is lower than the value obtained from the same sensor in test 1. The minimum average packet loss in this test is 11% and the maximum is 38%.

In the second graphic (Figure 52), there are odd behaviors in the clusters of relay nodes 3, 5 and 6. And it seems to be a cluster problem rather than a routing issue because nodes associated to relay node 7 are not affected. The minimum average packet loss is 3% and the maximum is 27%.

There are two sources of loses: the link between relay nodes and their associated sensor nodes and the link between relay nodes (multi-hop backbone). The link between each relay node and its sensor nodes depends on the position of sensor nodes because tests have been done with very low transmission power. Therefore, it is assumed that the odd behavior of test 2 is due to the fact that loses in the link between relay and sensor nodes are much higher than in the backbone.
Chapter V. Conclusions and Recommendations

V.I Conclusions

The main objective of the project was to develop a gateway platform to control and manage sensor nodes that acquire data from train wheels and transmit them to the central server to be treated for train maintenance.

The project has successfully developed the required modules to implement these functions: a system to poll sensor nodes and retrieve data, a system to analyse these data and detect malfunctions, a system to transmit data to the central system which stores it in a database, the management of all communication interfaces of the gateway (serial port towards the sensor network, serial port towards the GPS and cellular, 802.11 and Ethernet networks interfaces) and finally a web server to remotely configure the gateway.

The system has been widely tested under two kinds of tests. Firstly, during trials in the laboratory working at low transmitted power that endured for several continuous days. In this case the system showed that the vibration file transmission protocol is pretty reliable with an average of 90% of correctly received files, as well as the routing protocol that behaves as expected with no major differences between values obtained from sensor nodes located farther from the gateway (more hops) and those situated closer to the gateway. Related to the window size of the vibration files transmission protocol used in tests, the best global results were obtained when using a window size of 10 packets. When transmitting packets with a window size of 6 packets, the sensor nodes send all packets within one window faster than they receive confirmation frames from the gateway node, then they lose time waiting for acknowledgement, increasing in this way the overall time for file transmission. On the other hand, the network is overloaded when using a window size of 14 packets; each relay node has a limited buffering capacity, if the buffer is full they drop packets, leading to increments in transmission times and in packet loss.
The second kind of trials was on the field. The system has been tested twice inside a moving train from Toledo to Barcelona and the system showed a correct behaviour.

Therefore we can conclude that the system has been successfully developed, achieving the objectives set at the beginning of the project.

V.II Recommendations

There are several recommendations to enhance the system based on the trials, they are:

- For future improvements in the system it is recommended to modify the Serial Protocol in order to use non-blocking writing in the serial port. In this way, the protocol for vibration file transmission could be implemented in the gateway, and the capture and transmission times would be reduced because the gateway might send data while receiving a vibration file.

- An error detection mechanism, like CRC, could be implemented at the application layer to check that the received data has not been corrupted during the transmission.

- Other features, like controlling the gateway via SMS and acquiring other measurements from the wireless sensor network, may be developed.
References


Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

Acronyms

AES: Advanced Encryption Standard.

BSS: Basic Service Set.

CRC: Cyclic Redundancy Check

DFT: Discrete Fourier Transform.

EDGE: Enhanced Data rates for GSM Evolution.

EEPROM: Electrically Erasable Programmable Read Only Memory.


GPRS: General Packet Radio Service.

HSPA: High Speed Packet Access.

LCP: Link Control Protocol.

Mbps: Mega bits per second.

MT: Mobile Termination.

PPP: Point to Point Protocol.

SDRAM DDRII: Double Data Rate Synchronous Dynamic Random Access Memory.

SIM: Subscriber Identity Module.

TE: Terminal Equipment.

TKIP: Temporal Key Integrity Protocol.
UMTS: Universal Mobile Telecommunication System.

USB: Universal Serial Bus.

USIM: Universal Subscriber Identity Module.

WEP: Wired Equivalent Privacy.

WiMAX: Worldwide Interoperability for Microwave Access.

WPA: Wi-Fi Protected Access.

Annexes

Source Code
The source code developed for data collection and management as well as the threads to detect the alarms is presented in this section.

```c
#include "config.h"
#include "serialsource.h"
#include "serialpacket.h"
#include "serialprotocol.h"
#include "timer.h"
#include "actividad.h"
#include "packsend.h"
#include "th_def.h"

static char *msgs[] = {
    "unknown_packet_type",
    "ack_timeout",
    "sync",
    "too_long",
    "too_short",
    "bad_sync",
    "bad_crc",
    "closed",
    "no_memory",
    "unix_error"
};

/*Variables Globales*/
uint8_t alarma=0;

uint16_t id_sensor[NUM_SENSORES]={0};  /*Array que mantiene las direcciones de los sensores activos.*/
/*Se utiliza para relacionar los campos de los demás arrays*/

uint8_t contador_sensores=0; /*Número de sensores activos.*/

sensor_t lista_sensor[NUM_SENSORES]; /*Información (ver struct sensor_t) actualizable de todos los sensores.*/

fft_vib datos_vib[NUM_SENSORES]; /*Datos de los ficheros de vibración de los sensores para calcular la fft.*/

param_flags changes[NUM_SENSORES]; /*Controla los cambios de parámetros que deben ser notificados a los sensores.*/

uint16_t temp_max=8000; /*Temperatura máxima permitida.*/

uint16_t dif_temp_max=20; /*Diferencia de temperatura máxima medida por sensores en ruedas de un mismo eje.*/

double time_btw_temp=120; /*Tiempo para considerar un sensor inactivo*/

char gwid[20]="gateway10"; /*Identificador del gateway*/

uint8_t init_flag=0; /*Flag para iniciar el thread de read_fifo() luego de revisar el fichero de parámetros.*/

uint8_t req_capture=0; /*VERDADERO(1) O FALSO(0). Depende de la velocidad del tren dada por las lecturas del GPS.*/

uint8_t freq=40; /*x 100 Hz. Variable para detectar cambios de la frecuencia y enviar peticiones a los sensores.*/
```
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

uint32_t f_muestreo=4000; /* Frecuencia de muestreo */
uint8_t long_muestra=12; /* (2 ^long_muestra) muestras*/
uint8_t t_envio_sensor=60; /*Segundos. Periodo en que los sensores envian temperaturas.*/
float velocity=0; /*Varía dependiendo de las lecturas del GPS. Se lee del fifo con el proceso del GPS.*/
uint32_t muestras; /*Número total de muestras requeridas para los ficheros de vibración.*/
uint8_t param_changed[(PARAMETROS+1)]={0}; /*Indica qué parámetros han cambiado. Es modificado por check_config().
   *param_changed[0]=0 significa que no hay cambio.*/
dim_vib real; /*Parámetro temporal para la variación de dimensiones introducidas por el usuario.*/
int dim_len=1; /*Número de dimensiones requeridas para vibraciones.*/
uint8_t dim_value[3]={1,7,7}; /*Dimensiones requeridas x=0 y=1 z=2. Se inicializa para pedir la dimensión y.
   *Las dimensiones con valor 7 no son requeridas. Sólo valen 0, 1 o 2*/

/*Variables globales para do_fft*/
der double amplitudes[5]={1,2,3,4,5}; /*Amplitudes máximas de los armónicos requeridos.*/
uint32_t armonicos[5]={1,2,3,4,5}; /*Hz. Frecuencias que se deben revisar al calcular la fft.*/

/*Funciones llamadas desde el main*/

/*Imprimir todos los campos de packet con longitud len*/
void hexprint(uint8_t *packet, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("%02x ", packet[i]);
}

/*Escribir ceros en todos los campos del array a*/
void vaciar_arreglo(char *a)
{
    int i;
    for (i=0;i<sizeof(a);i++)
        a[i]=0;
}

/*Liberar memoria ocupada con la función malloc*/
void memory_free()
{
    int i;
    for(i=0; i<NUM_SENSORES;i++)
        if (id_sensor[i] != 0)
            free(datos_vib[i].datos_x);
            free(datos_vib[i].datos_y);
            free(datos_vib[i].datos_z);
}

Page 97
Revisar los flags de vibración

```c
int check_sensor()
{
    int i;

    for(i=0;i<NUM_SENSORES;i++)
    {
        if (id_sensor[i] != 0){
            if((lista_sensor[i].extra.capture_acked!=0)||(datos_vib[i].ready_to_process==1))
                return 1;
        }
    }

    return 0;
}
```

Revisar el array de cambios de parámetros para el sensor con dirección id_sensor[pos]

```c
int check_param(int pos)
{
    if(changes[pos].f!=0) /*Tiene un cambio de frecuencia de muestreo sin confirmar*/
        return 1;
    if(changes[pos].l!=0) /*Tiene un cambio de longitud de muestra sin confirmar*/
        return 2;
    if(changes[pos].t!=0) /*Tiene un cambio de periodo de envío de temperaturas sin confirmar*/
        return 3;
    return 0; /*No tiene cambios pendientes*/
}
```

Activar los flags que correspondan con los parámetros a cambiar para cada sensor

```c
int activate_flags()
{
    int i,h;
    uint8_t frecuencia=0, lon=0, period=0;

    for(h=1;h<(param_changed[0]+1);h++)
    {
        switch(param_changed[h]){  
        case 1:  
            frecuencia=1;
            break;
        case 2:  
            lon=1;
            break;
        case 3:  
            period=1;
            break;
        default:  
            break;
        }
        param_changed[h]=0;
    }

    for(i=0;i<NUM_SENSORES;i++)
    {
        lista_sensor[i].change_param=1;
        changes[i].f=frecuencia;
        changes[i].l=lon;
        changes[i].t=period;
    }
}
```
param_changed[0]=0;
return 0;
};

/*Contar los sensores que han confirmado captura*/
uint8_t confirmed()
{
    uint8_t i, cont=0;
    for(i=0;i<NUM_SENSORES;i++)
    {
        if (id_sensor[i] != 0){
            if (lista_sensor[i].extra.capture_acked!=0){
                cont++;
            }
        }
    }
    return cont;
}

/*Imprimir alguno de los mensajes de *msgs[]*/
void stderr_msg(serial_source_msg problem) {
    fprintf(stderr, "Note: %s\n", msgs[problem]);
}

/*Devuelve la posición en que se encuentra la dirección del sensor en el array id_sensor[].
 Si no se encuentra la dirección o no hay campos vacíos en el vector, devuelve -1.
*/
int encontrar_direccion(uint16_t direccion)
{
    int i;
    uint8_t j;
    for(i=0;i<NUM_SENSORES;i++)
    {
        if (id_sensor[i]==direccion){
            return i;
        }
    }
    for(i=0; i<NUM_SENSORES;i++)
    {
        if (id_sensor[i]==0){
            if ((lista_sensor[i].direccion != 0) && (direccion != lista_sensor[i].direccion)){
                for (j=0; j<NUM_SENSORES; j++){
                    if (lista_sensor[j].direccion == 0){
                        break;
                    }
                }
                lista_sensor[i].change_param=lista_sensor[j].change_param;
                changes[i].f=changes[j].f;
                changes[i].l=changes[j].l;
                changes[i].t=changes[j].t;
            }
            id_sensor[i]=direccion;
            contador_sensores++;
            return i;
        }
    }
    return -1;
}
/*MAIN*/
int main(int argc, char **argv)
{
    serial_source src;
    uint8_t flag_vib=0, j, vib_rcvd=0, error_flag=0, start_capture=0, start_tx=0;
    uint32_t move=0;
    FILE *ficherot=NULL;
    FILE *ficherov=NULL;
    char nombre_t[50],nombre_v[50], comandot[100], comandov[100];
    uint32_t contador=0, contadorv=0, cont_dim=0;
    uint16_t sensor_vib=0,relay_vib=0, sensor_source=0, relay_source=0;
    uint8_t num_seq=0, value_field;
    size_t index;
    double *ptr_datos=NULL;
    uint8_t dimension_vib=0, param;
    uint32_t pos_vib=0;
    uint16_t vibraciones[PACKET_SAMPLES];
    uint8_t total_packets, num_seq_max;
    int len, posicion;
    uint8_t am_type;
    /*Comprobar que se ejecuta con los argumentos necesarios*/
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <device> <rate> - dump packets from a serial port\n", argv[0]);
        exit(2);
    }
    /*Abrir puerto serie*/
    src = open_serial_source(argv[1], atoi(argv[2]), 0, stderr_msg);
    if (!src){
        fprintf(stderr, "Couldn't open serial port at %s:%s\n", argv[1], argv[2]);
        exit(1);
    }
    /*Inicializar variables*/
    for(j=0;j<NUM_SENSORES;j++){
        lista_sensor[j].direccion=0;
        lista_sensor[j].change_param=0;
    }
    for(j=0;j<3;j++)
        real.dimension[j]=dim_value[j];

    muestras=4096;
total_packets=(muestras*SAMPLE_BITS)/PACKET_BITS;
samples_last_packet=muestras-(total_packets*PACKET_SAMPLES);
if(samples_last_packet>0){
    total_packets++;
}
num_seq_max=total_packets-1;

/*Eliminar fichero de actividad antiguo*/
remove(FICH_ACT);

/*Crear los threads*/
pthread_create (&t_config, NULL, check_config, NULL); /*Revisa /etc/alstom/parametros por cambios de variables*/
pthread_create (&t_fft, NULL, do_fft, NULL);          /*Revisa alarmas de vibracion*/
pthread_create (&t_alarm_temp, NULL, registrar, NULL); /*Revisa alarmas de temperatura*/
pthread_create (&t_fifo, NULL, read_fifo, NULL);       /*Revisa la velocidad en el fifo con la aplicacion del GPS*/

/*Ciclo infinito para estar siempre escuchando el puerto serie*/
for (;;) {
    posicion=0;
    packet=NULL;
    ptrmsg=NULL;

    /*Si se pide un fichero de vibraciones y no hay respuesta, se inician las variables para pedir a otro sensor.*/
    if ((flag_vib==1)&&(vib_rcvd==0)&&(timer_expired(&request_vib_timer)){
        printf("Se vence el timer\n");
        flag_vib=0;
        sensor_vib=0;
        if ( (lista_sensor[pos_vib].extra.capture_acked++)  == 3) {
            lista_sensor[pos_vib].extra.capture_acked=0;
        }
    }

    /*Leer el puerto serie*/
    packet = read_serial_packet(src, &len);
    if (packet){
        printf("un"):print("velocity=%f, req_capture=%d\n", velocity, req_capture);
        printf("un"):hexprint(packet, len);printf("un"); /*Para las pruebas...¡¡¡QUITAR!!!*/
        if ((len >= (1 + SPACKET_SIZE)) && (packet[0] == SERIAL_TOS_SERIAL_ACTIVE_MESSAGE_ID)) {
            ptrmsg = new_tmsg(packet + 1, len - 1);
            if (ptrmsg){
                am_type=spacket_header_type_get(ptrmsg);
                sensor_source=spacket_data_tbyte_get(ptrmsg, 0);
                relay_source=spacket_data_tbyte_get(ptrmsg, 2);

                switch(am_type){
                    /*Si es un paquete de temperatura*/
                    case TEMP_PACKET:
                        /*Comprobar que el sensor está en la BD o añadirlo. -1 indica error*/
                        if (((posicion=encontrar_direccion(sensor_source)) != -1)){
                            /*Completar la informacion de temperatura del sensor*/
                            lista_sensor[posicion].direccion=sensor_source;
                            temperatura=spacket_data_tbyte_get(ptrmsg, 4);
                        }
                }
            }
        }
    }
}
gettimeofday(&lista_sensor[posicion].tiempo_actual, NULL);
if ((temperatura & 0xFF00) != 0x0000) {/*Para estadísticas!! QUITAR!!*/
    lista_sensor[posicion].temperatura_actual=temperatura;
    /*Si es el primer paquete -> abrir fichero nuevo*/
    if (contadort == 0) {
        sprintf(nombre_t, "%sT_%s_%ld.%ld", DIR_LISTEN, gwid,
            lista_sensor[posicion].tiempo_actual.tv_sec,
            lista_sensor[posicion].tiempo_actual.tv_usec);
        if ((ficherot = fopen(nombre_t, "w")) == NULL) {
            perror("Error abriendo el fichero "n");
            goto always;
        }
        sprintf(comandot, "/etc/alstom/lock_datosr %s", nombre_t);
        vaciar_arreglo(nombre_t);
    }
    /*Escribir datos en el fichero de temperatura*/
    fwrite(&lista_sensor[posicion].direccion,sizeof(uint16_t),1,ficherot); /*Dirección del sensor*/
    fwrite(&lista_sensor[posicion].temperatura_actual,sizeof(uint16_t),1,ficherot);
    /*Temperatura*/
    fwrite(&lista_sensor[posicion].tiempo_actual,sizeof(struct timeval),1,ficherot);
    /*Timestamp*/
    /*Actualizar el estado del sensor en el fichero activity*/
    actividad(posicion);
    /*Incrementar el contador de paquetes de temperatura recibidos*/
    contadort++;
    /*Si se alcanza el total de paquetes por fichero (MAX_PACK_TEMP) ->
     * cerrar fichero e inicializar variables.
     */
    if (contadort == MAX_PACK_TEMP) {
        fclose(ficherot);
        printf("Fichero de temperatura cerrado\n");
        system(comandot);
        vaciar_arreglo(comandot);
        contadort=0;
    }
    printf("Source Sensor Address=%d \n", sensor_source);
    printf("Temperature=%x\n", spacket_data_tbyte_get(ptrmsg, 4));
    free(ptrmsg);
    /*Revisar las condiciones para enviar requests*/
    if (vib_rcvd==0) {
        /*Si hay algún cambio en los parámetros que involucran a los sensores -> Activar flags*/
        if (((param_changed[0] != 0) && (req_capture==0) && (check_sensor()==0) && (flag_vib==0))
            printf("param_changed[0]=%d\n", param_changed[0]);
            for(j=1; j<(param_changed[0]+1); j++) {
                if (param_changed[j] == 1) {
                    _muestreo=freq * 100;
                }
            }
            if (param_changed[2] == 2) {
                muestras=2;
                potencia=1;
                while (potencia < long_muestra) {
                    muestras=muestras*2;
                    potencia++;
                }
            }
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

```
total_packets=(muestras*SAMPLE_BITS)/PACKET_BITS;
samples_last_packet=muestras-(total_packets*PACKET_SAMPLES);
if(samples_last_packet>0){
    total_packets++;
}
num_seq_max=total_packets-1;
}
activate_flags();
printf("Se activan los flags\n");
}
printf("lista_sensor[\%d].change_param=\%d\n", posicion,lista_sensor[posicion].change_param);
printf("lista_sensor[\%d].extra.capture_acked=\%d\n", posicion,lista_sensor[posicion].extra.capture_acked);
/*Revisar si no se debe pedir algún cambio de parámetro.*/
if(lista_sensor[posicion].change_param==0){
    /*Revisar si se debe enviar petición de captura de fichero de vibración.*/
    if ((req_capture == 1)
        && (receiving == 0)
        && (flag_vib == 0)
        && (lista_sensor[posicion].extra.capture_acked==0))
    {
        if (start_capture == 0){
            timer_set(&capture_timer, TIMER_INTERVAL);
            start_capture=1;
            start_tx=0;
            alarma=1;
            printf("alarma=\%d\n", alarma);
        }
        capturing=1;
        printf("Enviando request de captura...
");
        send_capture_req(src,sensor_source,relay_source, dim_value, dim_len);
        lista_sensor[posicion].extra.vel_reached=velocity;
    }
    /*Revisar si se debe pedir la transmisión del fichero de vibración.*/
    else if (((capturing == 0) || (req_capture == 0))
        && (lista_sensor[posicion].extra.capture_acked!=0)
        && ((flag_vib==0)||(sensor_source==sensor_vib))
    )
    {
        if (start_tx==0){
            timer_set(&vibration_timer, ((t_envio_sensor+MEDIA_TX)*contador_sensores*1000));
            printf("Se activa el timer de petición de fichero de \%d ms\n", vibration_timer.interval);
            start_tx=1;
            start_capture=0;
            receiving=1;
            capturing=0;
        }
        if (flag_vib==0){
            timer_set(&request_vib_timer, ((t_envio_sensor+(t_envio_sensor/2))*1000));
        }
        flag_vib=1;
        sensor_vib=sensor_source;
        pos_vib=posicion;
        relay_vib=relay_source;
        printf("Enviando request de fichero de vibración...
");
```

Page 103
send_vib_req(src,sensor_vib,relay_vib);
}
}

/*Hay algún cambio de parámetro que notificar -> Revisar qué parámetro es y enviar petición.*/
else{
    param=check_param(posicion);
    printf("param=%d\n", param);
    switch(param){
        case 1:
            modify_param_req(src,sensor_source,relay_source,
                FREQ_MUESTREO,freq); /*Petición de cambio de frecuencia de muestreo*/
            break;
        case 2:
            modify_param_req(src,sensor_source,relay_source,
                LONG_MUESTRA,long_muestra); /*Petición de cambio de número de muestras*/
            break;
        case 3:
            modify_param_req(src,sensor_source,relay_source,
                TIME_TEMP,t_envio_sensor); /*Petición de cambio de período de envío de temperaturas*/
            break;
        default:
            lista_sensor[posicion].change_param=0; /*No hay cambios pendientes, desactivar flag*/
            break;
    }
}

/*Para estadísticas!! QUITAR!!*/
else{
    if ((fe = fopen("/home/status/packets_sent", "a+")) != NULL){
        fprintf(fe, "%d %d %ld,%ld\n",sensor_source,temperatura,
            lista_sensor[posicion].tiempo_actual.tv_sec,
            lista_sensor[posicion].tiempo_actual.tv_usec);
        fclose(fe);
    }
}

/*Si es un paquete de vibración*/
case VIB_PACKET:
    if (sensor_source==sensor_vib){
        num_seq=packet_dataobyte_get(ptrmsg,4); /*Número de secuencia del paquete*/
        /*Si es el primer paquete del fichero -> abrir fichero*/
        if ((contadorv == 0)&&(num_seq==0)){
            vib_rcvd=1; /*Flag que está activo mientras se está recibiendo un fichero de vibración.*/
            if ((fd = fopen("/home/status/vib_file_log", "a+")) != NULL){
                gettimeofday(&tv, NULL);
                fprintf(fd, "Apertura: %d %ld,%ld\n",sensor_vib,tv.tv_sec,tv.tv_usec);
                fclose(fd);
            }
            if ((fd = fopen("/home/status/sent_request_reply", "a+")) != NULL){

fprintf(fd, "%x %x %ld,%ld
", VIB_PACKET, sensor_vib, tv.tv_sec, tv.tv_usec);
fclose(fd);

move=0; /*Parámetro que se incrementa a medida que se escribe en los arrays de datos.*/
dimension_vib=spacket_data_obyte_get(ptrmsg, 5); /*Dimensión recibida*/
switch(dimension_vib){
    /*Adjudicar espacio a los punteros de datos de vibración dependiendo de la
dimensión requerida y del número de muestras.*/
    case 0:
        datos_vib[pos_vib].datos_x=(double *)malloc(sizeof(double)*muestras);
        ptr_datos=datos_vib[pos_vib].datos_x;
        cont_dim++; /*Contador de dimensiones recibidas.*/
        break;
    case 1:
        datos_vib[pos_vib].datos_y=(double *)malloc(sizeof(double)*muestras);
        ptr_datos=datos_vib[pos_vib].datos_y;
        cont_dim++; /*Contador de dimensiones recibidas.*/
        break;
    case 2:
        datos_vib[pos_vib].datos_z=(double *)malloc(sizeof(double)*muestras);
        ptr_datos=datos_vib[pos_vib].datos_z;
        cont_dim++; /*Contador de dimensiones recibidas.*/
        break;
    default:
        /*Error de dimension*/
        printf("Error de dimension %d\n", dimension_vib);
        error_flag=2;
        goto reject;
        break;
}
sprintf(nombre_v,"%sV_%s_%d_%ld.%ld", DIR_LISTEN, gwid, sensor_vib,
        lista_sensor[pos_vib].extra.capture_time.tv_sec,
        lista_sensor[pos_vib].extra.capture_time.tv_usec); /*Nombre del fichero de
vibración.*/
if ((ficherov = fopen(nombre_v, "w" )) == NULL){ /*Crear fichero para escribir.*/
    perror("Error abriendo el fichero de vibracion\n");
    error_flag=2;
    goto reject;
}
sprintf(comandov,"/etc/alstom/lock_datos r %s",nombre_v);
timer_set(&vib_file_error,TIMER_INTERVAL); /*Timer de errores en recepción de
fichero*/

/*Copiar el timestamp de captura para las alarmas de vibración (do_fft)*/
datos_vib[pos_vib].timestamp.tv_sec=lista_sensor[pos_vib].extra.capture_time.tv_sec;
datos_vib[pos_vib].timestamp.tv_usec=lista_sensor[pos_vib].extra.capture_time.tv_usec;

/*Escribir cabecera de fichero*/
fwrite(&sensor_vib,sizeof(uint16_t),1,ficherov); /*Direccion del sensor*/
fwrite(&dimension_vib,sizeof(uint8_t),1,ficherov); /*Dimensión (x,y,z) de los datos*/
fwrite(&f_muestreo,sizeof(uint32_t),1,ficherov); /*Frecuencia de Muestreo*/
fwrite(&muestras,sizeof(uint32_t),1,ficherov); /*Número de muestras*/
fwrite(&lista_sensor[pos_vib].extra.capture_time,
        sizeof(struct timeval),1,ficherov); /*Fecha y Hora de la Captura*/
fwrite(&lista_sensor[pos_vib].extra.vel_reached,
        sizeof(float),1,ficherov); /*Velocidad del tren cuando se captura*/
}

/*Extraer datos del paquete y llenar el array (apuntado por ptr_datos) para la fft*/
if ((vib_rcvd==1)){ //&&(spacket_data_obyte_get(ptrmsg,5)==dimension_vib)){

samples_limit=PACKET_SAMPLES; /*Número de muestras a leer del payload del paquete.*/

if ((num_seq == num_seq_max) && (samples_last_packet>0)){
    /*El último paquete de vibración tiene menos muestras que PACKET_SAMPLES.*/
    samples_limit=samples_last_packet;
}

for (j=0;j<samples_limit;j++){
    index=48+(SAMPLE_BITS*j); /*Los datos empiezan a partir del 6to byte = 48vo bit*/
    vibraciones[j]=spacket_data_get(ptrmsg,index);
    *ptr_datos+move=(double)vibraciones[j];
    move++;
    if (vibraciones[j] != 0){
        fail=1;
    }
}

/*Escribir datos en el fichero*/
fwrite(vibraciones,sizeof(uint16_t),samples_limit,ficherov); /*Todas las Muestras*/
/*Incrementar contador de paquetes de vibracion*/
contadorv++;
printf("Se recibió el paquete de vibracion %d, del sensor %x, contadorv=%d
",num_seq,sensor_source, contadorv);
}
break;

/*Si es un paquete de control -> Confirmación de alguna petición.*/
case CONTROL_PACKET:
    value_field=spacket_data_obyte_get(ptrmsg,5); /*Tipo de dato confirmado.*/
    if ((posicion=encontrar_direccion(sensor_source)) != -1)
    {
        switch(value_field){
        case CAPTURE_VIB: /*Confirmación de Captura*/
            if(lista_sensor[posicion].extra.capture_acked == 0){
                printf("Se recibe confirmación de captura...\n");
                lista_sensor[posicion].extra.capture_acked=1; /*Activar flag de captura.*/
                gettimeofday(&lista_sensor[posicion].extra.capture_time,NULL); /*Timestamp de captura.*/
            }
            break;
        case FREQ_MUESTREO: /*Confirmación de cambio de frecuencia de muestreo*/
            changes[posicion].f=0;
            break;
        case LONG_MUESTRA:
            changes[posicion].l=0; /*Confirmación de cambio de número de muestras*/
            break;
        }
case TIME_TEMP:
    changes[posicion].t=0; /*Confirmación de cambio de período de envío de
temperaturas.*/
    break;
default:
    break;
}

if (value_field != CAPTURE_VIB){ /*La confirmación ha sido de algún parámetro*/
    if((param=check_param(posicion))==0){  /*Ha confirmado todos los cambios de
parámetros*/
        lista_sensor[posicion].change_param=0; /*Desactivar flag*/
    }
    else{ /*Faltan cambios por confirmar*/
        switch(param){
        case 1:
            modify_param_req(src,sensor_source,relay_source,
            FREQ_MUESTREO,freq); /*Petición de cambio de
frecuencia de muestreo*/
            break;
        case 2:
            modify_param_req(src,sensor_source,relay_source,
            LONG_MUESTRA,long_muestra);/*Petición de cambio de
número de muestras*/
            break;
        case 3:
            modify_param_req(src,sensor_source,relay_source,
            TIME_TEMP,t_envio_sensor); /*Petición de cambio de
período de envío de
             *temperaturas*/
            break;
        default:
        lista_sensor[posicion].change_param=0; /*No hay cambios pendientes,
             desactivar flag*/
            break;
        }
    }

    default: /*Si el valor del AM no es alguno de los definidos, se vuelve a escuchar el puerto.*/
    break;
} /*End switch(am_type)*/

/* vib_rcvd=1 implica que se está recibiendo un fichero de vibracion.
* num_seq_max es el número de secuencia del último paquete del fichero.
* vib_file_error es un timer para cerrar el fichero en caso de algun error en la transmisión.
* Si se recibe el último paquete o hay algun error -> cerrar fichero e inicializar variables.
*/
always:
if ((vib_rcvd == 1) &&
    (((num_seq==num_seq_max)&(&(contadorv==total_packets))||(timer_expired(&vib_file_error)))))
{
    if (ficherov != NULL){
        fclose(ficherov);
        ficherov=NULL;
    }
}
printf("Fichero de vibracion cerrado\n");
gmtimeofday(&tv, NULL);
}
if ((num_seq==num_seq_max) && (contadorv==total_packets)){
if ((fd = fopen("/home/status/vib_file_log", "a+")) != NULL){
  if (fail == 1){
    fprintf(fd, "Cierre FAIL_DATA: %d %s\n",sensor_vib,ctime(&tv.tv_sec));
  } else{
    fprintf(fd, "Cierre OK: %d %ld,%ld\n",sensor_vib, tv.tv_sec, tv.tv_usec);
  }
  fclose(fd);
}
  system(comandov);
} else{
  if ((fd = fopen("/home/status/vib_file_log", "a+")) != NULL){
    fprintf(fd, "Cierre FAIL: %d %ld,%ld\n",sensor_vib, tv.tv_sec, tv.tv_usec);
    fclose(fd);
  }
  if (nombre_v[0] != 0){
    remove(nombre_v);
  }
  error_flag=1;
  switch (dimension_vib){
  case 0:
    free(datos_vib[pos_vib].datos_x);
    break;
  case 1:
    free(datos_vib[pos_vib].datos_y);
    break;
  case 2:
    free(datos_vib[pos_vib].datos_z);
    break;
  default:
    break;
  }
}
}
if (error_flag!=0){ /*Pedir el mismo fichero de vibración hasta 3 veces en caso de error*/
  if ((lista_sensor[pos_vib].extra.capture_acked+=3) == 3) {
    lista_sensor[pos_vib].extra.capture_acked=0;
  }
}
else { /*Si no hay error, se puede pedir una nueva captura. (Desactivar flag)*/
  lista_sensor[pos_vib].extra.capture_acked=0;
}
The thread for checking temperature alarms is:

```
#include "config.h"
#include "th_def.h"

/*Variables globales*/
extern uint16_t id_sensor[NUM_SENSORES];
extern sensor_t lista_sensor[NUM_SENSORES];
extern fft_vib datos_vib[NUM_SENSORES];
extern uint16_t temp_max;
extern uint16_t dif_temp_max;
extern double time_btw_temp;
extern uint8_t contador_sensores;

int direccion_eje(uint16_t direccion)
/*Devuelve la posición en que se encuentra la dirección del sensor en el arreglo*/
{
    int i;
    for(i=0;i<NUM_SENSORES;i++){
        if (id_sensor[i]==direccion){
            return i;
        }
    }

    return -1;
```
void *registrar(void *argumento)
{
    int i, pos = -1;
    uint8_t alarma_i, alarma_pos, alarma_eje, alarma;
    uint8_t checked[NUM_SENSORES];
    double dif_tiempo;
    uint16_t sensor_eje = 0, dif_temperatura;
    uint16_t coche_bogie;
    uint8_t sensor_id;
    struct timeval tiempo;
    char nombre[50];
    FILE *falarm = NULL;

    while (1)
    {
        for (i = 0; i < NUM_SENSORES; i++)
            checked[i] = 0;

        gettimeofday(&tiempo, NULL);
        sprintf(nombre, "%stalarm/T_%ld.%ld", DIR_ALARM, tiempo.tv_sec, tiempo.tv_usec);

        for (i = 0; i < NUM_SENSORES; i++)
        {
            alarma = 0;
            alarma_i = 0;
            alarma_pos = 0;
            alarma_eje = 0;

            if ((checked[i] == 0) && (id_sensor[i] != 0))
            {
                /*printf("id_sensor[%d]=%x\n",i,id_sensor[i]);*/
                sensor_id = 0x000F & id_sensor[i];
                coche_bogie = 0x0FF0 & id_sensor[i];

                switch (sensor_id)
                {
                    case 0x01:
                        sensor_eje = coche_bogie + 0x0002;
                        break;
                    case 0x02:
                        sensor_eje = coche_bogie + 0x0001;
                        break;
                    case 0x03:
                        sensor_eje = coche_bogie + 0x0004;
                        break;
                    case 0x04:
                        sensor_eje = coche_bogie + 0x0003;
                        break;
                    default:
                        sensor_eje = 0x000F;
                        break;
                }
            }

            gettimeofday(&tiempo, NULL);
            if (sensor_eje != 0x000F)
            {
                pos = direccion_eje(sensor_eje);
                if (pos != -1)
                {
                    /*printf("Existe el sensor_eje\n");*/
                    if (lista_sensor[pos].temperatura_actual >=
                        lista_sensor[pos].temperatura_actual -
                        dif_temperatura)
                    {
                        /*printf("Existe el sensor_eje\n");*/
                        if (lista_sensor[pos].temperatura_actual -
                            dif_temperatura > lista_sensor[pos].temperatura_actual)
                        {
                            /*printf("Existe el sensor_eje\n");*/
                        }
                    }
                }
            }
        }
    }
}
lista_sensor[pos].temperatura_actual;

} else{
    dif_temperatura=lista_sensor[pos].temperatura_actual-
    lista_sensor[i].temperatura_actual;
}

dif_tiempo=difftime(tiempo.tv_sec, lista_sensor[pos].tiempo_actual.tv_sec);

if (dif_temperatura >= dif_temp_max){
    alarma_eje=2;
}

if (lista_sensor[pos].temperatura_actual >= temp_max){
    alarma_pos=1;
}

if (dif_tiempo >= time_btw_temp){
    alarma_pos=3;
    id_sensor[pos]=0;
    contador_sensores--;
    lista_sensor[pos].extra.capture_acked=0;
    datos_vib[pos].ready_to_process=0;
}

if ((alarma_pos != 0) || (alarma_eje != 0)){
    alarma=1;
}
    checked[pos]=1;
}
}

dif_tiempo=difftime(tiempo.tv_sec, lista_sensor[i].tiempo_actual.tv_sec);

if (lista_sensor[i].temperatura_actual >= temp_max){
    alarma_i=1;
}

if (dif_tiempo >= time_btw_temp){
    alarma_i=3;
    id_sensor[i]=0;
    contador_sensores--;
    lista_sensor[i].extra.capture_acked=0;
    datos_vib[i].ready_to_process=0;
}

if (alarma_i != 0){
    alarma=1;
}

checked[i]=1;

if (alarma != 0){
    falarm = fopen(nombre, "a+b");
    if(falarm == NULL) falarm = fopen(nombre, "w+b");
    if (alarma_i != 0){
        fwrite(&alarma_i,sizeof(uint8_t),1,falarm);
        fwrite(&lista_sensor[i].direccion,sizeof(uint16_t),1,falarm);
        if (alarma_i != 3){
            fwrite(&lista_sensor[i].direccion,sizeof(uint16_t),1,falarm);
        }
    }
}
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

```c
fwrite(&lista_sensor[i].tiempo_actual,sizeof(struct timeval),1,falarm);
fwrite(&lista_sensor[i].temperatura_actual,sizeof(uint16_t),1,falarm);
}
else{
fwrite(&tiempo,sizeof(struct timeval),1,falarm);
}
}
if (pos != -1){
if (alarma_pos != 0){
fwrite(&alarma_pos,sizeof(uint8_t),1,falarm);
fwrite(&lista_sensor[pos].direccion,sizeof(uint16_t),1,falarm);
if (alarma_pos != 3){
fwrite(&lista_sensor[pos].tiempo_actual,sizeof(struct timeval),1,falarm);
fwrite(&lista_sensor[pos].temperatura_actual,sizeof(uint16_t),1,falarm);
}
else{
fwrite(&tiempo,sizeof(struct timeval),1,falarm);
}
}
if (alarma_eje != 0){
fwrite(&alarma_eje,sizeof(uint8_t),1,falarm);
fwrite(&lista_sensor[i].direccion,sizeof(uint16_t),1,falarm);
fwrite(&lista_sensor[i].tiempo_actual,sizeof(struct timeval),1,falarm);
fwrite(&dif_temperatura,sizeof(uint16_t),1,falarm);
fwrite(&lista_sensor[pos].direccion,sizeof(uint16_t),1,falarm);
fwrite(&lista_sensor[pos].tiempo_actual,sizeof(struct timeval),1,falarm);
}
fclose(falarm);
}
if (falarm != NULL){
    system("/etc/alstom/comprimir_alarm.sh 1");
    faalarm=NULL;
    sleep(5);
}
return NULL;
```

The thread for checking vibration alarms is:

```c
#include <math.h>
#include "/root/openwrt/trunk/staging_dir/target-ameb_uClibc-0.9.30/usr/include/fftw3.h"
#include "config.h"
#include "th_def.h"

extern uint16_t id_sensor[NUM_SENSORES];
extern fft_vib datos_vib[NUM_SENSORES];
extern uint32_t muestras;
extern double amplitudes[5];
extern uint32_t armonicos[5];
extern uint32_t t_f_muestreo;
extern float velocity;
extern uint8_t alarma;
```
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

```c
void *do_fft(void *argumento)
{
    int aux, i, j, fs;
    double *in;
    double *ptr;
    double sumas=0, media=0;
    int N=(muestras/2)+1;
    uint8_t dim;
    fftw_complex *out;
    fftw_plan p;
    long double amp;
    struct timeval tiempo;
    char nombre[50];
    FILE *fa;

    while(1){
        fs=(f_muestreo/muestras);
        for(i=0; i NUM_SENSORES; i++){
            if(datos_vib[i].ready_to_process==1){
                for(dim=0; dim<3; dim++){
                    switch(dim){
                        case 0:
                            ptr=datos_vib[i].datos_x;
                            break;
                        case 1:
                            ptr=datos_vib[i].datos_y;
                            break;
                        case 2:
                            ptr=datos_vib[i].datos_z;
                            break;
                        default:
                            goto end;
                            break;
                    }
                    if (ptr != NULL){
                        gettimeofday(&tiempo,NULL);
                        sprintf(nombre, "%svalarm/V_%d_%ld.%ld",
                                DIR_ALARM, id_sensor[i], tiempo.tv_sec, tiempo.tv_usec);
                        suma=0;
                        media=0;
                        in = (double*)malloc(sizeof(double) * muestras);
                        out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
                        for (aux=0; aux<muestras; aux++)
                        {
                            in[aux] = *(ptr+aux);
                            suma=suma+in[aux];
                        }
                        media=suma/muestras;
```
for (aux=0; aux < (int)muestras; aux++)
{
    in[aux] = ((in[aux]-media) * (450.0/4096.0));
    fprintf(fa,"in[%d]=%.1lf\n",aux,in[aux]);
}

p = fftw_plan_dft_r2c_1d(muestras, in, out, FFTW_ESTIMATE);
fftw_execute(p);

for (j=0;j<5;j++) //for (aux=0;aux<(int)N;aux++)
{
    if ((armonicos[j] != 0) || (amplitudes[j] != 0)){
        aux=(armonicos[j]/fs);
        amp=2*sqrt((out[aux][0]*out[aux][0])+(out[aux][1]*out[aux][1]))/muestras;
        if (amp >= amplitudes[j]){    
            alarma=1;
            fa = fopen(nombre, "a+b");
            fwrite(&id_sensor[i],sizeof(uint16_t),1,fa);
            fwrite(&datos_vib[i].timestamp,sizeof(struct timeval),1,fa);
            fwrite(&armonicos[j],sizeof(uint32_t),1,fa);
            fwrite(&amp,sizeof(double),1,fa);
            fclose(fa);
        }
    }
}

fftw_destroy_plan(p);
free(in);
fftw_free(out);

switch(dim){
    case 0:
        free(datos_vib[i].datos_x);
        break;
    case 1:
        free(datos_vib[i].datos_y);
        break;
    case 2:
        free(datos_vib[i].datos_z);
        break;
    default:
        break;
    }
}

/*End if(ptr!=NULL)*/
/*End for(dim...)*/

end:
datos_vib[i].ready_to_process=0;
/*End if(datos_vib[i].ready_to_process==1)*/

if (alarma == 1){
    system("/etc/alstom/comprimir_alarm.sh 2");
    alarma=0;
}
/*End for(i=0;i<NUM_SENSORES;i++)*/
if (alarma){
    system("/etc/alstom/comprimir_alarm.sh 2");
}
sleep(5);
} /*End while(1)*/
return NULL;
} /*End void *do_fft(void *argumento)*/

The thread used to check modification in the configuration file is:

#include "config.h"
#include "myutils.h"
#include "th_def.h"

/*Variables globales externas*/
extern sensor_t lista_sensor[NUM_SENSORES];
extern uint16_t temp_max;
extern uint16_t dif_temp_max;
extern double time_btw_temp;
extern char gwid[20];
extern uint8_t freq;/*Hz*/
extern uint8_t long_muestra;/* (2^long_muestra) muestras*/
extern uint8_t param_changed[(PARAMETROS+1)];
extern uint8_t_t_envio_sensor;
extern uint8_t init_flag;
extern dim_vib real;
extern uint8_t dim_value[3];
extern int dim_len;
extern uint8_t req_capture;

uint8_t num_vel=2;
limits_t vel[10];

int change_dim()
{
    int i;
    for(i=0;i<NUM_SENSORES;i++)
    {
        if(lista_sensor[i].extra.capture_acked==1)
            return 1;
    }
    return 0;
}

void *check_config(void *argumento)
{
    char salida[50];
    int i, len, count, flag=0;
    char *ptr;
    for (i=0;i<10;i++)
    {
        vel[i].max=0;
        vel[i].min=0;
    }
    while(1){
        count=0;
        if (read_conf("TEMP_MAX",salida,NULL) == 0){
if (atoi(salida) != temp_max)
    temp_max=atoi(salida);
}

if (read_conf("DIF_MAX",salida,NULL) == 0){
    if (atoi(salida) != dif_temp_max)
        dif_temp_max=atoi(salida);
}

if (read_conf("TIME_BTW_TEMP",salida,NULL) == 0){
    if (atof(salida) != time_btw_temp)
        time_btw_temp=atof(salida);
}

if (read_conf("GW_ID",salida,NULL) == 0){
    if (strcmp(salida,gwid) != 0){
        if (strlen(salida)>strlen(gwid))
            len=strlen(salida);
        else
            len=strlen(gwid);
        for (i=0;i<len;i++)
            gwid[i]=salida[i];
    }
}

if (read_conf("NUM_VEL",salida,FFT_PARAM_FILE) == 0){
    if ((atoi(salida) != num_vel) && (atoi(salida) != 0))
        num_vel=atoi(salida);
}

if (read_conf("VELOCIDADES",salida,FFT_PARAM_FILE) == 0){
    ptr = strtok( salida, ",");
    if (atof(ptr) != vel[0].max)
        vel[0].max=atof(ptr);
    ptr = strtok( NULL, ",");
    if (atof(ptr) != vel[0].min)
        vel[0].min=atof(ptr);
    i=1;
    while( (ptr = strtok( NULL, "," )) != NULL ){
        if (atof(ptr) != vel[i].max)
            vel[i].max=atof(ptr);
        ptr = strtok( NULL, "," );
        if (atof(ptr) != vel[i].min)
            vel[i].min=atof(ptr);
        i++;
    }
}

if (read_conf("FREQ_MUESTREO",salida,NULL) == 0){
    if (atoi(salida) != freq){
        freq=atoi(salida);
        count++;
        param_changed[count]=1;
    }
}

if (read_conf("MUESTRAS",salida,NULL) == 0){
    if (atoi(salida) != long_muestra){
        long_muestra=atoi(salida);
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

```
count++; 
param_changed[count]=2; 
}
}
if (read_conf("T_DATOS_SENSOR",salida,NULL) == 0){
    if (atoi(salida) != t_envio_sensor){
        t_envio_sensor=atoi(salida); 
        printf("t_envio_sensor=%d\n",t_envio_sensor); 
        count++; 
        param_changed[count]=3; 
    }
}
if (count != 0)
    param_changed[0]=count; 
if (read_conf("DIMENSION",salida,NULL) == 0){
    ptr = strtok( salida, ",\" "\" "); 
    if (atoi(ptr) != real.x){
        real.x=atoi(ptr); 
        flag=1; 
    }
    ptr = strtok( NULL, ",\" "\" "); 
    if ((ptr != NULL) && (atoi(ptr) != real.y)){
        real.y=atoi(ptr); 
        flag=1; 
    }
    ptr = strtok( NULL, ",\" "\" "); 
    if ((ptr != NULL) && (atoi(ptr) != real.z)){
        real.z=atoi(ptr); 
        flag=1; 
    }
}
if ((flag==1)&&(req_capture==0)&&(change_dim()==0)){
    len=0;
    for(i=0;i<3;i++){
        if (real.dimension[i] != 7){
            dim_value[len]=real.dimension[i]; 
            len++; 
        }
    }
    dim_len=len; 
    flag=0; 
}
init_flag=1; 
/*printf("Hilo check_config dormido por 10 segundos\n");*/ 
sleep(10); 
} return NULL; 
```

The final thread used for reading the FIFO shared with the GPS data collection process is:
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <stdint.h>
#include <unistd.h>
#include <termios.h>
#include "th_def.h"
#include "myutils.h"

#define FIFO_GPS_VEL "/home/fifos/gps_vel"

/*Variables globales*/
extern uint8_t req_capture;
extern float velocity;
extern float amplitudes[5];
extern uint32_t armonicos[5];
extern uint8_t num_vel;
extern limits_t vel[10];
extern uint8_t init_flag;

void *read_fifo(void *argumento)
{
    int fd;
    pthread_mutex_t mutex_vel;
    float previous_max=180, previous_min=180;
    uint8_t i, j, capture_flag=0;
    char entrada[6];
    char salida[50];
    char *ptr;
    while(1){
        if (init_flag != 0){
            capture_flag=0;
            fd=open(FIFO_GPS_VEL,O_RDONLY);
            if(fd>=0){
                pthread_mutex_lock(&mutex_vel);
                read(fd,&velocity,sizeof(float));
                pthread_mutex_unlock(&mutex_vel);
                close(fd);
                for (i=0;i<num_vel;i++){
                    if ((velocity>=vel[i].min) && (velocity < vel[i].max)){
                        if ((vel[i].min != previous_min) || (vel[i].max != previous_max)){
                            previous_min=vel[i].min;
                            previous_max=vel[i].max;
                            printf("num_vel=%d, vel[%d].min=%f, vel[%d].max=%f\n", num_vel, i, vel[i].min, i, vel[i].max);
                            sprintf(entrada, "%d_FRQ",i);
                            if (read_conf(entrada,salida,FFT_PARAM_FILE) = = 0){
                                ptr = strtok( salida, ", " );
                                if (atoi(ptr) != armonicos[0])
                                    armonicos[0]=atoi(ptr);
                                j=1;
                                while( (ptr = strtok(NULL, ", " )) != NULL ){
                                    if (atoi(ptr) != armonicos[j])
                                        armonicos[j]=atoi(ptr);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Page 118
Development of a Gateway to control and connect an itinerant Wireless Sensor Network with a central system via Internet

```c
j++;
}
printf(entrada, "%d_AMP", i);
if (read_conf(entrada,salida,FFT_PARAM_FILE) == 0){
    ptr = strtok( salida, ".");
    if (atof(ptr) != amplitudes[0])
        amplitudes[0]=atof(ptr);
    j=1;
    while((ptr = strtok( NULL, "," )) != NULL ){
        if (atof(ptr) != amplitudes[j])
            amplitudes[j]=atof(ptr);
        j++;
    }
    capture_flag=1;
} break;
}
if (capture_flag==1){
    req_capture=1;
} else{
    req_capture=0;
}
}
/*/Fin del while(1)*/
return NULL;
```