ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Design of an IoT based multi-channel temperature monitoring system

In collaboration with IIHE

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil en informatique à finalité spécialisée

Daniel Gómez de Gracia

Promoteur
Prof. Frédéric Robert

Co-promoteur
Prof. Barbara Clerbaux

Superviseurs
Yifan Yang, Pierre-Alexandre Petitjean

Service
IIHE

Année académique
2020 - 2021

ULB UNIVERSITÉ LIBRE DE BRUXELLES

ULB UNIVERSITÉ LIBRE DE BRUXELLES

Exemplaire à apposer sur le mémoire ou travail de fin d'études,
au verso de la première page de couverture.

Fait en deux exemplaires, **Bruxelles, le** 28/5/2021
Signature

Réservé au secrétariat :	**Mémoire réussi***	**OUI**
		NON

CONSULTATION DU MEMOIRE/TRAVAIL DE FIN D'ETUDES

Je soussigné

NOM : Gomez de Gracia

PRENOM : Daniel

TITRE du travail : Design of an IoT-based multi-channel temperature monitoring system

AUTORISE*

REFUSE*

la consultation du présent mémoire/travail de fin d'études par les utilisateurs des bibliothèques de l'Université libre de Bruxelles.

Si la consultation est autorisée, le soussigné concède par la présente à l'Université libre de Bruxelles, pour toute la durée légale de protection de l'œuvre, une licence gratuite et non exclusive de reproduction et de communication au public de son œuvre précisée ci-dessus, sur supports graphiques ou électroniques, afin d'en permettre la consultation par les utilisateurs des bibliothèques de l'ULB et d'autres institutions dans les limites du prêt inter-bibliothèques.

Formulaire version 04/2015

* *Biffer la mention inutile*

* *Biffer la mention inutile*

Abstract - English

In the scope of the Jiangmen Underground Neutrino Observatory (JUNO) project, 6 back-end card (BEC) mezzanines connected to one BEC base board are in charge of compensating the attenuated incoming data from 48 front-end channels over 48 100-meterlong Ethernet cables. Each of the mezzanines has 16 equalizers that may be subject to overheating. It is important to monitor their temperature in real time. However, collecting data from a relatively large (1080) number of mezzanines is not a trivial task. In this work, we propose a solution based on Wi-Fi mesh, and we provide the technical details and test results to validate our implementation.

Abstract - Français

Dans le cadre du projet JUNO (Jiangmen Underground Neutrino Observatory), 6 cartes back-end mezzanines, qui sont connectées a une carte back-end (BEC) pricipale, sont chargées de compenser les données atténuées qui proviennent des 48 canaux frontaux à travers 48 câbles Ethernet de 100 mètres de long. Chaque mezzanine possède 16 égaliseurs qui risquent de surchauffer en cas de mauvaise ventilation. Il est important de surveiller leur température en temps réel. Néanmoins, rassembler des données venant d'un nombre important de mezzanines (1080 en total) n'est pas une tache triviale. Dans ce travail, nous proposons une solution pour ce problème basée sur un réseau de type Wi-Fi mesh. On y fournit tant les détails techniques comme les expériences et résultats obtenu dans le but de valider notre implémentation.

Acknowledgements

Foremost, I would like to express my sincere gratitude to Prof. Robert and Prof. Clerbaux for their guidance and support during the elaboration of my master thesis.
Besides my promoters, I would like to thank my supervisors Yifan Yang and Pierre-Alexandre Petitjean. Due to the restrictions in place during Covid19, it was an additional challenge to carry out experiments, order material, and discuss about the state of the work. Without their help and guidance, this thesis would have never happened.
My grateful thanks are also extended to Marta Colomer for her comments on my work.
I would like extend my thanks to Fred, my arm wrestling coach who taught me to never give up and motivated me all along my studies.
Finally, I must express my very profound gratitude to my parents and my family for providing me with unfailing support and continuous encouragement during my years study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Daniel Gómez de Gracia

Contents

1 Definition of the problem **3**
1.1 JUNO . . . 3
1.2 JUNO electronics . . . 5
1.2.1 Wet electronics . . . 5
1.2.2 Dry electronics . . . 5
1.2.3 Back-End card system . . . 7
1.3 Temperature Control . . . 9

2 Requirements and choices **10**
2.1 Hardware choices . . . 10
2.1.1 ESP32 . . . 10
2.1.2 TMP100 . . . 13
2.2 Software choices . . . 15
2.2.1 Data transmission . . . 15
2.2.2 Configuration . . . 15

3 Engineering Analysis and Development **16**
3.1 Temperature retrieval . . . 16
3.1.1 I^2C Bus . . . 16
3.1.2 SMBus . . . 17
3.1.3 Configuration of the temperature sensors . . . 18
3.1.4 Reading of the values . . . 25
3.1.5 Programming . . . 26
3.2 Retrieval to the computer . . . 27
3.2.1 Wi-Fi architecture . . . 27
3.2.2 Implementation . . . 29
3.2.3 Central computer communication . . . 35

4 Validation of the solution **43**
4.1 Temperature monitoring test . . . 43
4.1.1 Introduction . . . 43
4.1.2 Continuous monitoring test . . . 44
4.1.3 Fan configuration test . . . 46
4.2 Maximum message rate test . . . 47
4.3 Range tests . . . 49
4.3.1 Wall Test . . . 49
4.3.2 Bridge node to Mesh node Test . . . 50
4.3.3 Mesh node to Mesh node range . . . 50
4.3.4 Results discussion . . . 51

4.4 Small scale test 51
4.4.1 Overview 51
4.4.2 Set-up 51
4.4.3 Real time monitoring test 53
4.4.4 Performance analysis 53
4.4.5 Final thoughts on the test results 58

5 Summary **60**

A Table for the different tests **62**
A.1 Test 1 - Assessment test 62
A.2 Test 2 - Increased bridge test 63
A.3 Test 3 - Memory logging disconnections 64
A.4 Test 4 - 2 minute reboot time test 65

Introduction

The Jiangmen Underground Neutrino Observatory (JUNO) is a multipurpose neutrino detector. Its main goal is to determine the neutrino mass hierarchy and to precisely measure some of the neutrino oscillation parameters, using as a source electronic anti-neutrinos coming from the Yangjiang and Taishan Nuclear Power Plants. The detector will also allow physicists to observe potential supernova neutrinos, to study the atmospheric and solar neutrinos, geo-neutrinos, and to perform exotic searches in the neutrino sector.

The JUNO detector is located at 700 m underground and consists of 20 ktons of liquid scintillator contained in a 35 m diameter acrylic sphere, instrumented by 18000 20-inch photomultiplier tubes (PMTs), and 25600 3-inch small PMTs, with a cathode coverage of about 77%. A muon veto detector consisting of a 20 ktons ultrapure water Cerenkov pool is placed around the central detector and is instrumented by 2000 20-inch PMTs. An electronic anti-neutrino interaction in the detector will be identified by observing the inverse beta decay (IBD) signal in the liquid scintillator, relying on the twofold coincidence between the prompt signal given by the positron ionization and annihilation and the delayed signal given by the neutron capture on hydrogen.

JUNO uses a complex electronic system. This system starts at the in-water electronics, where the large PMTs, the small PMTs, and the veto PMTs send hit information by groups of three to the Global Control Units (GCUs). The GCUs digitize and store the received signals in a large local memory under the control of a Field-Programmable Gate Array (FPGA). They wait for the trigger decision and they send out event data as well as trigger requests to the outside-water system. On the surface, a back-end card (BEC) is used as a concentrator to collect trigger requests from 48 GCUs. Each of the incoming trigger request signals passes through an equalizer to compensate for the attenuation due to the long cables. There are 180 BECs in the system, each containing 6 Mezzanines equipped with 16 equalizers. This makes up for a total of 17280 equalizers. Measuring the temperatures of the multiple equalizers placed on all the BECs is necessary to make sure that the equalizers perform stably and don't overheat.

The main objective of this thesis is to monitor the equalizer's temperatures remotely. To measure the overall temperature of the 96 equalizers present on six Mezzanines, three temperature sensors are placed on the BEC. Each of them is 8 mm apart from a Mezzanine. This placement allows the sensors to measure the temperature of the hot air present above the equalizers. A development board is used to read out the sensors' values through an I^2C bus. However, remotely monitoring these values is not a trivial task.

One option would be to use cables to link the development boards to a central computer. However, carrying this out would need the use of at least 180 additional cables,

which ought to be avoided. This thesis proposes to implement a retrieval system based on Wi-Fi. Installing a router in the JUNO electronics system would allow the development boards to connect to a server computer and transmit the data they read from the sensors. Due to the possibility of having diminished performance caused by channel overlapping when too many devices (180 in this case) are connected to a same Wi-Fi network, we propose designing a Wi-Fi mesh network consisting of the 180 ESP32 boards in order to link them all to the central computer.

This work is structured in the following manner. The first chapter introduces JUNO in more detail, and clearly defines the problems that need to be solved. The second chapter gives an overview of the desired solution, along with some hardware and software choices that were made. The third chapter discusses in detail the implementation of the solution. The fourth chapter shows the experiments that were designed in order to test and evaluate the performance of the proposed solution.

Chapter 1

Definition of the problem

This chapter defines the problem that this thesis intends to solve. It sets the context of the problem by introducing what JUNO is, and how its electronic system works. We describe the main electronic components of the system, and present with more detail the ones that interest us the most. We end by identifying a potential issue in a piece of hardware, and briefly comment on the strategy that will be used to deal with it.

1.1 JUNO

The Jiangmen Underground Neutrino Observatory (JUNO) is a multipurpose neutrino experiment designed to determine the neutrino mass hierarchy and precisely measure some of the oscillation parameters by detecting electronic anti-neutrinos coming from the reactors at Yangjiang and Taishan Nuclear Power Plants (see figure 1.1). The required energy resolution to discriminate between the normal and inverted neutrino mass hierarchies at a 3-4 sigma for about 6 years of data taking is 3% at an energy of 1 MeV, and the requirement on the energy scale is better than 1% [1].

Figure 1.1: Location of JUNO and nuclear reactors.

JUNO is located in Kaiping, Jiangmen, in Southern China, at about 53 km from the Yangjiang and Taishan nuclear power plants, both of which are under construction. The planned total thermal power of these reactors is 36 GW. The detector will be located deep underground through a tunnel, and the total overburden will be 700 m of rock. Its construction, which started in 2014 and is scheduled to complete in 2022, includes the

building of a tunnel, an underground experiment hall, a water pool, a central detector, a muon tracking detector and some ancillary facilities.

The central detector is filled with 20 kilotons of LAB-based liquid scintillator. This liquid is made of 3 components. The solvent is Linear Alkyl Benzene, the fluor is 2.5-diphenyloxazole, and the wavelength shifter is p-bis-(o-methylstyryl)-benzene (bis-MSB)[2]. When neutrinos go through the detector, a very small part of them interact with the liquid scintillator, producing scintillation light, which can be detected by the 18000 large (20-inch) photomultiplier tubes (PMTs) and 25600 small (3-inch) PMTs. These tubes, which surround the central detector, transmit an analog signal that is fed into a Global Control Unit (GCU). The GCU is a multi purpose card. On one side, it is used to receive the hit information from the PMTs and to digitize them using a high speed ADU (Analog to Digital converter Unit). These are stored in a large local memory under the control of an FPGA (Field Programmable Gate Array). On the other side, it sends trigger requests to the Back-End Card (BEC). These signals are then sent through the "synchronous link" (which is a reduced implementation of the IEEE 1588 Precision Time Protocol [3], an Ethernet protocol used to precisely synchronize distributed clocks to sub-microsecond resolution) to the 180 BECs. The BECs are used as concentrators. They receive the trigger requests and treat them by passing the signals through equalizers. They are equipped with an FPGA Mezzanine card (called Trigger Timing Interface Mezzanine (TTIM)), which collects the trigger requests, adds them, and sends them to a Reorganize and Multiplex Unit (RMU). The RMUs then send these requests through an optical fiber to the Central Trigger Unit (CTU). BECs also receive the trigger decision from the CTU, and pass them to the GCUs to trigger data acquisition [4].

Figure 1.2 shows the Vessel, which is the structure that contains the liquid scintillator, the photomultiplier tubes, and other elements needed to perform the detection of neutrino interactions and the conversion of the observed signals into wavelengths.

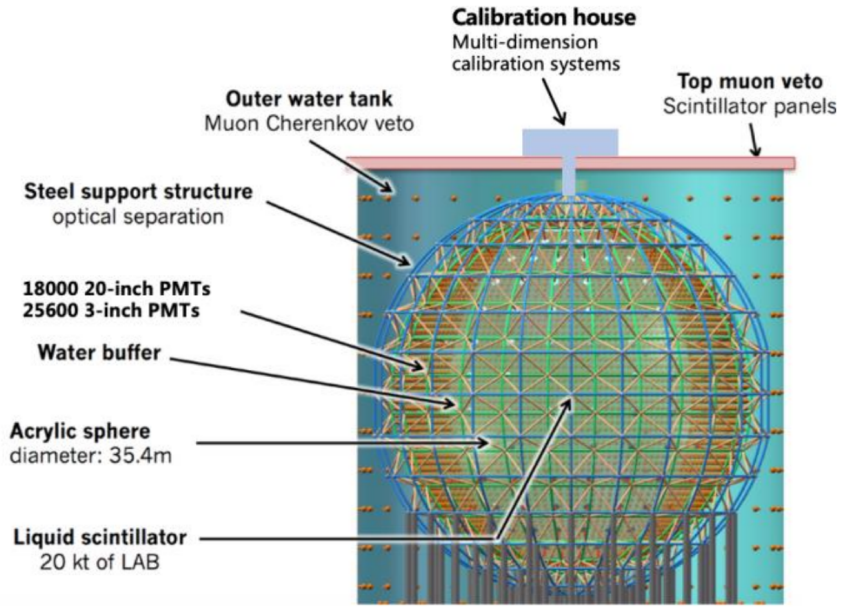


Figure 1.2: Schematic view of the Vessel.

1.2 JUNO electronics

The JUNO electronic system has two parts. The "wet electronics" system, which performs analog signal processing, and the "dry electronics" system, which performs data acquisition (DAQ) and triggering (TRG). Both systems are inter-connected with 100m Ethernet cables [2]. Figure 1.3 shows how the components of both systems are interconnected.

Three cables pull from the wet electronics. One of them, the synchronous link, is directly connected to the trigger system. It carries the trigger information, which indicates when to perform data acquisition. The second one , the asynchronous link, communicates with the DAQ system to send the waveforms and the different information. This cable system is developed by the Institute of High Energy Physics (IHEP) in Beijing [5, 6].

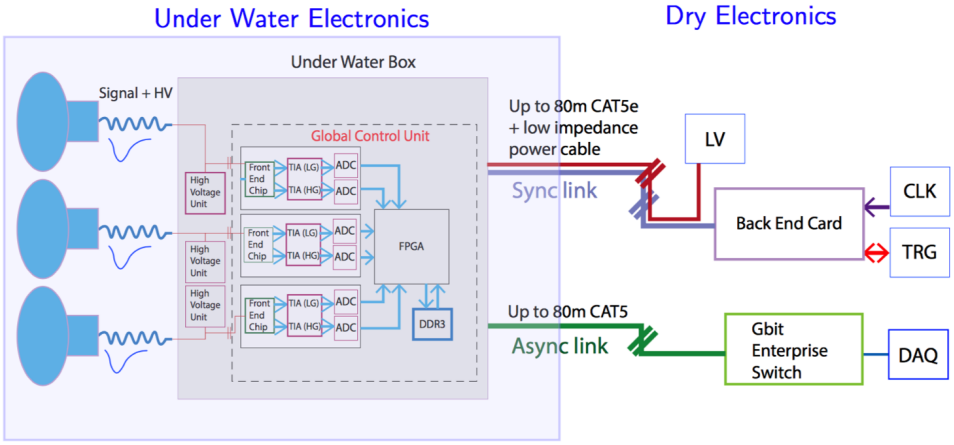


Figure 1.3: Schematic view of JUNO electronic system.

1.2.1 Wet electronics

The wet electronics of JUNO performs basic tasks of data acquisition. The PMTs collect (in groups of three) the light induced by the interaction of the neutrinos with the liquid scintillator. This light is transformed into an electrical signal, which is usually called "PMT waveform" in the literature. This signal will be digitized inside the GCU, and then sent to the BEC through the synchronous link.

1.2.2 Dry electronics

The main function of the dry electronics system is to control the trigger requests. The information goes back and forth through most channels between the Central Trigger Unit (CTU) and the GCU.

The PMTs are organized in groups of three. The hit information captured by these groups i to the dry electronics system, along with the trigger requests generated by the GCUs. They are collected and aligned in an FPGA Mezzanine card (TTIM). Then, the FPGA makes a sum of the hit information from 48 different GCUs. This result is sent

towards the Reorganize Multiplex Units (RMU). Each RMU works with 7 BECs. The RMUs send the received data to a single CTU (figure 1.4) [7].

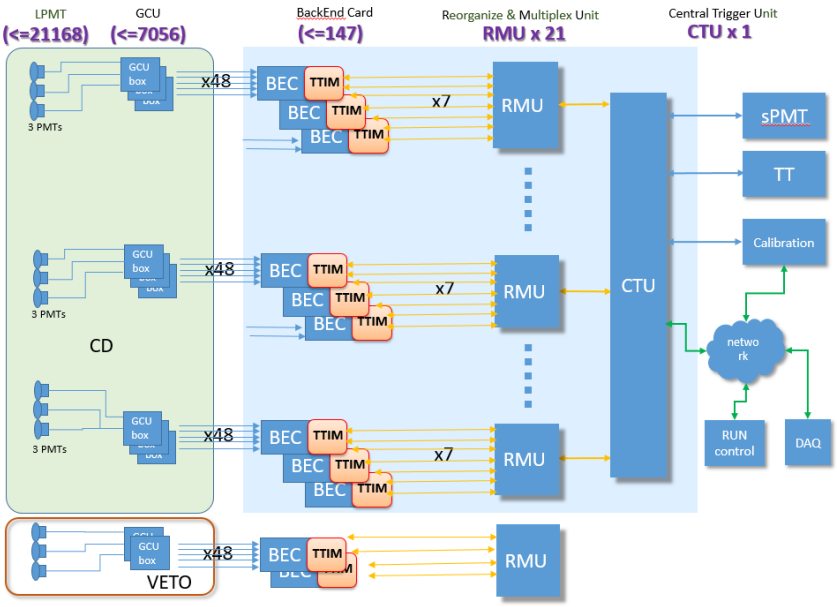


Figure 1.4: Trigger information flow.

Figure 1.4 shows the actual trigger system of JUNO [2], which works in two ways. We will explain how the signals flow from right to left, and then from left to right of the image.

From right to left, the trigger decision is computed in the CTU. These are signals emitted by the CTU to trigger data acquisition. The BECs transmit the incoming trigger decision signals (which arrive through the TTIM) towards the GCUs.

From left to right, PMTs send an analog signal to the GCUs. These then digitize the signals using the ADU, and store them in a large local memory under the control of an FPGA. They send these signals along with trigger requests to the Back-End Card (BEC), and they arrive through 6 Mezzanine cards placed on each BEC, with 8 Ethernet cables each. Therefore, a total of 48 trigger signals are transmitted through Ethernet cables. The BECs also contain an FPGA that performs the synchronization of the signals. They distribute the clock signal to the GCU to be able to synchronize the acquisition. To manage the acquired signals and trigger requests, a total of 48 Ethernet cables are connected to a single BEC through 6 mezzanine cards.

An open box is used to gather some of the elements of the dry electroniccs system, like the Mezzanines, the BEC and the TTIM. Their exact placement is shown in figure 1.5.

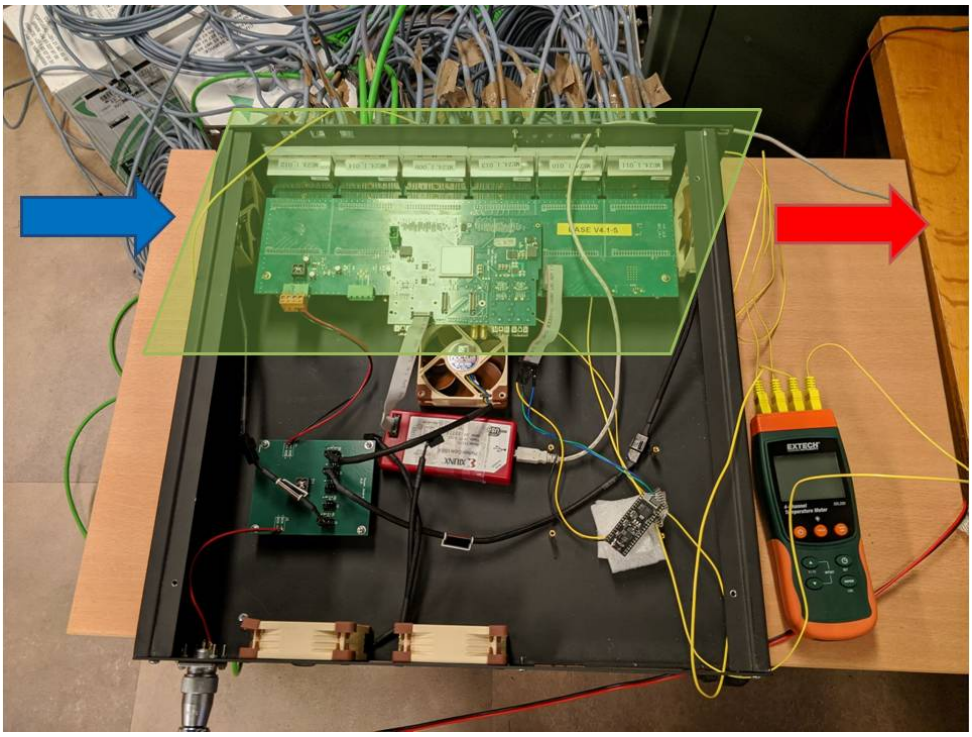
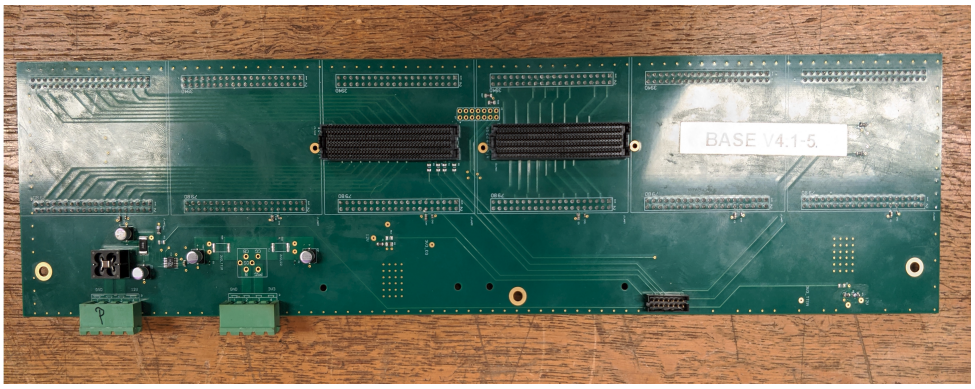


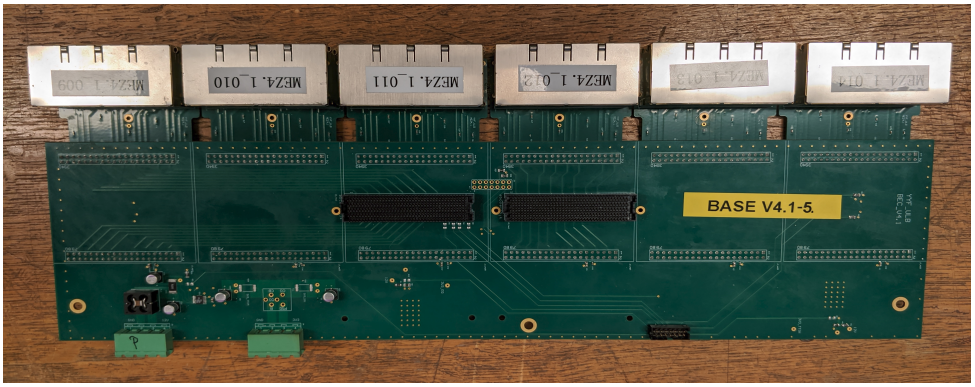
Figure 1.5: Open box. Inside the green rectangle, we can see the 6 Mezzanines, the BEC and the TTIM mounted on the BEC. The airflow that cools the system is represented by the blue arrow (cold), and red arrow (hot).

1.2.3 Back-End card system

The back-End card system is the main component of the dry electronics (figure 1.6b). There are 16 equalizers placed in two rows of 8 (figure 1.7) on each of the 6 mezzanines present on the BECs, which are used to treat the signals coming from the GCU. These equalizers are expected to be the components from the BEC system with the higher heat dissipation.



(a) The BEC base board - Front view.



(b) The BEC base board (green) and the 6 Mezzanines at the top – Front view.

Figure 1.6: The BEC baseboard without (top) and with (bottom) the 6 mezzanines.

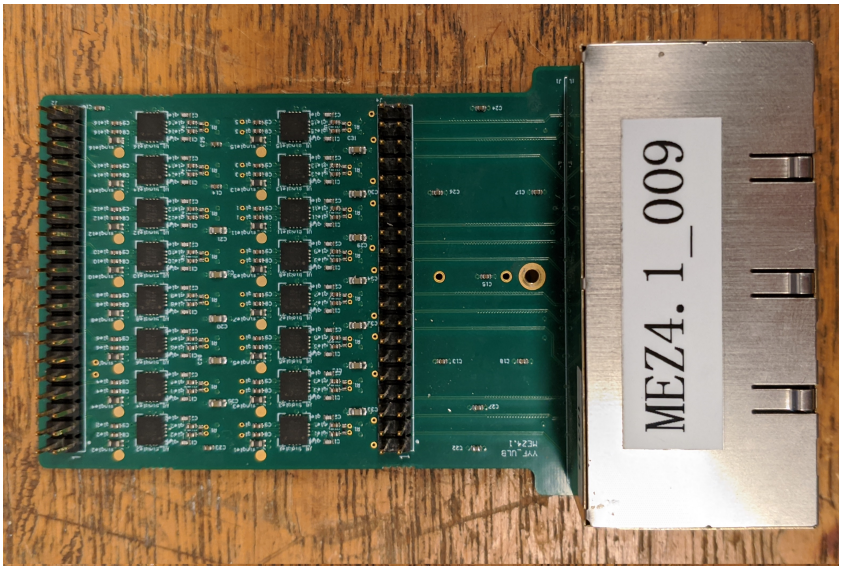


Figure 1.7: The extender mezzanine with the 16 equalizers (on the green PCB) and the 8 RJ45 connectors on the front side.

1.3 Temperature Control

A real-time monitoring of the temperatures of the equalizers is essential to verify the proper functioning of the concentrator card for a period of time of 6 years, as they use about 0.15 W of energy (which is enough to increase their temperature above the correct functioning threshold). These values should be transmitted on a reliable channel at regular intervals to a computer, from which real-time monitoring could be performed. The values should be logged to a database or file to facilitate the analysis. Finding a solution to this problem is the main subject of this thesis.

Chapter 2

Requirements and choices

The main goal of this thesis is to design a system to monitor the temperature of the 17280 equalizers present in the 180 BECs. We want to retrieve the temperatures from the sensors to a central computer. Two main logistic options are available to transport the information. The first one is the use of cables directly plugged into the computer and into the sensors. However, this solution would need at least 180 additional cables, and they would need to be about 20 meters long. The second solution is to send the temperatures over the air using some wireless protocol. Due to the already huge amount (almost five thousand) of long cables that have to be used for other tasks of the dry electronics system, a wireless implementation was preferred. We now present the hardware and software requirements to design this system.

2.1 Hardware choices

In order to interact with the sensors and send the information over the air to the central computer, some kind of programmable computer, chip or motherboard is needed, and the sensors must be accessible using some protocol compatible with the rest of the hardware. We present the characteristics of the hardware components we used in order to achieve our goal, as well as the reasons for our choices.

2.1.1 ESP32

The motherboard that we chose for this work is the "ESP32 pico board". This is an ESP32-based mini development board produced by Espressif [8]. Its relatively small size allows to easily place it inside the aluminium box of the dry electronics system. It is a very cheap board that costs about 7$. It has two 32-bit cores, an 8KB RAM, good Wi-Fi capabilities (including support for Wi-Fi mesh), and a moderate energy consumption of 200 mA when working with Wi-Fi and the CPU. Moreover, there are plenty of resources online to aid at the development of firmware for this board, as well as for the development of software able to interact with it. It is programmable in C using the popular Arduino Integrated Development Environment (IDE), or Eclipse IDE. It also has support for FreeRTOS, an open source real-time operating system providing many OS functions for embedded systems [9].

2.1.1.1 Physical characteristics

The dimensions of the ESP32 pico board are 52 x 20.3 x 10 mm (figure 2.1). This board fits perfectly inside the 480 x 450 x 95 mm aluminium box.

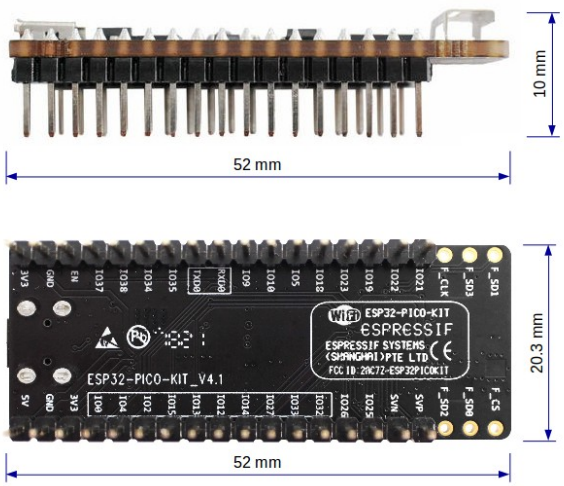


Figure 2.1: ESP32 from two views [8].

2.1.1.2 General features

This board can be connected to a computer through a cable (using the CP2102 driver) in order to flash it with a firmware [8]. It supports firmware compiled with the "xtensa-esp32-elf" compiler [10], and programmed with the ESP-IDF library [11]. Additionally, ESP-IDF has been ported to an Arduino-based library (called arduino-esp32 [12]) that allows the programmer to use common Arduino functions in the firmware's code, and to compile it within the Arduino IDE.

ESP32 is a relatively powerful board in terms of capabilities. It has an antenna that allows it to communicate through WiFi and Bluetooth. It can be powered by a micro-usb cable, or by connecting it to a 3V or 5V source. It can also communicate through an I^2C bus (see 3.1.1) and control other devices by acting as a "master" device. Additionally, it has a button that can be programmed, and another button that reboots the system. For all these useful features, we decided to go along with this board.

2.1.1.3 Permanent memory

ESP32s permanent memory is a flash memory of 512 bytes. Data saved in the permanent memory remains there even when the ESP32 resets or is powered off. This non-volatile storage uses a portion of main flash memory. Single-level cell flash memories usually support around 100000 P/E cycles. A P/E cycle stands for Program/Erase cycle, which is a sequence of events in which data is written to the flash memory cell, then erased, and then rewritten [13]. A permanent memory is useful in many applications, in particular, when we need to save information and keep it after the system reboots.

2.1.1.4 GPIO

The ESP32 has 40 General Purpose Input/Output (GPIO). Some can be programmed for many different tasks. We might need at least a few to achieve some tasks like communication with sensors. Others serve as Analog to Digital Converter (ADC) channels, Digital to Analog Converter (DAC) channels, Real-Time Clock (RTC) channels, Pulse-width modulation (PWM) channels or I^2C channels.

2.1.1.5 Buttons

The ESP32 pico has two buttons. The BOOT button is entirely programmable (GPIO0) and it can be configured to respond both to single and to long presses. The EN button reboots the ESP32 when pressed, and cannot be programmed to do anything else.

2.1.1.6 WiFi

Overview The best way to communicate over the air at relatively long distances (several meters) is by using a wireless network. ESP32 supports many versions of the popular IEEE 802.11 protocol, which is widely used around the world to create Wireless Local Area Networks (WLAN) for computer communication between devices. These versions include 802.11 B, 802.11 BG, 802.11 BGN, 802.11 BGNLR, and 802.11 LR. The latter is an Espressif-patented mode that can achieve a one-kilometer line of sight range between LR-compatible devices [14]. IEEE 802.11 uses various frequencies, which means more than one network can coexist at the same physical place without interference. Using this protocol, an ESP32 can act both as an access point device to create a network, or can connect to an access point as a station device. The default frequency is set to 1 (figure 2.2).

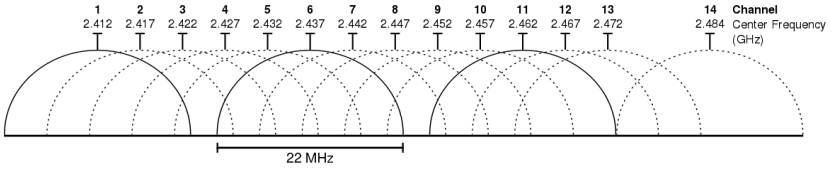


Figure 2.2: Different allowed WiFi bandwidths. Some overlap, but 1 and 6 don't [15].

AP An ESP32, when set as an access point (AP), can be accessed by any other ESP32 or device with WiFi capabilities. It can create a WiFi network with a Service Set Identifier (SSID), a password and a channel. It can sustain a maximum of 10 clients connected at the same time (figure 2.3).

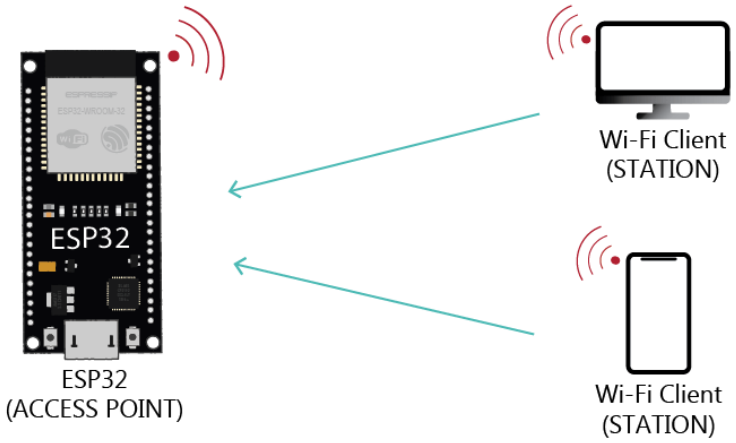


Figure 2.3: ESP32 AP mode [16]

Station An ESP32, when set as a WiFi station, can connect to other networks (like a router or a mesh of boards). If it is connected to a router, it is assigned an IP address (usually by Dynamic Host Configuration Protocol (DHCP) [17]) in the LAN, and it can communicate with other devices.

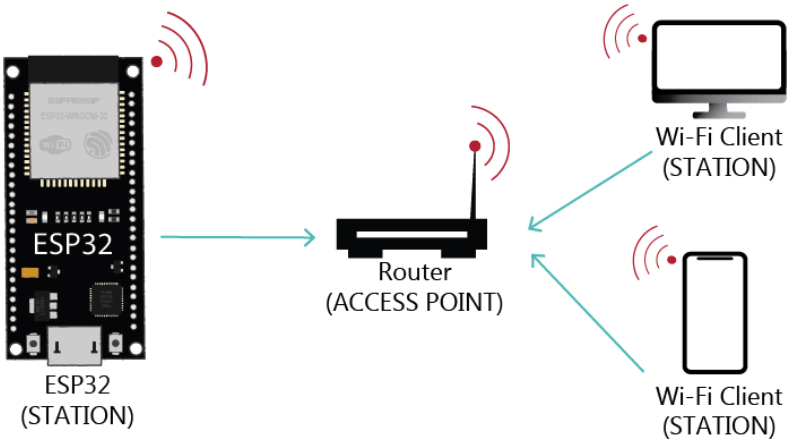


Figure 2.4: ESP32 Station mode [16].

AP and Station The ESP32 can be set both as a WiFi station and an access point simultaneously. This allows it to receive multiple connections, and to connect to one (and only one) other AP, at the same time.

2.1.2 TMP100

2.1.2.1 Overview

The sensors that are used are TMP100s from Texas Instruments. These are digital temperature sensors ([18]). They offer an average accuracy of ±1°C, and a maximum error of ±2°C with respect to the actual temperature of the BEC's silicon on the spot the sensor is placed at. The maximum resolution offered is 0.0625°C, which is the smallest change that can be detected in the temperature measured. These characteristics allow us to monitor

the temperatures with sufficient accuracy and precision. Their body size is of 2.90 mm × 1.60 mm, which fits perfectly on the BECs.

2.1.2.2 Sensor placement

We chose to place 3 temperature sensors on each BEC. Two of them will be at the sides, and one at the center (figure 2.5). As they are placed on the BEC, there is a small gap of 8 mm between the sensor and the equalizers (figure 2.7). We should be aware that there might be a difference between the temperature of the equalizers and the temperature captured by the TMP100 (see Chapter 4).

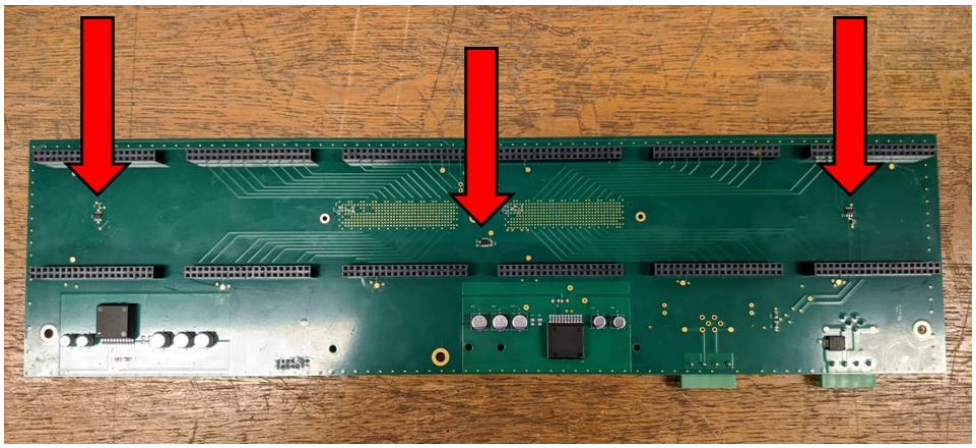


Figure 2.5: The BEC base board and the 3 temperature sensors (red arrows) - Rear view.

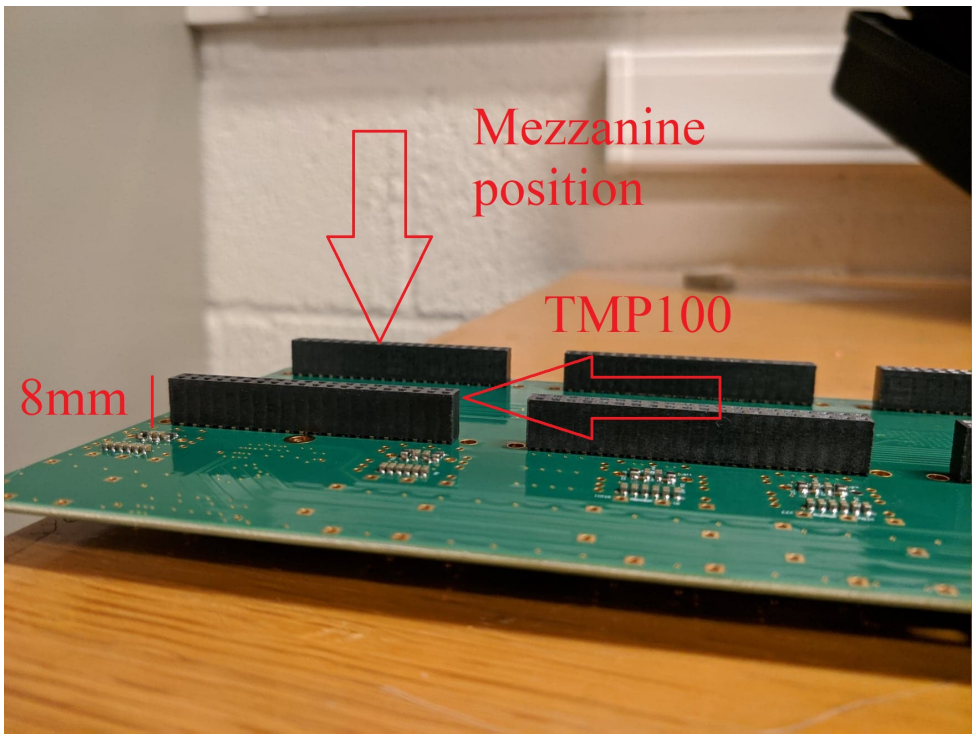


Figure 2.6: Side view of a BEC and a slot for a Mezzanine. Arrows indicating the positions of the different elements.

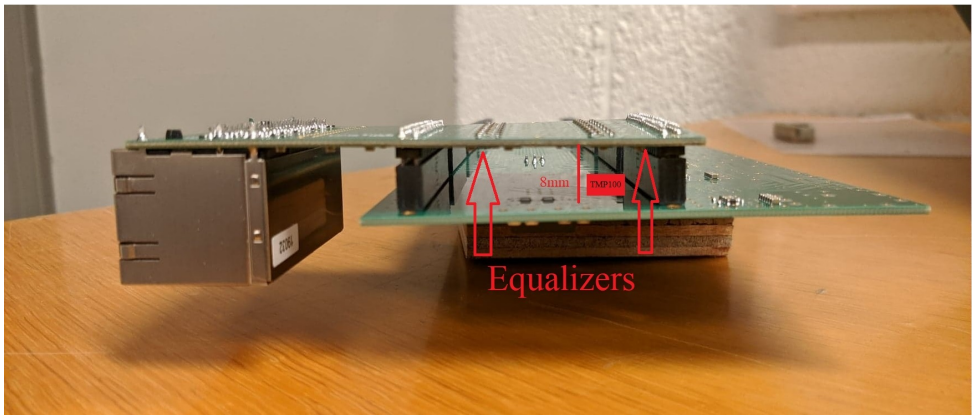


Figure 2.7: Lateral view of a BEC with a Mezzanine plugged on the top. The TMP100 sensor is represented by a red square, and is placed 8mm appart from the equalizers.

2.2 Software choices

2.2.1 Data transmission

Each ESP32 board should transmit its values at a frequency of 0.2 Hz, i.e. one message every 5 seconds. This rate of transmission is sufficient to monitor the temperature of the equalizers and catch any spikes in temperature that could take place. This message should clearly separate the three temperature values, and should also contain an error code for debugging, as well as the name of the board where the temperatures were measured.

When the message arrives to the central computer, it should be logged into a database with the time and date it was received at. We should be able to access any message and any part of the message efficiently. Moreover, we would like to monitor the status of all the motherboards both in real time and after a certain data-taking time to properly validate the implementation.

2.2.2 Configuration

We want to be able to configure some parameters of the ESP32 after it being flashed. Having WiFi and Bluetooth, the boards should include some kind of interface to be able to interact with a smartphone or with a remote computer. This interface should accept a number of commands that could change the value of some variables in the stack or permanent memory, as well as to change the board's behaviour by rebooting it into a new mode.

Chapter 3

Engineering Analysis and Development

This chapter provides an in-depth analysis of the solution we designed. We start by explaining the logistic used to read the temperature values from the sensors at regular intervals. We give details about the functioning of the hardware, the different protocols, and the practical implementation. Then, we proceed to design the retrieval system based on wireless communication. We explain the logic behind our choices for the network architecture, and we provide a practical implementation that aims to provide the best results possible.

3.1 Temperature retrieval

The first step in the solution designed to meet the requirements of this thesis is to retrieve the temperature values from the TMP100 sensors. The main objective is to perform a simple *read* operation on the sensor every 5 seconds, and get the temperatures with the best resolution possible.

3.1.1 I^2C Bus

TMP100 sensors are I^2C interface-compatible. I^2C is a bus that allows a device called "master" to interact with several devices called "slaves". A bus is a communication system that transfers data between devices inside a computer, or between computers. It is made of hardware components (like wires and transistors) and software components (like protocols). Communication takes place using two lines: the SDA (data) line, and the SCL (clock) line (figure 3.1). This bus is synchronous, and synchronization is done through the SCL line, which is always driven by the bus controller (the master). The SDA line serves to transmit bytes of data. The sensors have been integrated in the BEC in such a way so that they can be accessed through an I^2C, bus with a different address each. We will now analyze the most important parts of the bus for us to understand how we should configure the sensors and how to interact with them.

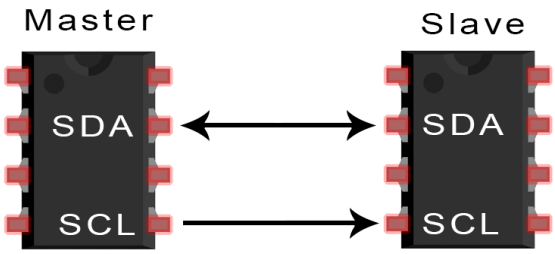


Figure 3.1: Master-Slave architecture [19].

3.1.1.1 Hardware

The two lines present on the bus are "open drain" lines (figure 3.2). An open drain terminal is an output type of an integrated circuit which behaves like a switch. It is connected to the ground when a high voltage (logic 1) is applied to the gate, and it presents a high resistance when a low voltage (logic 0) is applied to the gate. This high resistance state occurs because the terminal is at an undefined voltage (called floating). Therefore, such a device requires an external pull-up resistor connected to the positive voltage rail in order to provide a logic 1 as output.

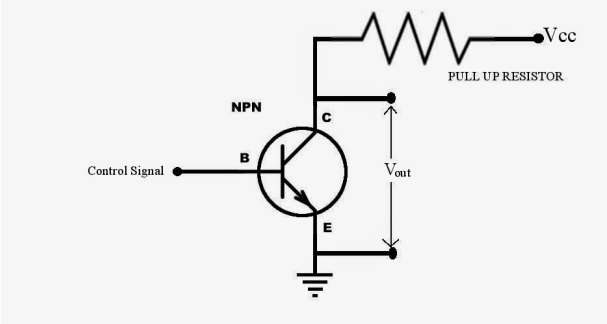


Figure 3.2: Open drain - Transistor diagram [20].

3.1.1.2 Software

Messages The protocol of communication between devices in the bus is based on messages. Messages are broken up into two types of frame: an address frame, which contains the bus address of the slave to which the message is destined for, and one or more data frames which contain 8-bit data messages that are sent from master to slave, or vice versa. Data is placed on the SDA line after the SCL goes low, and it is sampled after the SCL goes high. To initiate the transmission, the master device leaves SCL high and pulls SDA low. This puts all peripheral devices on notice that a transmission is about to start. Now the master can send the frame. To end the transmission, the master pulls the SDA high [21].

3.1.2 SMBus

A System Management Bus (SMBus) is a two-wire bus. It uses single ended signaling (figure 3.3), in which one wire carries a varying voltage that represents the signal, while

the other wire is connected to a reference voltage, usually to the ground. The TMP100 has an SMBus that is used to trigger ALERT commands by raising the voltage of the line. We will see that these commands can be useful in certain configurations of the sensor.

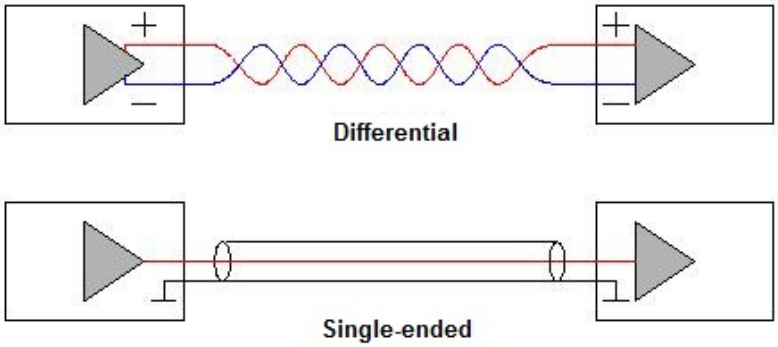


Figure 3.3: Single-Ended line [22].

3.1.3 Configuration of the temperature sensors

3.1.3.1 Addresses

As we have seen in the message section (see section 3.1.1.2), we need to know the address of a certain device in the bus to be able to communicate with it. The documentation of both the BEC (figure 3.4) and the temperature sensor (figure 3.6) provide information to help us address the three devices in function of how they are connected in the circuit. The TMP100 device features two address pins (ADD0 and ADD1) to allow up to eight devices to be addressed on a single I^2C bus.

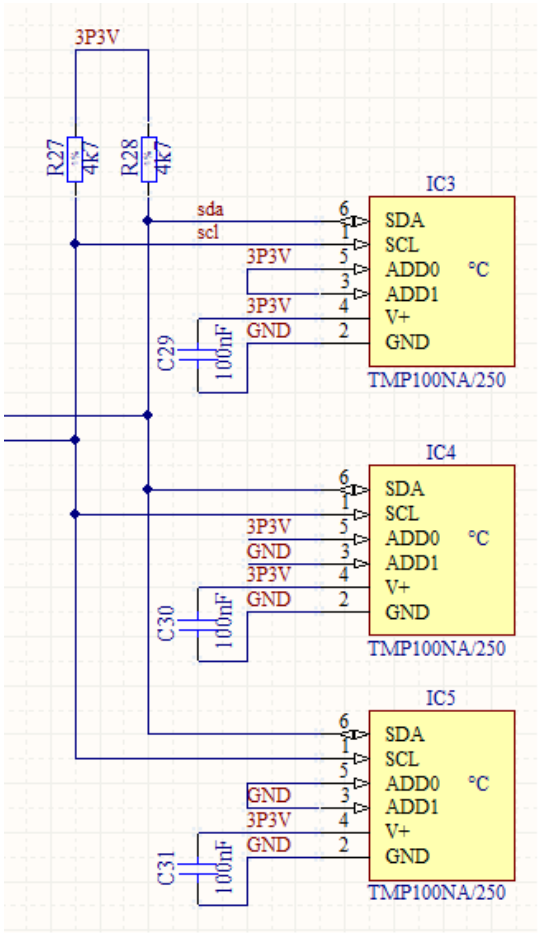


Figure 3.4: TMP100s distribution in the BEC. We have the three sensors (IC3, IC4 and IC5) connected with different voltages for the ADD0 and ADD1 bits.

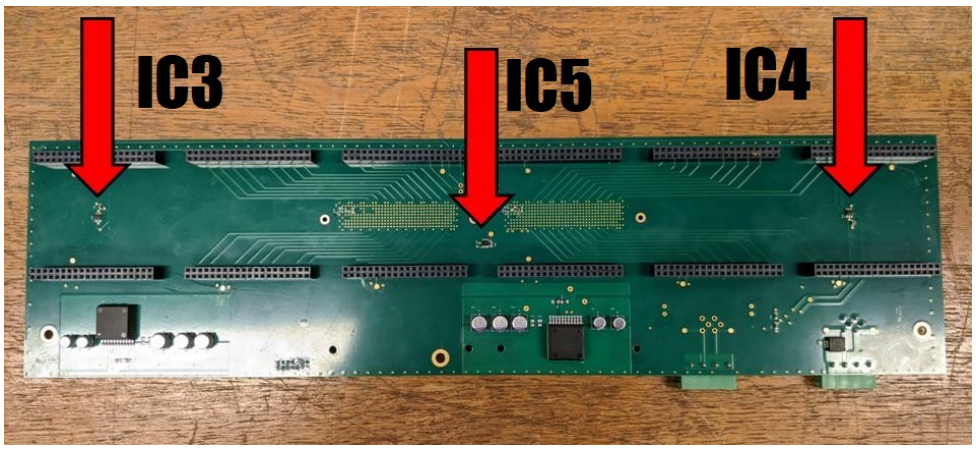


Figure 3.5: TMP100s location in the BEC.

ADD1	ADD0	SLAVE ADDRESS
0	0	1001000
0	Float	1001001
0	1	1001010
1	0	1001100
1	Float	1001101
1	1	1001110
Float	0	1001011
Float	1	1001111

Figure 3.6: Slave addresses in the bus in function of the values of the ADD0 and ADD1 bits [18].

From figures 3.4, 3.5 and 3.6, we deduce that the following addresses should be used:

TMP100 nb	Address (bin)
IC3	1001110
IC4	1001010
IC5	1001000

Table 3.1: Addresses of the three sensors present in a BEC.

3.1.3.2 Registers

There are 5 registers in TMP100 sensors: pointer, temperature, configuration, T_{LOW} and T_{HIGH} registers. They are illustrated in figure 3.7. The Pointer Register serves as an intermediary register to access the other 4 registers. When a master device wants to access any of those 4, it sends their corresponding register address (table 3.8) through SDA, immediately after sending the address of the slave device it wants to access (as the I^2C protocol indicates). The data-type frame that has to be sent has the structure of table 3.9. As we can see, the two least significant bits (LSBs) of the byte of data serve to identify the register we want to access. This register address is written into the Pointer Register by the IO Control Interface. The other 4 registers are used to retrieve the temperature, configure the sensor, and define the temperature thresholds.

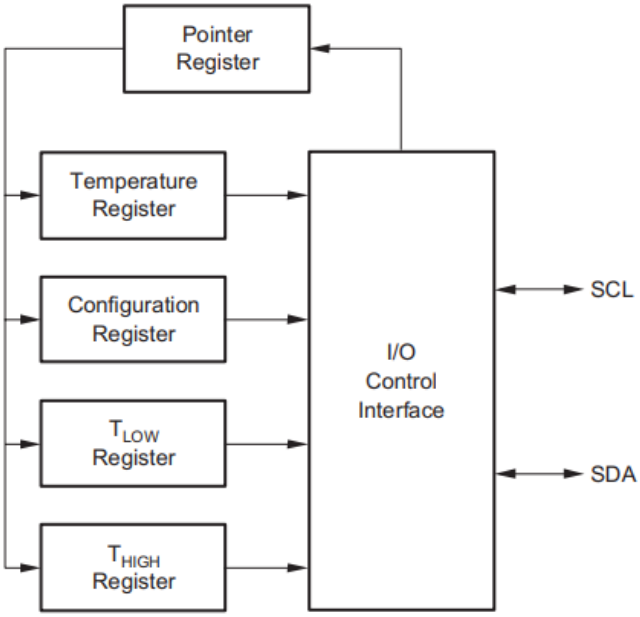


Figure 3.7: I^2C TMP100 registers [18].

P1	P0	TYPE	REGISTER
0	0	R only, default	Temperature Register
0	1	R/W	Configuration Register
1	0	R/W	T_{LOW} Register
1	1	R/W	T_{HIGH} Register

Figure 3.8: Bits used to address the 4 registers [18].

P7	P6	P5	P4	P3	P2	P1	P0
0	0	0	0	0	0	Register Bits	

Figure 3.9: I^2C Register byte [18].

Bit distribution of the Configuration Register

Resolution The "Converter Resolution" bits control the resolution of the internal analog-to-digital converter (ADC). These bits allow the user to chose between a higher resolution or a faster conversion time. They are set according to table 3.10.

R1	R0	RESOLUTION	CONVERSION TIME (Typical)
0	0	9 bits (0.5°C)	40 ms
0	1	10 bits (0.25°C)	80 ms
1	0	11 bits (0.125°C)	160 ms
1	1	12 bits (0.0625°C)	320 ms

Figure 3.10: I^2C Resolution registers [18].

Fault Conditions A "fault condition" occurs when the measured temperature exceeds the limits set by the user in the T_{HIGH} and T_{LOW} registers. Additionally, the number of fault conditions required to generate an ALERT command can be programmed using the "Fault Queue" bits in figure 3.11. The "Fault Queue" bits can be used to prevent the generation of a false ALERT command resulting from environmental noise.

F1	F0	CONSECUTIVE FAULTS
0	0	1
0	1	2
1	0	4
1	1	6

Figure 3.11: I^2C Fault conditions [18].

Polarity The Polarity bit (POL) of the TMP100 device lets the user adjust the polarity of the ALERT command. If the POL bit is set to 0 (as it is by default), the ALERT line becomes active low. When the POL bit is set to 1, the ALERT line becomes active high (the state of the ALERT line is inverted).

"Device Functional Modes" bits The TMP100 can be configured in different ways. Depending on the user's goal, different bits of the configuration register allow to modify the sensor's behavior.

- **Shutdown mode:** This mode allows the sensor to shutdown all its circuitry except for the serial interface. This reduces consumption down to less than 1 µA.
- **One-shot/Alert mode:** This mode allows a sensor in shutdown mode to start a temperature reading whenever a 1 is written to the ALERT line. After the temperature is read, the sensor returns to shutdown mode. This mode can be activated by setting $OS = 1$.
- **Thermostat Mode:** Bit that indicates whether the device should operate in Comparator mode or Interrupt mode.
- **Comparator Mode:** The sensor enters this mode when the "Thermostat Mode" bit is set to 0 ($TM = 0$). The ALERT line is put to 1 when the temperature equals or exceeds the value in the T_{HIGH} register, and remains active until the temperature falls below the value in the T_{LOW} register.
- **Interrupt Mode:** The sensor enters this mode when the "Thermostat Mode" bit is set to 1 ($TM = 1$). The ALERT line is put to 1 when the temperature exceeds the value of register T_{HIGH} or goes below the value of the T_{LOW} register. The ALERT line is put to low when the host controller reads the temperature register.

Table 3.12 shows the complete structure of the byte that is stored in the configuration register. There are all the bits we have seen: the One-shot/Alert bit, the R1 and R0 resolution bits, the F0 and F1 fault condition bits, the POL polarity bit, and the TM thermostat mode bit [18].

BYTE	D7	D6	D5	D4	D3	D2	D1	D0
1	OS/ALERT	R1	R0	F1	F0	POL	TM	SD

Figure 3.12: I^2C Configuration byte with each bit.

Communication operations

Write To write some data onto a slave's register, the temperature sensor works on "Slave Receiver Mode" (see figure 3.13 for a diagram representation).

- Master sends slave address, concerned slave responds with ACK (acknowledgement) byte.
- Master sends the pointer to the register where it wants to write, slave responds with ACK.
- Master sends as much data as it wants, slave responds with ACK.
- Master sends stop byte when it wants to end the transmission.

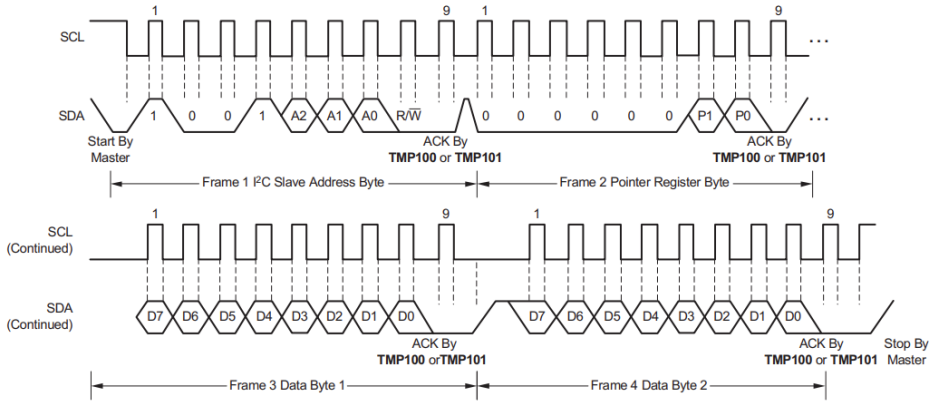


Figure 3.13: I^2C timing diagram for write operation [18].

Read To read some data from a slave's registers, the temperature sensor works on "Slave Transmitter Mode" (see figure 3.14 for a diagram representation).

- Master sends slave address, concerned slave responds with ACK (acknowledgement) byte.
- Master sends the pointer to the register from where it wants to read, slave responds with ACK.
- Master sends slave address again, slave responds with ACK.
- Slave sends content of the register, master responds with ACK.
- Master sends stop byte when it wants to end the transmission.

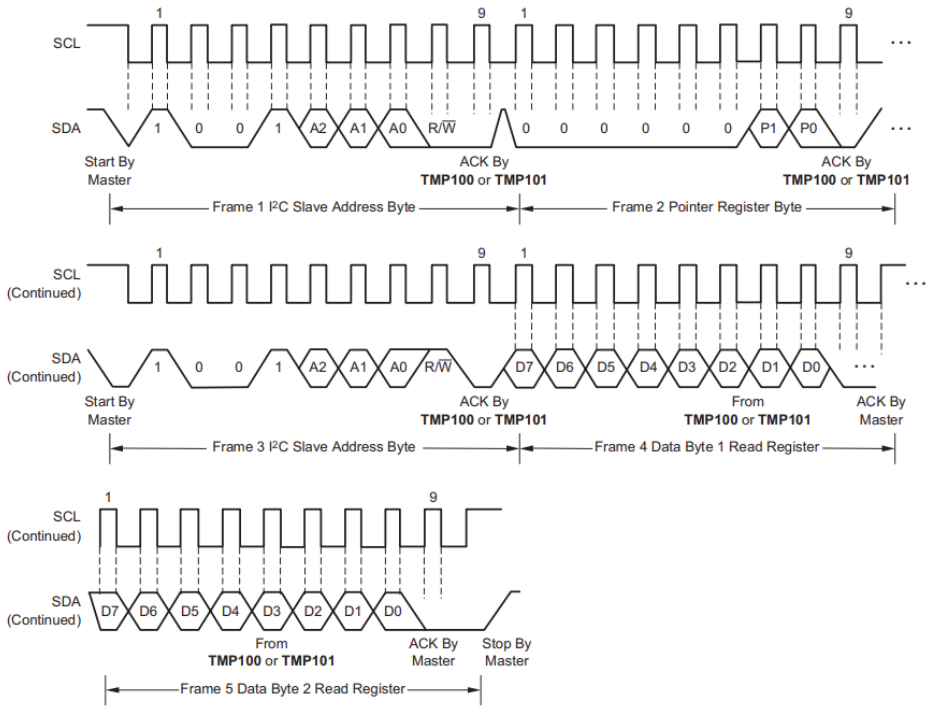


Figure 3.14: I^2C timing diagram for read operation [18].

3.1.3.3 Configuration choices

We chose to configure each bit of the configuration register (figure 3.12) in the following way:

- **Resolution:** We chose a 12 bit resolution for the temperature values ($R0 = 1$ and $R1 = 1$). This is the highest resolution available, and the 320 ms conversion time can be neglected compared to the 5 s sampling frequency that we need to use.
- **Fault condition:** As we don't use T_{HIGH} and T_{LOW}, we can set these bits to $F0 = 0$ and $F1 = 0$.
- **Shut down mode:** The BEC card is plugged permanently to an energy source, so we don't need to save power on the temperature sensor.
- **Alert mode:** One-shot/Alert mode is not needed, as we don't need to keep the temperature between two thresholds. A simple monitoring is enough.
- **Polarity:** We leave the polarity bit to 0, as we don't use the One-shot/Alert mode.
- **Thermostat mode:** There is no need to use this mode as we don't use the threshold temperatures ($TM = 0$).
- **Configuration register byte:** set to "0110 0000" (0x60 in hexadecimal).

To configure the three sensors in practice, the master needs to send two messages to each of the sensors. The first message contains the sensor's address as the first frame, and the configuration register address as the second frame. The second message also contains the sensor's address as the first frame, and the configuration byte as the second frame. From

figure 3.8, we have $P0 = 0$ and $P1 = 0$ (which is 0x01 in hexadecimal) as the configuration register's address. From table 3.1 we get the sensor's addresses.

- **Message 1 sensor IC5, Master sends:** frame1: 0x48, frame2: 0x01, slave ACK
- **Message 1 sensor IC4, Master sends:** frame1: 0x4A, frame2: 0x01, slave ACK
- **Message 1 sensor IC3, Master sends:** frame1: 0x4E, frame2: 0x01, slave ACK
- **Message 2 sensor IC5, Master sends:** frame1: 0x48, frame2: 0x60, slave ACK
- **Message 2 sensor IC4, Master sends:** frame1: 0x4A, frame2: 0x60, slave ACK
- **Message 2 sensor IC3, Master sends:** frame1: 0x4E, frame2: 0x60, slave ACK

3.1.4 Reading of the values

The documentation shows how the temperature values are stored in two bytes (figure 3.15).

Byte 1 of the Temperature Register

D7	D6	D5	D4	D3	D2	D1	D0
T11	T10	T9	T8	T7	T6	T5	T4

Byte 2 of the Temperature Register

D7	D6	D5	D4	D3	D2	D1	D0
T3	T2	T1	T0	0	0	0	0

Figure 3.15: Register where the temperature of one measure is stored [18].

As we chose a 12-bit resolution, the master needs to read the whole first byte, and half of the second byte to get the temperature. Table 3.16 shows some decimal numbers, and their binary representation in 12 bit 2's complement. The maximum temperature that can be captured is 128 °C, which can be encoded with 1 byte. The 4 remaining bits allow for the encoding of decimals.

TEMPERATURE (°C)	DIGITAL OUTPUT	
	BINARY	HEX
128	0111 1111 1111	7FF
127.9375	0111 1111 1111	7FF
100	0110 0100 0000	640
80	0101 0000 0000	500
75	0100 1011 0000	4B0
50	0011 0010 0000	320
25	0001 1001 0000	190
0.25	0000 0000 0100	004
0	0000 0000 0000	000
–0.25	1111 1111 1100	FFC
–25	1110 0111 0000	E70
–55	1100 1001 0000	C90
–128	1000 0000 0000	800

Figure 3.16: Register where the temperature of one capture is stored [18].

Now, to retrieve the temperature values and compute the decimal value, the master sends the following frame to each sensor (only IC5 is shown as an example):

- **Message 1 sensor IC5, Master sends:** frame1: 0x48, frame2: 0x00 (send temperature register), slave ACK
- **Message 2 sensor IC5, Master sends:** frame1: 0x48 (send start sequence again), slave ACK
- **Message 2 sensor IC5, Slave sends:** frame1: Byte 1 of temperature, Master ACK
- **Message 2 sensor IC5, Slave sends:** frame1: Byte 2 of temperature, Master ACK

The documentation tells us that the most significant bit (MSB) of the 12-bit temperature representation is byte 1, and the LSB is byte 2. This information allows us to build the following formula to convert the 12 bits into a decimal value:

$$T_{Dec} = \frac{Byte1_{Dec} \cdot 256 + ((Byte2)_{Hex} \wedge 0xF0)_{Dec}}{16} \cdot Res_{Dec} \tag{3.1}$$

Where the resolution is $Res_{Dec} = 0.0625$ in our case.

3.1.5 Programming

Arduino's programs contain two main parts. The first part is a "setup" method. This method is run once the board boots. It is useful to initialize several parts of the program, like the Wi-Fi connections or the I^2C connections. The second part is the "loop" method. This method is the main task of the program, and loops infinitely. The main routines of the program are written in this method.

The library used to implement an I^2C master is the *Wire* library [23]. This library allows you to communicate with I^2C devices. We defined the SDA to be ESP32s GPIO number 21, and the SCL to be GPIO number 22. Inside Arduino's setup function, we start a connection to the I^2C bus using the *begin()* method. Then, we send the address of the configuration register to each sensor using the *beginTransmission()* method. We follow by writing the configuration byte using the *write()* method, and finish by ending the transmission with the *endTransmission()* method. It is important to pause the program for a small delay of 250ms (using arduino's *delay* function) after the configuration of one single sensor is over before jumping to configure the next one. This is due to the fact that the I^2C bus needs some time for the stop condition to complete, and to be available again for the next interaction.

To retrieve the temperatures every 5 seconds, we made a routine that can be called each time we want to access one of the sensors. We begin the transmission with the slave by sending its address on the bus, just like before. Next, we send the register address corresponding to the temperature register. We then send the address again, and use the *requestFrom* function to request two bytes of data. These two bytes contain the 12-bit resolution temperature that we proceed to decode with equation 3.1.

3.2 Retrieval to the computer

Once the three temperatures of the three sensors are read by the ESP32s, it is time for it to send them to the Central Computer over Wi-Fi. ESP32 has an antenna that can be used to communicate by Bluetooth or Wi-Fi. In this section we explain how the complete process of retrieval is designed, from the time the boards get the temperatures to when the messages are saved into the database. We also give additional explanations about some features added to the system in order for it to be easier to configure or use in practice.

3.2.1 Wi-Fi architecture

As we explained in the Introduction to JUNO (see Chapter 1), the sensors of each of the 180 BECs have to be monitored every 5 seconds. The trivial solution to allow all the boards to stream their messages to the central computer would be to connect them all at the same time to the Wi-Fi AP, using a WLAN. However, connecting this many boards simultaneously would saturate the AP, and the retrieval would be unreliable at best (figure 3.17). Moreover, the aluminium boxes containing the BECs are placed in a straight line, separated by half a meter, which means the AP might not be reachable by some boards.

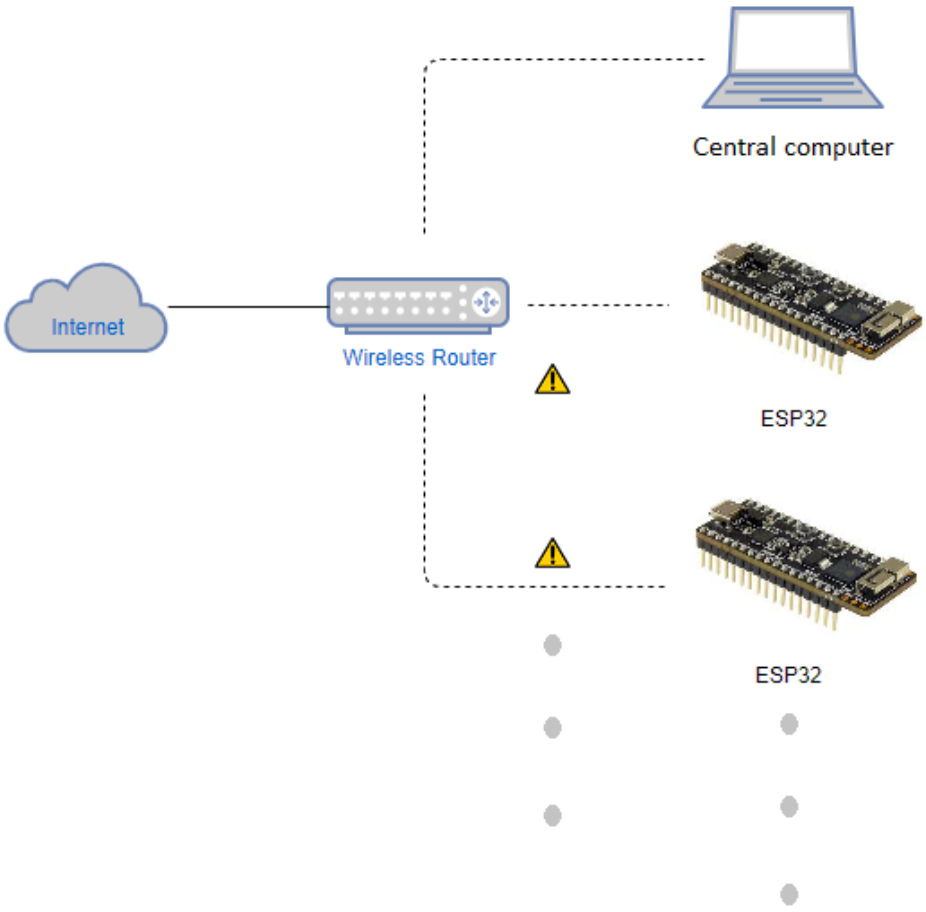


Figure 3.17: View of the temperature retrieval by many ESP32 board inside a LAN [24].

The solution we came up with is the deployment of a Wi-Fi-Mesh architecture made of the 180 boards. This type of configuration ensures that all the boards that form this mesh act both as a Wi-Fi Access Point and as a Wi-Fi Client, which guarantees that there exists a path between any pair of connected boards. ESP32s allow a maximum of 10 incoming connections when they work as access points, and can connect to at most one node when working as a station. The topology formed by a few boards will look like figure 3.18.

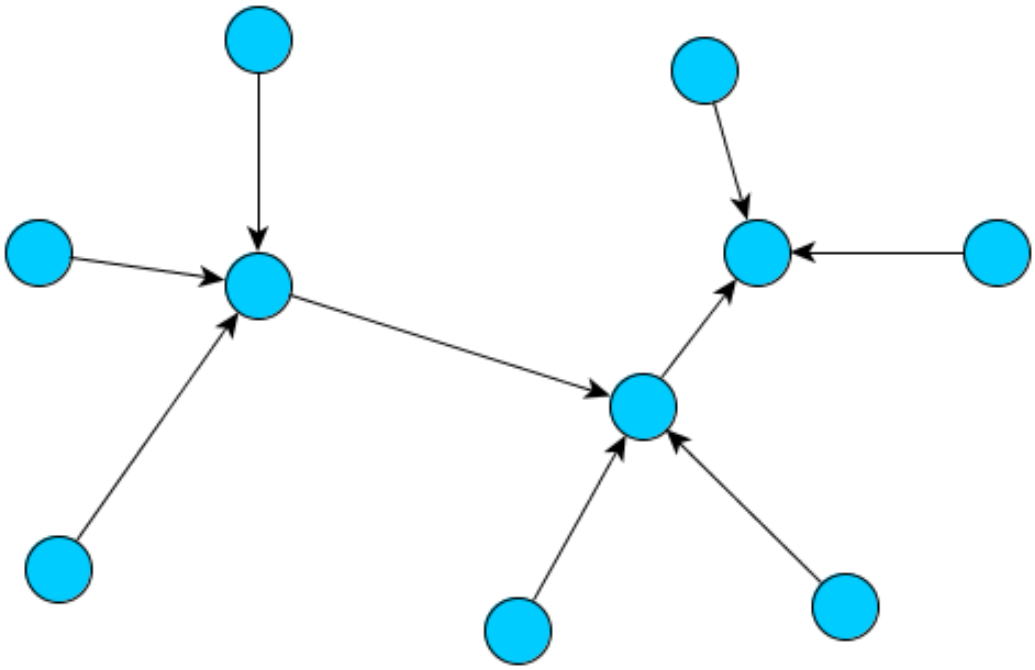


Figure 3.18: Wi-Fi-mesh topology. The direction of the arrow represents the direction in which connections are made [25].

The mesh of boards could be connected to the access point in the way indicated by figure 3.19. Three nodes are programmed to be Bridge nodes (see section 3.2.2.4), and are connected to the AP while they accept incoming connections from other boards. The other nodes are called Mesh nodes (see section 3.2.2.5), and are not connected to the AP (they might as well be out of reach of it). These nodes connect to other nodes to form the mesh. This way, the 9 ESP32s from the image can communicate with the central computer while only having three nodes connected directly to the router.

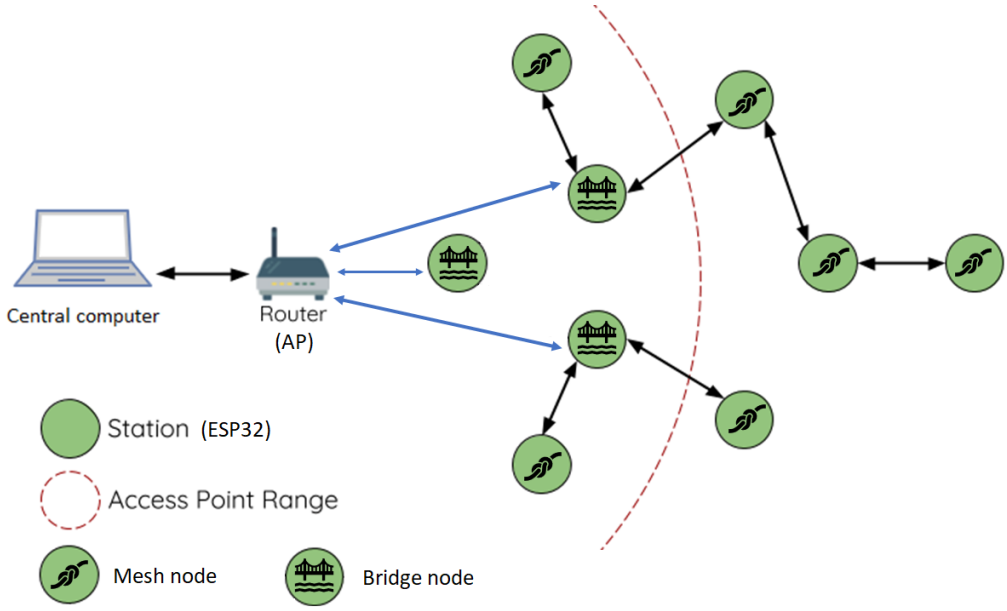


Figure 3.19: Wi-Fi-mesh topology. The direction of the arrow represents the direction in which connections are made. In blue we have the Bridge Connections, and in black, the Mesh Connections [26].

A Wi-Fi mesh should be a reliable way of transmitting data. A previous study [27] determined that in a mesh of 16 ESP8266 boards, the maximum delay (defined by the paper as the time that a message takes to leave the sending node and arrive at the destinatary node) was no higher than 100 ms. Moreover, the study found that a node can receive up to 461 messages/second and 28 messages/second for a payload (message size) of 10 bytes and 4400 bytes, respectively. It also observed that "sending messages with a payload greater than 4400 bytes results in broken and incomplete messages." Considering the message rate and the message size that we chose for our implementation, we were confident that a mesh of ESP32 boards would do a good job, knowing that they have better hardware capabilities in terms of memory and CPU than ESP8266s, and that those are the characteristics that influence the most the performance of the mesh network [27].

3.2.2 Implementation

3.2.2.1 Overview

We decided to implement the Wi-Fi-Mesh architecture using the PainlessMesh library. PainlessMesh is a library that allows to create a simple mesh network using the ESP32 hardware, without having to worry about how the network is structured or managed. According to the documentation, any system of one or more nodes will self-organize into a fully functional mesh, and the maximum size of the mesh is probably limited by the amount of memory in the heap that can be allocated to the sub-connections buffer (which should be relatively high). In our case, creating a mesh of 180 nodes should be easily feasible, as suggested by [25].

3.2.2.2 Programming paradigm

PainlessMesh is compatible with the "arduino-esp32" library, which is a port of the ESP-IDF library (the official library from Espressif to develop ESP32 firmware) to Arduino's environment. It implements most of Arduino's libraries, and makes them compatible with ESP32.

Many libraries are compatible with arduino-esp32. The FreeRTOS library is implemented inside arduino-esp32 directly. FreeRTOS is a real-time operating system that allows to implement basic operating system features in an ESP32 firmware. These include tasks, queues and semaphores. The Taskscheduler library implements cooperative multitasking, which is a multitasking style in which there are no preemptions [28]. Each task is responsible for yielding the processor periodically.

The chosen logic to implement the architecture is with two separate tasks. The first one is the main task contained in the loop method. The second one is a task defined with Taskscheduler that broadcasts the temperature values every 5 seconds. The cooperative scheduling paradigm is chosen because it is recommended by Painlessmesh developers. Classical, preemptive FreeRTOS tasks may cause problems, like CPU starvation of other tasks, or delays that are too long and might cause interference among tasks.

3.2.2.3 Setup of the board

In the firmware's setup function, the board begins by reading the first byte of the permanent memory using the *EEPROM* library. This byte contains the mode in which the ESP32 should boot. If the read value is 0, it should boot in BLE mode (see section 3.2.2.6). If the read value is 1, it should boot in Mesh mode (see section 3.2.2.5). If the read value is 2, it should boot in Bridge mode (see section 3.2.2.4). If none of these values are read, it means that the permanent memory has not been accessed yet (like in brand new boards for example). In this case, we write a 1 and reboot the board. Once the booting mode has been determined, the ESP32 performs several additional set-up tasks, which depend on the chosen mode. In the following sections we explain with more details the differences in booting between Mesh mode, Bridge mode and BLE mode.

3.2.2.4 Bridge nodes

The bridge nodes are in charge of linking the central computer with the rest of the mesh through a direct connection with the AP. After setting up the I^2C connection (see section 3.1.5), the Bridge initializes the GPIO of the BOOT button. This will allow engineers to use the BOOT button as defined in the loop function (see section 3.2.2.8). It also initializes a 1-byte long FreeRTOS queue (see section 3.2.2.11) with *xQueueCreate()* that keeps track of the status of the station-mode Wi-Fi connection (see section 2.1.1.6). Then, the board reads out the permanent memory to check which Wi-Fi credentials (SSID and password) it should use to connect to the AP, as well as to the central computer (see section 3.2.3). If there is nothing written on it, the board uses the default credentials hard-coded in the firmware. Then, the ESP32 starts the mesh routine with the function *mesh.init()*. This method takes as parameters the mesh credentials (a mesh SSID and a mesh password), a mesh TCP/IP port, a channel number (figure 2.2), a maximum number of boards the mesh can accept, the Wi-Fi mode (see section 2.1.1.6), and an optional pointer to a method containing a routine to be implemented in a non-preemptive task. We set the frequency to Wi-Fi channel 6, so that we don't have any interference between the mesh network

and the WLAN. We set the Wi-Fi mode to "Station and AP" because we are booting in bridge mode, and we set the optional method as our method in charge of processing the temperatures (see section 3.2.2.10). This method, called *taskSendMessage*, is defined with the TaskScheduler library [28], and is executed every 5 seconds, as desired. We also define a callback function with *mesh.onReceive()*, called *receivedCallback*, to process the messages that are received from the mesh. The initialization routine will look for a mesh with the defined mesh credentials in its surroundings. If it can't find a mesh, it creates one. If it finds a mesh, it connects to it. Boards connect to the closest node they find according to a measurement of their signal strength [25]. To finish the setup, the board launches the *taskSendMessage* routine, and uses the method *mesh.stationManual(SSID,Password)* to connect to the AP. It also initializes the TCP client, which will establish the connection with the central computer's application (see section 3.2.3.1).

3.2.2.5 Mesh nodes

The initialization of the Mesh nodes is very similar to the one of the Bridge nodes. The only difference is that we don't initiate the connection to the AP, nor we set up the TCP client.

3.2.2.6 Bluetooth configuration menu ("BLE mode")

Overview The Bluetooth configuration booting mode is launched when the board reads a 0 at the first byte of the permanent memory. This mode launches a Bluetooth Low Energy (BLE) server. In BLE, the Generic Attribute Profile (GATT) defines the way in which two devices interact. The ESP32 will run a GATT server with the "Write" characteristic. A GATT client can connect to the ESP32 and use this service to send commands to the ESP32. It can send at most 20 bytes in a run, so longer commands can take several runs to complete. An example of GATT client can be a mobile phone with a client app installed in it.

The BLE mode is used to configure the SSID, password and IP address of the central computer. These values must be saved into the permanent memory, so that when the board reboots (by pressing the RESET/EN button for example), it can retrieve those values at the initialization step. We designed a small protocol to achieve this task using a GATT client.

Protocol Commands Two commands can be sent to the GATT server:

- **SSID,Password,IP**: The client sends the SSID of the AP, the corresponding password, and the IP address of the central computer inside the Local Area Network (LAN), separated by a comma. Writing into the permanent memory is done with the help of a cursor, which indicates the byte at which we have to start writing. By convention, the cursor starts at byte 10 of the permanent memory. If the provided credentials are larger than 20 characters, the server listens for several sendings of 20-byte blocks. Each block is written into the memory, and after writing it, we place the cursor one bit after the position of the last character. When all characters have been written, the board adds a "\0" character to signalize the end of the string.

- **erase**: This keyword erases the permanent memory. After erasing all bytes, the board re-writes the Boot Mode, and places the credential's cursor to byte number 10 again.

3.2.2.7 Telnet server

The boards configured as Bridges implement a Telnet server (RFC854 [29]), programmed with the ESPTelnet library [30]. This telnet server listens to TCP port number 23 for incoming connections from a device which is connected to the same LAN as the Bridge. The server listens for three types of commands:

- **ping**: This command pings the server, who returns "pong" immediately. It is useful to check if the communication is working.
- **send**: This command tells the Bridge node to broadcast a *send*-type of message (as defined in section 3.2.2.10).
- **cmd**: This command tells the Bridge node to boot into a new mode. The word *cmd* is followed by the delimiter ":" and the new boot mode. When the node executes this command, it automatically reboots to the new mode.

3.2.2.8 Loop of the program

The loop of the program is the main part of the architecture's implementation. The first part of the loop contains the routine that reads the state of the boot button if the board is in Bridge or Mesh mode. It checks whether there is a long or short press at any moment. The second part of the loop is in charge of maintaining the multiple network connections. It runs the *update()* method from PainlessMesh, which performs several background tasks to maintain the network. If the board is in Bridge mode, it additionally updates the TCP client that is connected to the central computer, and the telnet server.

Then, the board reads the value inside the Wi-Fi connection queue, which is a 0 or a 1. A value of 1 means the board is connected to another board of the mesh (if the board is in Mesh mode) or to the access point of the WLAN (if it is in Bridge mode). A 0 means there's no Wi-Fi connection, so it will wait for one minute until there is one. If the board can't connect to the network or to the AP during that minute, it reboots. If the board looses its connection to the network after that first minute has passed, it tries to reconnect during 1 minute. If it is unsuccessful reconnecting, it reboots. This usually solves any connection problems right away. If a Bridge node looses its connection to the central computer server, it tries to reconnect to it once every 5 seconds.

3.2.2.9 Node ID

The ID of a node is a 32-bit unsigned integer number generated by PainlessMesh's *getNodeId()* method. It is uniquely calculated from the Media Access Control (MAC) address of the node, and serves to identify each of the 180 boards. The MAC address is a Link-layer address of the Internet Protocol which is stored in the Read-Only memory of the board's Network Interface Controller (the piece of hardware that allows the ESP32 to connect to a computer network). Thanks to this address, it is not necessary to manually hard-code the ID of a node. It can be computed from its hardware.

3.2.2.10 Messaging

Message structure

Temperature message A message can be of two different types. The first one is the *temperature message*. Its structure is a JSON data type with the following structure (figure 3.20):

- **temp1** temperature of sensor IC5
- **temp2** temperature of sensor IC4
- **temp3** temperature of sensor IC3
- **error** error code (used for debugging)
- **node** ID of the node that built the message

The method in charge of broadcasting this information is the *taskSendMessage* method, defined when the mesh was initialized. Before this object is sent through the network, it has to be converted into a string by the Arduino_JSON library [31]. The length of the information sent is of about 35 bytes. The length of the strings that have to be added to correctly format the JSON data type (”“, ”“ and ”“ symbols, as well as the object's keys) is about 55 bytes. This gives a total length of around 90 bytes for the whole string. If the broadcasting node is a Bridge node, it sends the message directly to the central computer's server with its TCP client (see section 3.2.3). If it is a Mesh node, it uses the PainlessMesh's *Broadcast* method to send the message to every board of the mesh. This method ensures that the message arrives to every single board of the mesh while avoiding loops. Whenever a message is broadcast, the library sends as function parameters the ID (see section 3.2.2.9) of the original node that wrote the message, and the message itself. The callback function receives these two values as well.

The method in charge of receiving these messages is the callback function *receivedCallback*. If the board receiving the message is in Bridge mode, it will route the message towards the central computer (see section 3.2.3). If the board is in Mesh node, it ignores the message.

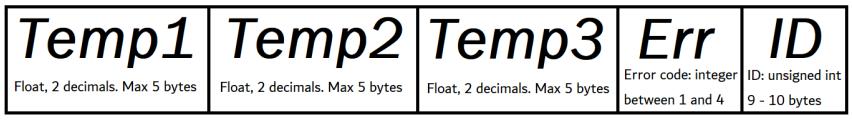


Figure 3.20: Diagram of the message.

Target message The second type of message is the *target message*. This message is a string that starts with the keyword ”send”. It uses the delimiter ”:” to separate the information it contains after the ”send” word. The first piece of information is the ID (see section 3.2.2.9) of the node to which the message is addressed. The second piece of information is the new boot mode the destinatary node should boot into.

In order to generate a target message, a telnet client must connect to the telnet server of a Bridge node. The client must send a correctly-formatted target message. The Telnet server routine will proceed to broadcast the message. The callback functions of the other nodes of the mesh will decode the message, and check if it is destined to them. If it is, they will reboot into the new mode specified by the message. If not, they will ignore it.

3.2.2.11 Wi-Fi queue

The Wi-Fi queue is an operating system queue defined by FreeRTOS. It contains a 0 or a 1 if the board is connected to an access point (which can be a router or another board) or not, respectively (figure 3.21). When a board is initialized, a 0 is pushed into the queue to indicate there's no Wi-Fi connection. There are two Wi-Fi events that the firmware listens to. The first one is the Wi-Fi station connection event. When a board connects to an AP or to another board (depending on the mode, Bridge or Mesh, that it has booted into), the first method is executed. This method pops the 0 from the queue and pushes a 1. The second one is the Wi-Fi station disconnection event. When a board disconnects from the AP or from another board, the second method is executed. This method pops the 1 from the queue, and pushes a 0. This way, the queue knows exactly if the board has a station connection or not, at any moment.

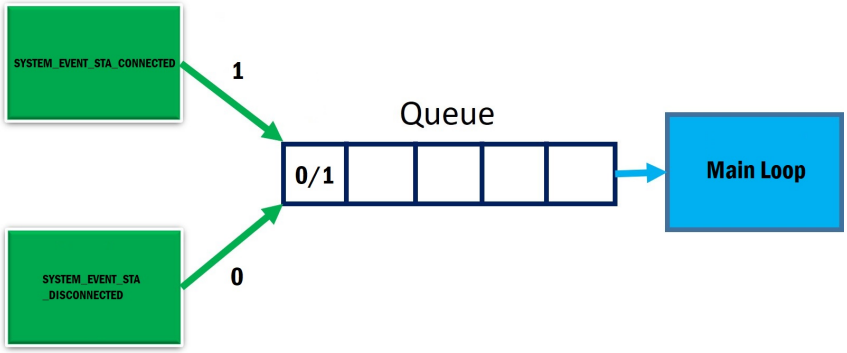


Figure 3.21: Diagram of the queue [32]. The two Wi-Fi events to which the board listens at all times (SYSTEM_EVENT_STA_CONNECTED for connection as a station, and SYSTEM_EVENT_STA_DISCONNECTED for disconnection as a station) interact with the queue by pulling the number present in the queue's only byte, and pushing a 1 or a 0. The main loop periodically checks the value of this queue.

3.2.2.12 Permanent memory

The permanent memory is used to store the values that have to be kept even after re-booting the ESP32. The values that are kept in the permanent memory are:

- Boot mode: A pointer to the first byte of the permanent memory stores a value of 0 (for Bluetooth mode), 1 (for Mesh node mode) or 2 (for Bridge node mode).
- Wi-Fi Credentials: The AP's SSID, the password, and the IP address of the central computer inside the WLAN are stored from byte number 10 on-wards.

JUNO will run for a total of 6 years without interruptions. It is important to not exceed 100000 P/E cycles as explained in section 2.1.1.3, although this is extremely unlikely if the credentials don't have to be changed often (in the order of several times a day).

3.2.2.13 Boot Button

The boot button is in charge of switching the board's boot mode. We defined two types of button pressing. Short-pressing, when the button is pressed for less than 250 ms, and

long pressing, when the button is pressed for more than 250 ms. The short press switches the boot mode from Mesh mode to Bridge node, or vice-versa. The long press changes the booting mode from Mesh or Bridge mode to Bluetooth configuration, but not the opposite. The board immediately reboots into the new mode after the button is pressed.

3.2.3 Central computer communication

The central computer's main task is to get the messages sent by the bridges, decode them and save them in a database. We now discuss how we implemented a system capable of achieving this goal in a reliable and fast way, using some available protocols and technologies.

3.2.3.1 Server application

Overview In order to receive data from a device through Wi-Fi, a server must listen at a TCP port at the application level. The application layer protocol chosen to establish the communication is Message Queuing Telemetry Transport (MQTT) [33] (figure 3.22).

MQTT MQTT is a publish-subscribe network protocol that transports messages between devices. A publish-subscribe protocol is a type of network architecture where senders of messages (called publishers) categorize published messages into classes without knowledge of which receivers (called subscribers) there may be. Subscribers express interest in one or more classes of messages, and only receive the messages that are of their interest. The action of expressing interest for one class of message is called "subscribing", and the action of sending a message of a certain class to the server is called "publishing". Subscribers are not aware of which publishers there are, they are just connected to a server which sends them the messages of the classes they are subscribed to. MQTT servers are often called "message brokers" or "brokers". To publish or receive messages, clients only need to know the hostname (or IP address) and the TCP port of the broker. Moreover, MQTT works asynchronously, which means that tasks are not blocked while waiting for a message or publishing a message. For all these properties, we decided to use MQTT in our implementation.

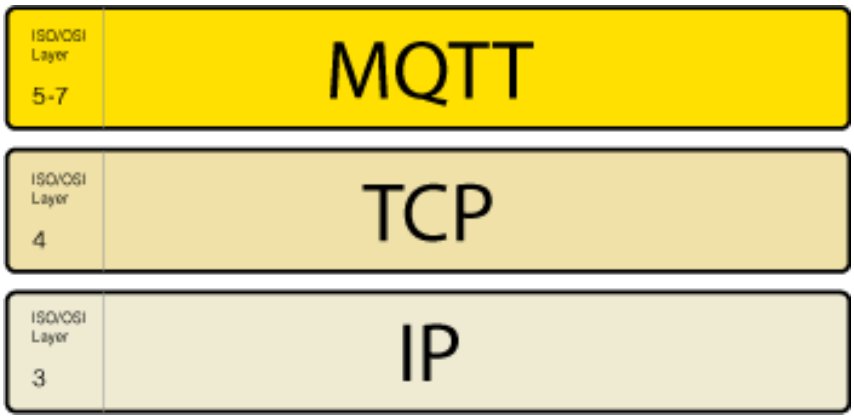


Figure 3.22: MQTT inside the Internet Protocol layers [34].

The broker is at the heart of any publish/subscribe protocol. It is responsible for receiving and filtering all the messages, as well as determining who is subscribed to which

message, and sending the messages to the corresponding subscribed clients [34]. The server chosen to run on the central computer is Mosquitto [35]. Mosquitto is an open source message broker that implements the MQTT protocol. It is lightweight, and it is suitable for use on all devices from low power single board computers to full servers running Windows or Linux. It implements the MQTT protocol versions 5.0, 3.1.1 and 3.1. The registered TCP port for MQTT use over a non-TLS (Transport Layer Security) channel is 1883.

The Quality of Service (QoS) is an agreement between the sender of a message and the receiver of a message that defines the guarantee of delivery for a specific message [34]. There are 3 QoS levels in MQTT. QoS level 0 is the minimal level (figure 3.23). The broker sends the message to the client, but the recipient does not acknowledge receipt of the message. The broker doesn't store and re-transmit it in case of failure, which means this service level doesn't guarantee delivery. QoS level 1 guarantees that a message is delivered at least one time to the receiver (figure 3.24). The sender stores the message until it gets a "PUBACK" (Publish Acknowledge) packet from the receiver that acknowledges receipt of the message. Finally, QoS level 2 is the highest level of service in MQTT (figure 3.25). It guarantees that each message is received only once by the intended recipients. The guarantee is provided by at least two request/response flows (called a four-part handshake) between the sender and the receiver. This makes QoS 2 the safest (but slowest) quality of service level available.

Figure 3.23: QoS level 0 [34].

Figure 3.24: QoS level 1 [34].

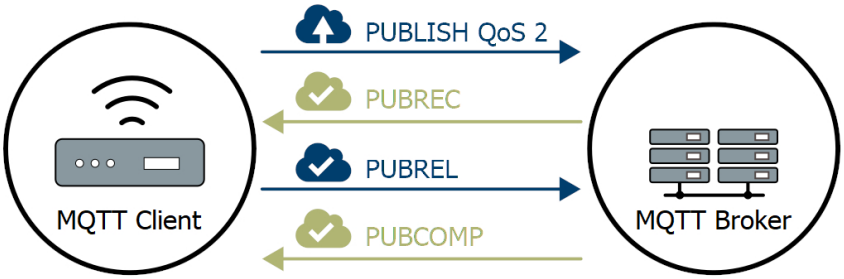


Figure 3.25: QoS level 2 [34].

Benchmarking In order to get an idea of how many connections and messages per second our locally deployed Mosquitto broker could handle, we used the MQTT benchmarking tool *mqtt-bench* [36]. We configured the program to run with 5 clients connected to the broker, to use 80-byte long messages, and to use a QoS level of 0. The rate of flow obtained was of about 35000 messages per second, which is close to the value found in [37]. With this performance, we can confidently use this broker in a scenario where it receives messages from 5 Bridges connected to 180 boards, with all boards transmitting their temperatures every 5 seconds.

3.2.3.2 Topics

In MQTT, a topic refers to an UTF-8 string that the broker uses to filter messages for each connected client. The topic consists of one or more topic levels, and each topic level is separated by a forward slash (called topic level separator). The broker accepts each valid topic without any prior initialization. When a client wants to publish something, it creates a topic of choice, and sends a packet to the broker with the topic and the corresponding message (which contains the temperatures).

3.2.3.3 Client

In the whole implementation of the retrieval system, there are two types of MQTT clients. One of them is the Bridge node client, which publishes the messages under the appropriate topics to the broker. The second one is the Node-RED software, which subscribes to the relevant topics to log the messages into the database.

3.2.3.4 Bridge client

The Bridge nodes run an MQTT client software based on the PubSubClient library for arduino [38]. It supports the latest MQTT version 3.1.1 (which makes it compatible with Mosquitto), and it only supports publishing at QoS level 0. It handles the implementation of callback functions with a default message size of 256 bytes, which is enough to hold our temperature messages.

Once the ESP32 Bridge is connected to the AP and is assigned an IP address by DHCP, it tries to connect to the MQTT server. If the connection fails, it tries again every 5 seconds until it succeeds. Once the board is connected, it starts publishing the messages with the topic *esp32mesh/nodeID* (where *nodeID* is replaced by the node's ID). PubSubClient uses the *loop()* method to run several maintenance tasks needed for it to function. This method is called inside the loop of the program so it executes periodically.

Outgoing and incoming message buffers are controlled by this method, which waits for packets to come in, and processes them afterwards. It also pings the server to maintain a *keep-alive*, a link designed to prevent the connection from dropping, of 15 seconds [39] (figure 3.26).

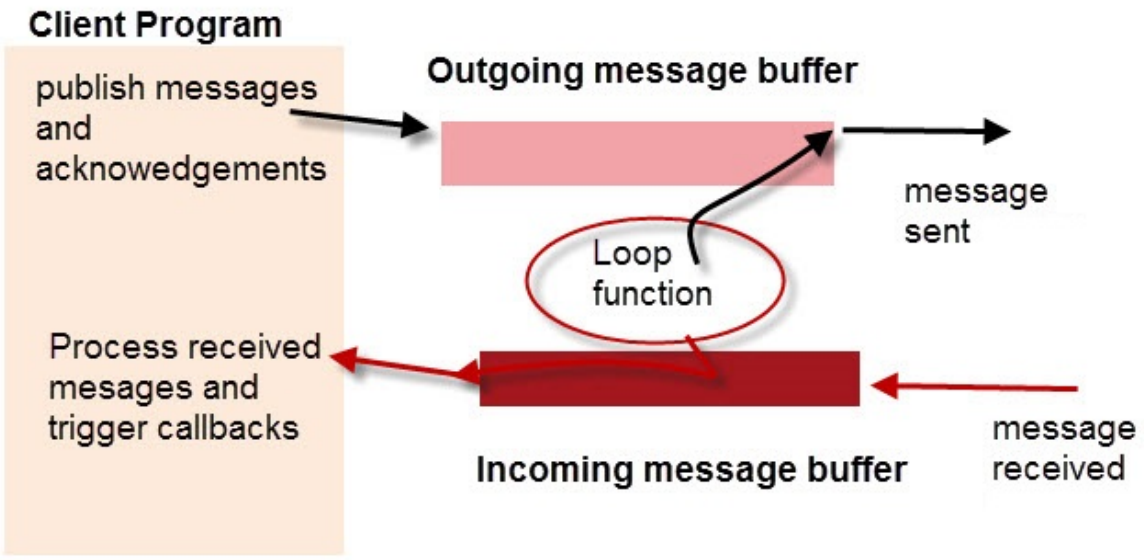


Figure 3.26: Loop function illustration [40]. Outgoing and incoming message buffers are managed by the loop function. It interacts with messages to publish and with received messages and callback functions (if any).

Bridge nodes don't implement any callback function, and don't receive any messages. They only publish their own temperature values every 5 seconds, and the temperatures received from other boards.

3.2.3.5 Node-RED Flow

Overview Node-RED is a visual programming tool developed originally by IBM for linking together hardware devices, application programming interfaces (APIs) and online services as part of an Internet of Things (IoT) implementation. This tool makes use of Javascript objects, which receive inputs and generate outputs in the scope of a program flow. They can be programmed to perform several tasks, including interacting with MQTT servers or with databases.

Node-RED uses messages to convey information between objects. Messages are also JavaScript objects that can have properties set on them. The choice of the properties that a message can have is free, but most nodes work with the *payload* property, or the *topic* property. Node-RED also adds a property called *_msgid*, which uniquely identifies each message. As we can see, in order to build a flow that achieves our goal, we have to carefully read the documentation of each Javascript Object, so we can know which properties need to be used, as well as which data each property must carry.

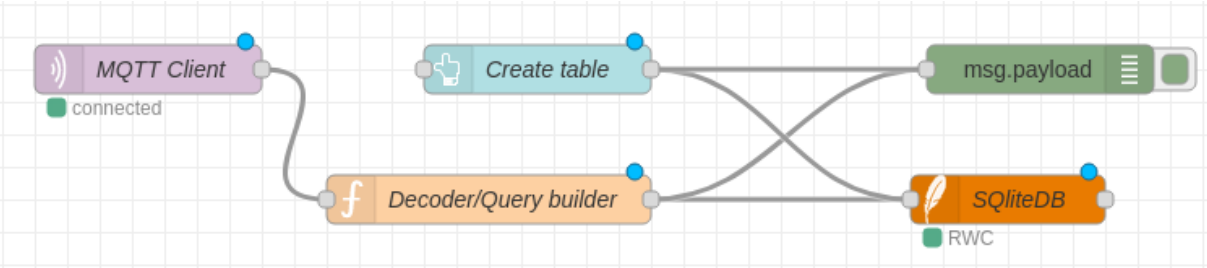
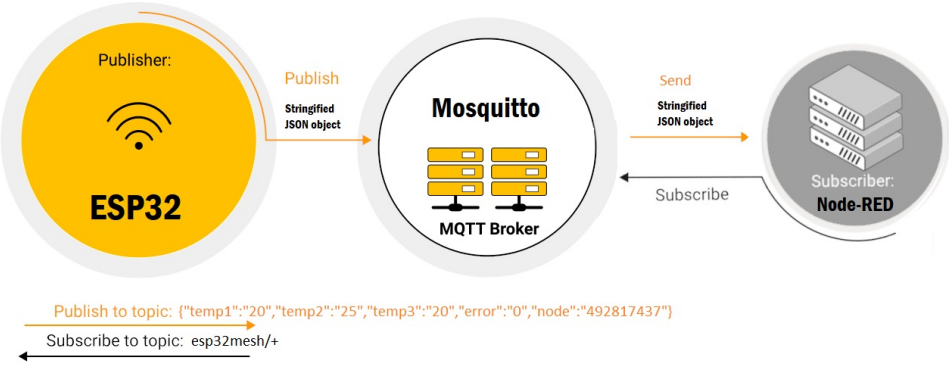


Figure 3.27: Node-RED flow diagram [16].

Database The database chosen to store all the messages is an SQLite database. SQLite is a relational database management system (RDBMS) contained in a C library. It is not a client–server database engine, which means the entire database is stored as a single file on a computer. It implements most of the features of the SQL-92 standard for SQL. SQLite tables include a hidden "rowid" index column on each row, which acts as an Integer Primary Key column. Integer Primary Keys uniquely identify each row of the table, which is useful for data analysis and processing. Moreover, several processes or threads can access the database concurrently, which can speed up the analysis if working with a multi-core system. An illustration of how rows are stored can be seen in figure 3.28.

msgDate	temp1	temp2	temp3	errorCode	boardNB
4/9/2021, 4:57:38 PM	29.0	20.0	22.0	3	492817185
4/9/2021, 4:57:38 PM	20.0	19.0	28.0	3	492813085
4/9/2021, 4:57:38 PM	18.0	25.0	23.0	3	492814525
4/9/2021, 4:57:38 PM	25.0	19.0	24.0	3	492709681
4/9/2021, 4:57:38 PM	20.0	22.0	17.0	3	492817421
4/9/2021, 4:57:38 PM	24.0	27.0	23.0	3	492817561
4/9/2021, 4:57:38 PM	19.0	26.0	20.0	3	492817581
4/9/2021, 4:57:39 PM	22.0	15.0	29.0	3	492817437
4/9/2021, 4:57:39 PM	26.0	20.0	27.0	3	492814833
4/9/2021, 4:57:39 PM	15.0	17.0	18.0	3	492814801
4/9/2021, 4:57:39 PM	22.0	24.0	25.0	3	492817137
4/9/2021, 4:57:39 PM	20.0	16.0	26.0	3	492817529
4/9/2021, 4:57:39 PM	27.0	23.0	16.0	3	492814869

Figure 3.28: Rows of the database after a test, visualized with SQLite Viewer.



Flow design

- **MQTT Client:** In purple, there is an MQTT client subscribed to the MQTT topic "esp32mesh/+". The "+" is the wildcard used to match a single level of hierarchy. This means the client is subscribed to all nodes, because each node publishes with the topic *esp32mesh* followed by its node ID at the second level in the hierarchy (see section 3.2.3.4). Its Quality of Service (QoS) is set to 2 (as Mosquitto server supports it). The node automatically converts the message string that is received from the broker into a JSON data type, and outputs a JavaScript object with a payload property that contains the message in JSON format. This object is fed into the next node, the *Decoder/Query builder* node.

- **SQliteDB:** This object creates an SQLite database when the flow is deployed. The query to be executed is received via the *topic* property of the message object from the *Decoder/Query builder* node, or the *Create table* node.

- **Create table:** This node builds the query necessary to create a table inside the SQLite database. It sets the message parameters as required by the database object: the "payload" property is set as the timestamp at which the query was built, and the "topic" property is set as the query. Then, it sends the message to the SQliteDB node.

- **Decoder/Query builder:** This object executes a JavaScript function that takes as input the JavaScript object containing the JSON message from the boards, and outputs a query that logs the message into the database. The function starts by getting the date and time at the instant it receives the message in the format "month/day/year, hour:minute:second AM/PM". Then, it gets the three temperature values, the error code and the board ID from the message object. It proceeds to build a query (figure 3.29), and it inserts a new tuple into the table of the database. This query is sent to the SQLite database object as the "topic" property of the output message.

- **msg.payload:** This object is used to print debug messages on Node-RED's development environment.

```
"INSERT INTO tableTemp(msgDate,temp1,temp2,temp3, errorCode, boardNB) VALUES ("month/day/year, hour:minute:second AM/PM", temp1, temp2, temp3, errorCode, nodeID);"
```

Figure 3.29: Query used to log the messages into the SQLite database.

GUI Node-RED offers a Graphical User Interface (GUI) in the form of an HTML page. It can contain useful objects like buttons, charts or real-time videos. In our implementation, we decided to add a button to the page that would execute the node *Create table* when clicked. We also added some basic charts to observe the temperature of a certain board as a function of time (figure 3.30).

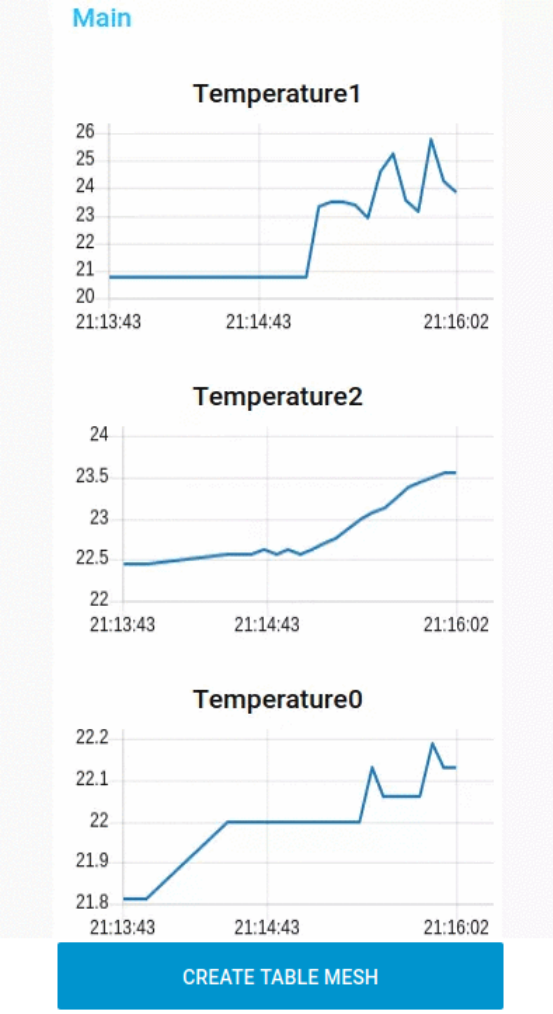


Figure 3.30: Temperature charts for the three sensors, and the button to create the table inside the database.

3.2.3.6 Summary of the implementation

The final implementation can be summarized with the diagram in figure 3.31. We see the two MQTT clients (ESP32 and Node-RED) interacting with the MQTT server. There is a one way connection between the ESP32 and Mosquitto, in which the client publishes the messages containing the temperatures with a QoS of 1. There is also a two way connection between Mosquitto and Node-RED, in which the client subscribes to topics, and the server sends the corresponding messages using a QoS of 2.

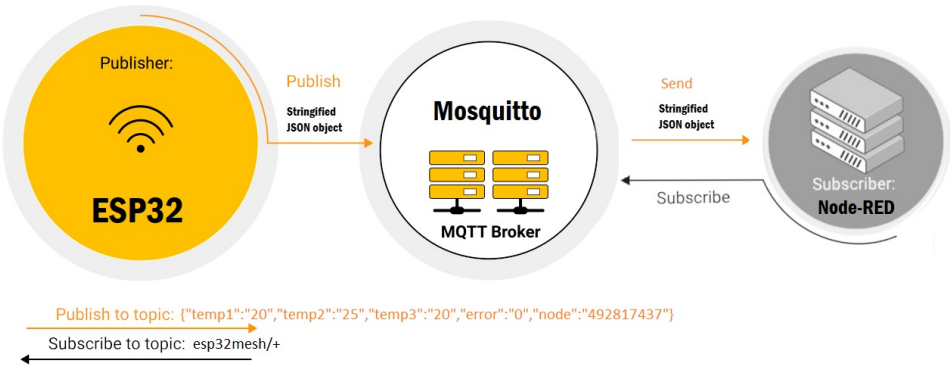


Figure 3.31: Diagram of the whole MQTT client/server, publish/subscribe architecture.

Chapter 4

Validation of the solution

This chapter reviews a series of experiments that helped broadening our knowledge about the measures we took and the performance of our implementation. We start by describing the experiments that were carried out to determine the relationship between the real temperature of the equalizers and the temperature measured by the sensors, as well as the evolution of these temperatures over time. Then, we studied the capacities and limitations of the Wi-Fi connections that could be made with our hardware. To finish, we tested the performance of our implementation of Wi-Fi mesh. Experiments were done to learn about the limitations of the network first. Then, we measured its actual performance when it comes to message retrieval.

4.1 Temperature monitoring test

4.1.1 Introduction

Some pertinent temperature monitoring tests were performed to observe how the temperature values retrieved with the ESP32 evolved in time, and also to compare them with those measured with a thermocouple directly on the equalizers. The sensors monitor the temperature of the silicon at the point where they are attached. There is however a thin (8 mm) layer of air between the silicon and the equalizers, which might introduce differences between both measurements (see figure 4.7):

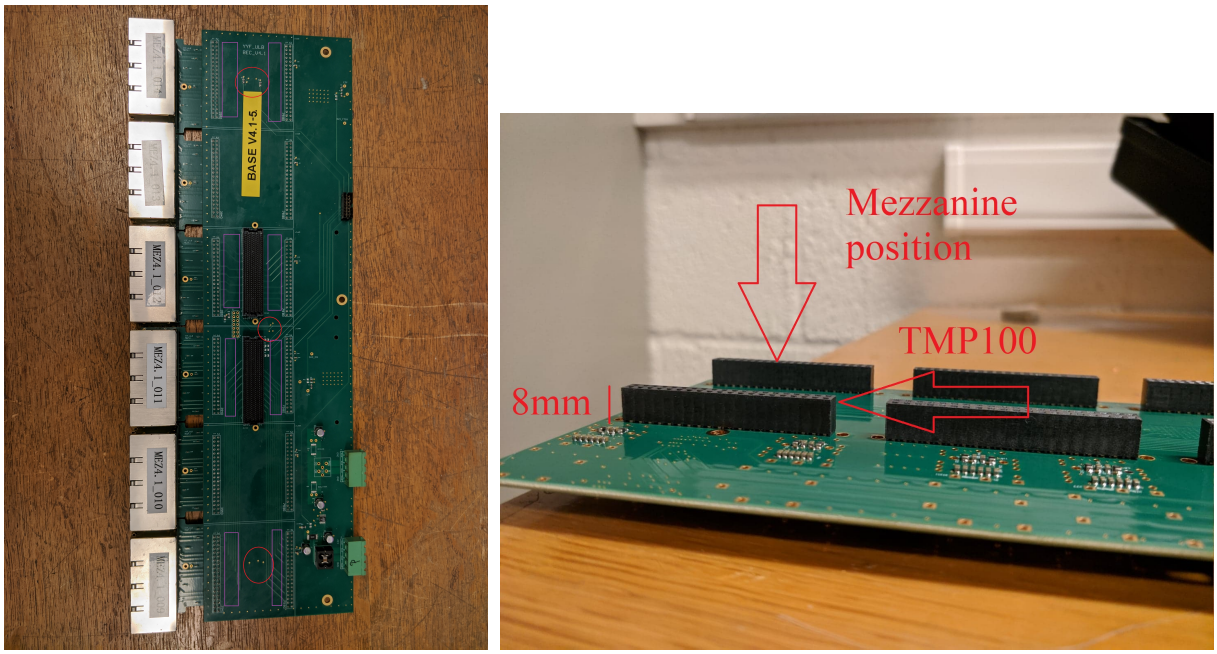


Figure 4.1: BEC - upper view (left) and side view (right). Sensors, mezzanines and the 8 mm distance between the baseboard and the mezzanine positions are shown.

4.1.2 Continuous monitoring test

This test was performed to make sure that the temperature monitoring was stable during a period of 5 days. The ESP32 was flashed with the firmware as a Bridge and plugged into the 3V and GND pins to the BECs power supply. The SDA and SCL lines were also plugged to ESP32's pins 21 and 22. The configuration of the different elements inside the box can be seen in figure 4.2. Fan number 3 was disconnected, and fans 1,2,4 and 5 were operational. The box was closed and placed on a rack, which tends to level the temperatures of fans 1 and 3. Fans 1, 3, 4 and 5 push air inside the box, while fan 2 pulls air outside of the box.

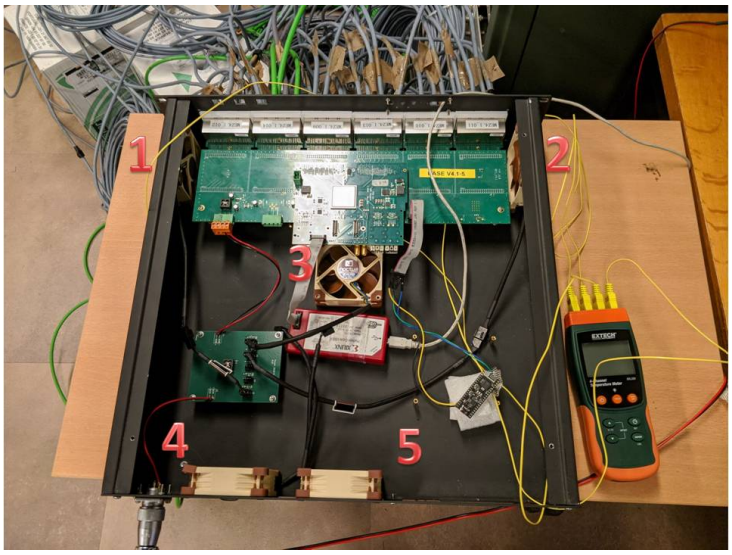


Figure 4.2: Aluminium box configuration with fan numbering.

We obtained the diagram of temperatures monitored by the three sensors (see figure 4.3).

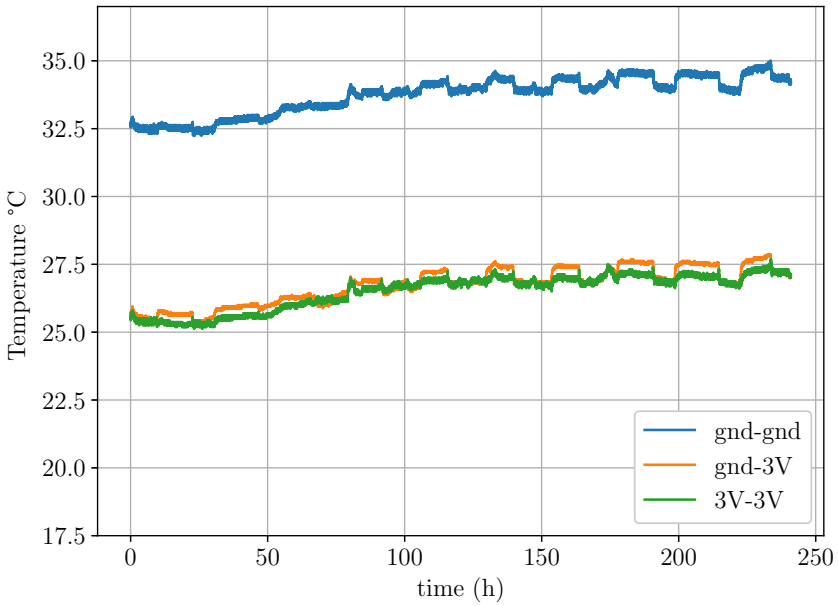


Figure 4.3: TMP100 temperature monitoring. IC5 = GND-GND, IC4 = GND-3V and IC3 = 3V-3V.

As we can see, the IC5 sensor is about 7°C warmer than the two other sensors, IC3 and IC4, which are very close in temperature. There is a temperature pattern that seems to go up and down many times, which is due to changes in temperature in the lab room, the building and outside.

The temperature seems to stabilize towards the end of the experiment. It increases slightly at an approximate rate of $\frac{1}{3}$°C per day, but this increase is caused by changes in the weather outside the lab, and changes in the air extraction schedule of the lab room.

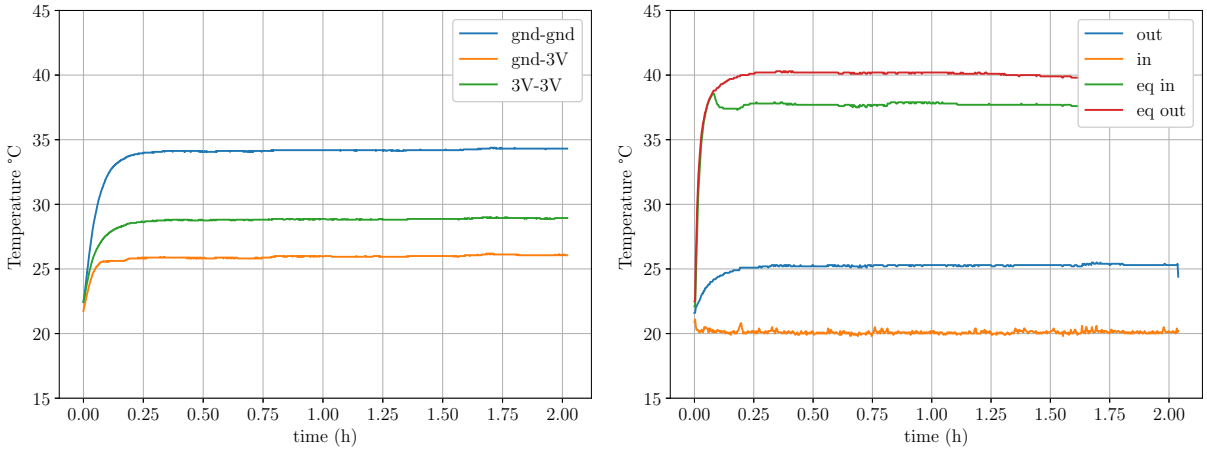
4.1.3 Fan configuration test

In order to examine the disparities between the temperatures of the sensors and the ones of the equalizers, we directly measured the temperature at four spots with a thermocouple (placed at the right of figure 4.2). The first spot, called *in*, is at the outside of the aluminium box, where the air enters fan number 1. The second spot, called *out*, is also at the outside of the box, where the air exits fan number 2. The third spot, *eq in*, is placed on the equalizers just under IC4. The fourth spot, *eq out*, is placed on the equalizers just under IC3.

We ran two experiments with two different fan configurations. The duration of the experiments is of two hours, as we empirically determined it was enough time for the system to become stable.

4.1.3.1 Fans 1 and 2 ON

In the first experiment, we disable all the fans except the two lateral fans (1 and 2 in figure 4.2). This way, we expect to cool IC3 and IC4 the most, while IC5 would be the hottest sensor, as it receives the least airflow.



(a) TMP100 temperatures.

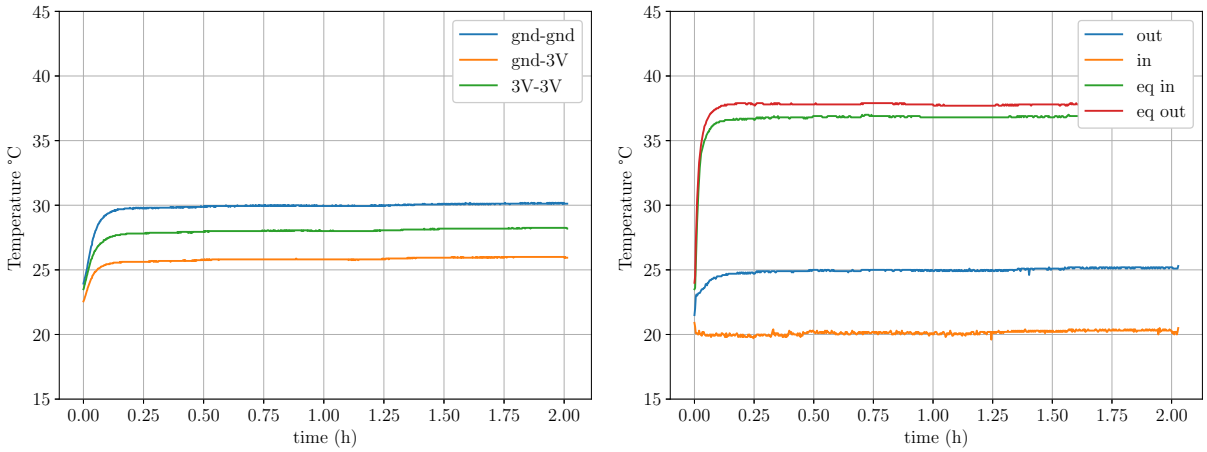
(b) Thermocouple temperatures.

Figure 4.4: Thermocouple and ESP32 temperatures. In pannel (a), the different correspondence between label and position on the BEC is the following : GND-GND = IC5 (center), 3V-3V = IC3, GND-3V = IC4. Fpr pannel (b) : *out*: temperature at fan 2. *in*: temperature at fan 1. *eq in*: temperature at equalizers under IC4 (near fan 1). *eq out*: temperature at equalizers under IC3 (near fan 2).

As we can see in figure 4.4, the system stabilizes after 25 minutes. The TMP100 sensors read a lower temperature than the one the thermocouple placed at the equalizers reads. IC4 sensor indicates a temperature 12°C lower than the one reported by *eq in*. IC3 has an 10°C lower temperature than the one reported by *eq out*. Moreover, the temperature of IC3 is 3°C higher than the one of IC4, due to the fact that fan 2 is 3°C hotter than fan 1 according to the thermocouple measurement of *in* and *out*. This is different than in the previous test of section 4.1.2, where both temperatures were very close. It can be explained by the fact that the current experiment was done with the box placed on a table instead of on a rack, which allows for a better airflow. IC5 is 5°C and 8°C hotter than IC3 and IC4 respectively, as it has the worst airflow of the three due to the geometry of the BEC.

4.1.3.2 Fans 1, 2 and 3 ON

In the second one, we disable all the fans except for the two lateral fans and the central fan (1, 2 and 3 in figure 4.2). This way, we expect to cool IC3 and IC4 the most, while IC5 would still be the hottest sensor, even though it will receive more airflow thanks to fan number 3 compared to the previous experiment.



(a) TMP100 temperatures.

(b) Thermocouple temperatures.

Figure 4.5: Thermocouple and ESP32 temperatures. In pannel (a), the different correspondence between label and position on the BEC is the following : GND-GND = IC5 (center), 3V-3V = IC3, GND-3V = IC4. Fpr pannel (b) : *out*: temperature at fan 2. *in*: temperature at fan 1. *eq in*: temperature at equalizers under IC4 (near fan 1). *eq out*: temperature at equalizers under IC3 (near fan 2).

In figure 4.5, we observe the same trends as in the previous experiment. However, the *eq in* and *eq out* equalizers are almost at the same temperature (36.5°C and 37.5°C respectively), and the difference between IC4 and IC3 is of 2°C instead of 3°C like before. Moreover, IC5 indicates a 4°C lower temperature than when fan number 3 was turned off. This allows to conclude that the third fan helps cool all the equalizers, particularly the central ones.

4.2 Maximum message rate test

This experiment aimed to observe how the message rate affected the stability of the system, and how quickly and reliably the mesh network could be established. We used the Bluetooth configuration mode to change the message rate of the boards with a smartphone, one at a time. Once all the boards were configured to use the same message rate, we launched a fake temperature retrieval test with one bridge. With the database analyzer script, we checked how many boards were connected in the last 10 seconds in real time. We tried to establish a mesh network that could retrieve the temperatures for a few minutes without interruption. Here is a table defining some reliability scores that we used to qualify the experiments (table 4.1).

Score	Description
1	The mesh network was established in a few seconds, and the temperatures were transmitted correctly for a few minutes
2	The mesh network was established after trying several times, and the temperatures were transmitted with some lag for a few minutes.
3	The mesh network couldn't be established after several tries, and the temperatures couldn't be transmitted.

Table 4.1: Connection and reliability score description.

In table 4.2 we show the results obtained after testing a mesh network of 7 and 14 boards, with only one bridge each. As we can see, with a group of 7 boards, we can reliably establish the network with a rate of two messages per second. For a group of 14 boards, we can establish the network with a rate of 1 message per second. If this relationship is linear, we could theoretically establish a reliable group of 70 boards with one bridge that functions at the desired rate of one message every 5 seconds.

We believe that the message rate capacity of the network decreases linearly with the number of nodes forming it, because the ability of a bridge to process messages and send them to the central computer is entirely dependent on the hardware capacities of the ESP32. More boards connected means more messages to treat per second if the rate is kept constant. Thus, to maintain the desired stability, we need to proportionally lower this rate.

Regarding our implementation, to form a group of 180 boards, we would need 3 bridges at least to ensure proper functioning. However, we should use more than three in case one or more bridges fail to function for any reason. Moreover, to avoid a Bridge node getting too many boards connected to it compared to the other bridges, we need to place the bridges apart from each other to cover as many boards as possible.

Number of boards	Message rate (messages per second)	Score
7	1	1
	2	1
	10	2
	20	2
	40	3
	100	3
14	1	1
	2	2
	4	3
	10	3

Table 4.2: Score in function of the message rate and the number of boards in the network.

4.3 Range tests

These experiments aimed to determine the maximum distance at which the ESP32s could transmit their temperatures without losing the connection. The concrete walls separating the rooms are about 20 cm thick, and they might also influence the reliability of the connection. We performed this test in the lab, which has the following shape (see figure 4.6).

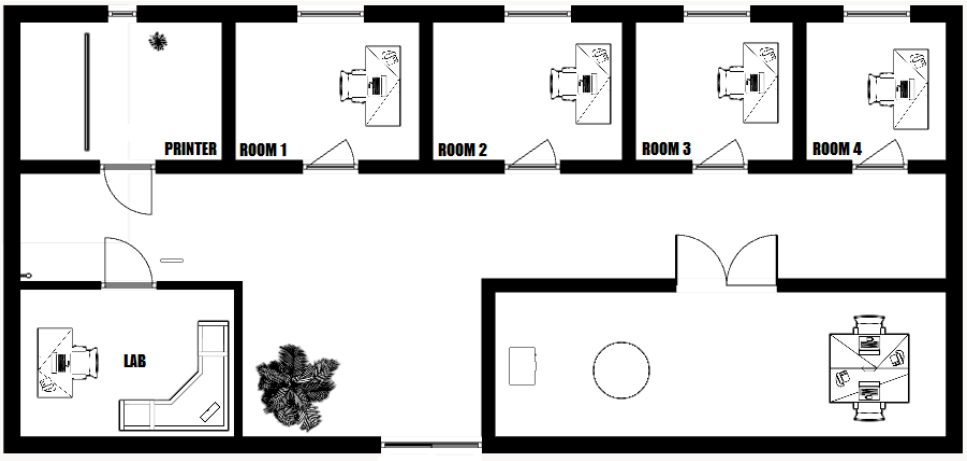


Figure 4.6: JUNO lab - diagram representation.

4.3.1 Wall Test

We placed ESP32s in the lab room, the printer room, room 1 and room 2. The one in the lab was in Bridge mode, and the others in Mesh mode. There are two concrete walls between the lab room and the printer room, and one concrete wall between the printer room and room 1, and room 1 and room 2. The exact placement can be visualized in figure 4.7.

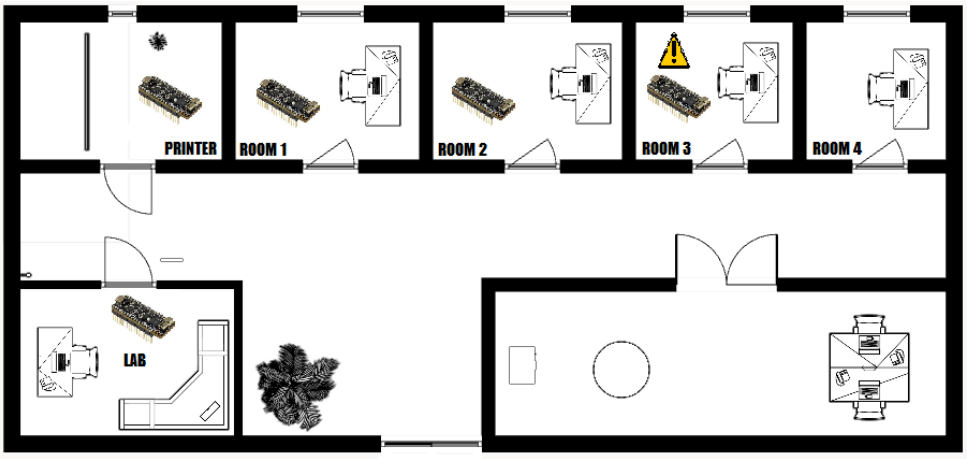


Figure 4.7: ESP32s placement for the Wall Test.

We observed a stable connection for several minutes without interruption in this configuration. However, when we moved the board from room 2 to room 3, it couldn't connect anymore to any node. This leads us to think that it wouldn't be a good idea to have more than one concrete wall between the ESP32s.

4.3.2 Bridge node to Mesh node Test

We placed one ESP32 outside of the lab, near its door. This board was configured as a Bridge. A second board, configured as a Mesh board, was plugged to a portable battery, and progressively moved away from the Bridge until it got disconnected from it. We found that the maximum distance before it disconnected was of 18 meters (see figure 4.8).

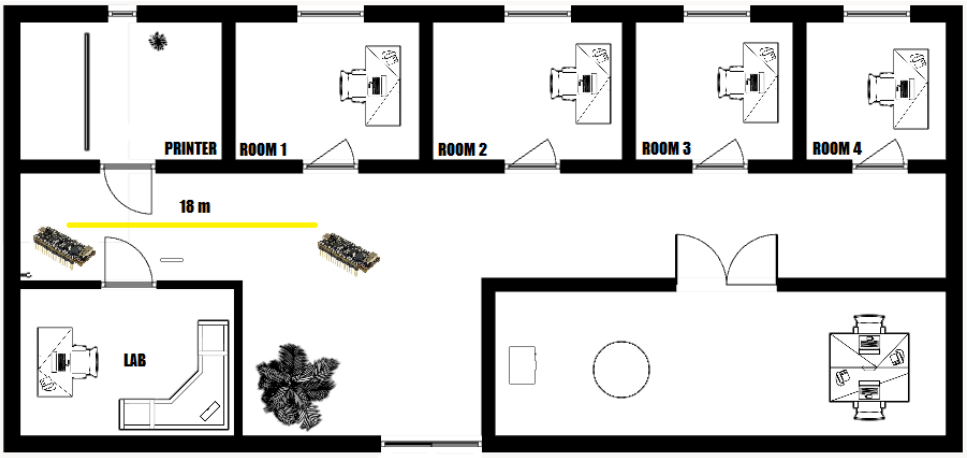


Figure 4.8: ESP32s placement for the Bridge-Mesh Test.

4.3.3 Mesh node to Mesh node range

We placed a first ESP32 configured as a bridge node inside the lab room and a second ESP32 configured as a mesh node outside the lab near the door. A third board, configured as a mesh node, was plugged to a portable battery, and progressively moved away from the

mesh node outside of the room until it got disconnected from it. PainlessMesh connects each mesh node to the node with the best signal quality there is around. The fact that the bridge node is inside the lab room and further away from the corridor means that the third board is connected to the second board. We managed to place it at a distance of 60 meters until it lost its signal (see figure 4.9).

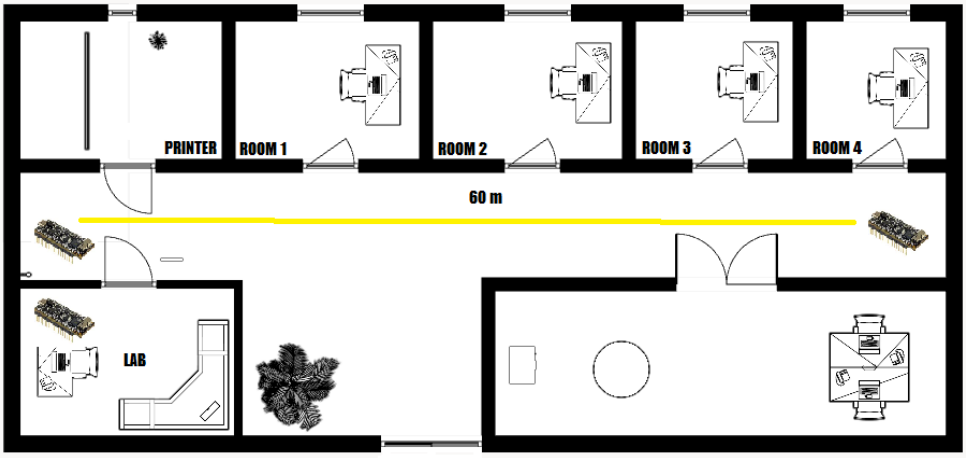


Figure 4.9: ESP32s placement for the Mesh-Mesh Test.

4.3.4 Results discussion

As we can see, the range between two Mesh nodes is three times longer than the range between a Bridge node and a Mesh node. This is probably due to the fact that the Bridge node is connected to two networks at the same time (the WLAN and the mesh network), and only has one antenna to keep both connections. The amount of power the antenna receives is limited, and therefore, the distance it can reach on a single frequency is shorter.

4.4 Small scale test

4.4.1 Overview

The small scale test is a set of tests that was conducted with less than one sixth of the ESP32 boards we would need to implement this in JUNO. It allows us to study the reliability and stability of the network in conditions closer to the ones we are interested in. We describe how the experiments were carried out, the results we obtained, and the conclusions we can draw from them.

4.4.2 Set-up

There are 30 boards available in the lab. To set up the experiment, we assigned the number 0 to the board present inside the aluminium box. The rest of the boards were numbered with a unique USB cable each, starting from 1 and increasing until 29. We flashed them one at a time with the Arduino IDE by plugging them to the computer. Immediately after flashing one board, we opened its serial interface, and noted the board's node ID, which was displayed in the console. Then, we unplugged it from the computer and plugged it to a

hub connected to a power source. There are a total of 5 power hubs, two of them powering 7 boards, one of them powering 6 boards, and the other two powering 4 boards (table 4.3). The last board is placed inside the aluminium box, and captures real temperatures. By following this procedure, we were able to flash all the boards relatively quickly (30 min), have all their ID's linked to their assigned number, and have the mesh network running. We must note however that it is not necessary to plug the boards one at a time into the hubs. They can be plugged at any moment, as the mesh will automatically build itself up with the nodes that are powered. We can see an image of the whole set-up in figure 4.10. The boards are pinned in groups to three soft-material plate, and two fans cool each plate to keep the board's temperatures under control. The boards were flashed with a slightly modified version of the firmware that instead of retrieving the sensor's temperatures, it sent random numbers between 15 and 30, representing random temperatures. This was done to simulate temperatures, as we didn't have 30 BECs available.

Hubs	IDs of the nodes
Hub1	2441135817, 492710013, 492710013, 492817533, 492814833, 492813081, 492814525
Hub2	492817581, 2441134253, 492817185, 492709681, 492817421, 492815001, 492817461
Hub3	492814869, 492817433, 492817529, 492817561
Hub4	492813085, 492817537, 492817065, 492817449
Hub5	492814845, 492814837, 492814801, 492817229, 492814989, 492817137, 492817125

Table 4.3: Boards identifiers (ID) (as a number obtained from their MAC address) contained by each power hub.

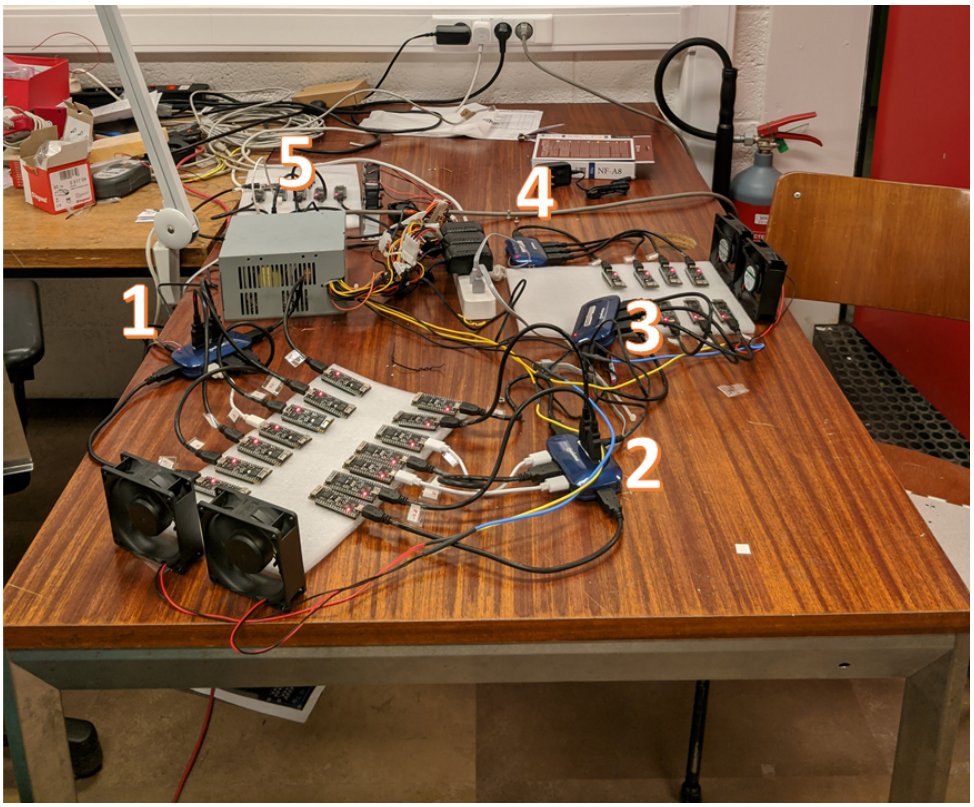


Figure 4.10: Overview of the set-up of the ESP32s.

4.4.3 Real time monitoring test

It is important to make sure that all the boards are connected to the mesh and broadcasting their values when launching the network. We implemented a python script to monitor in real time the state (connected or disconnected) of the boards. This script reads the entries from the end of the table and goes up to ten seconds in time by querying the database to check in real time if any board is absent from the rows for more than 10 seconds. If any board is missing, it is reported as *Disconnected*, and a message like *Board 492805861 was absent at time 2021-04-26 05:13:18* is printed in the console.

4.4.4 Performance analysis

The performance analysis aims to determine how reliable the mesh network is for a certain period of time. We have observed over the experiments that the boards' station connection (see section 2.1.1.6) tends to occasionally fall, causing a disconnection from the mesh network, as well as a loss of messages and delays in the transmission. In rare occasions, some boards spontaneously reboot due to software or hardware failure. This is the reason why in Chapter 3 we implemented a routine to force the board to reboot after two minutes of being disconnected.

We describe three approaches to detect and tackle the problem of loss of the station connection. The first one is to wait for PainlessMesh to reconnect the board again. The second one is to listen to Wi-Fi station disconnection events. If the event pops up, the board immediately reboots itself in the same mode it was in, and starts its usual routine to regain the network. This routine lasts for more than 5 seconds, so there will be some delay introduced in the message sending. The third solution is to listen for disconnection events, but wait for a period of time (2 minutes) before rebooting, to allow the library to reconnect the board.

We use several parameters in our tests to study the stability of the network. One of the parameters is *Discontinuity*. When a board produces a message, it adds to it a number to identify it, and increments it by one for each produced message. This number is kept in memory as a counter that is incremented each time a message is produced. However, if the board reboots, the memory is erased, and the counter starts over at zero. This phenomena allows the central computer to verify if at some point, the message number returns to a lower value than those of previous messages, and thus, detect a reboot event. Another parameter is the *Disconnection*. This number is incremented by one each time the board catches a Wi-Fi station disconnection event, and it is saved in the permanent memory to avoid losing its value if a reboot occurs. This parameter allows to keep track of the total number of disconnections during an experiment in a more accurate way. The third and fourth parameters are the *Percentage of Retrieved Messages* and the *Average Message Distance*. They are both calculated from the database. The former is computed by counting the total number of messages sent by a board, and compare it with the expected total number of messages that there should be in that timestamp. The latter is computed by calculating the average number of seconds there is between two messages of the same board.

4.4.4.1 Test 1 - Assessment test

This test was performed to get a first impression on how the network behaved. We let the system run for a day with the configuration parameters presented in table 4.4. We note

that there is only one bridge in this experiment, and that we don't perform any reboot if the Wi-Fi station disconnects. This is because we still don't exactly know if, and for how long, we should wait for a board that has lost its connection to reconnect.

characteristics	
Initial date	5/17/2021, 3:59:00 PM
Final date	5/17/2021, 6:17:47 PM
Total boards	27
Bridges	2441135817
Reboot time	No reboot, Painlessmesh autoconfig

Table 4.4: Information about the experiment.

Table A.1 shows that on average, more than 90% of the messages arrived at the computer, and that the average delay is of 5.34 seconds, a bit higher than the desired 5 seconds. Only 6 discontinuities were observed in total, which means the boards had almost no reboots. In the next experiment, we add two bridges to see if the system performs better.

4.4.4.2 Test 2 - Increased bridge test

This test was performed to determine if three bridges gave better results than one bridge. We let the system run for a day with the configuration parameters presented in table 4.5.

characteristics	
Initial date	5/25/2021, 7:49:03 PM
Final date	5/26/2021, 2:58:16 AM
Total boards	27
Bridges	2441135817,492817461,492814989
Reboot time	No reboot, PainlessMesh autoconfig

Table 4.5: Information about the experiment.

Unfortunately, the test stopped after a few hours. Table A.2 shows very bad results, with an average percentage of retrieved messages of 39.58%, and an average distance of 7.26 s. The extremely low rates of message transmission in some boards indicates that many of them disconnected from the network and never got back. Apparently, the lack of bridges wasn't the cause for the first test's poor performance. There seems to be a problem with the boards disconnecting and never getting back.

In the next experiments, we tried to determine the cause of this bad behaviour. We designed some tests to see if they were caused by hardware and software limitations, or by PainlessMesh itself.

4.4.4.3 Test 3 - Memory logging disconnections

This test aimed to determine if any disconnections took place during the experiment, as well as the cause of them and their influence over the stability of the network. We added a piece of code to save each Wi-Fi station disconnection event into the permanent memory. This way, even after a reboot event, the board keeps this information saved.

The characteristics of this tests can be seen in table 4.6. We reboot the boards right after any Wi-Fi station disconnection event to prevent the network from breaking down like before.

characteristics	
Initial date	5/24/2021, 10:25:28 PM
Final date	5/25/2021, 5:26:26 PM
Total boards	28
Bridges	2441135817,492817461,492817137
Reboot time after disconnection	Instantaneous reboot
Hubs containing bridges	1, 2 and 5

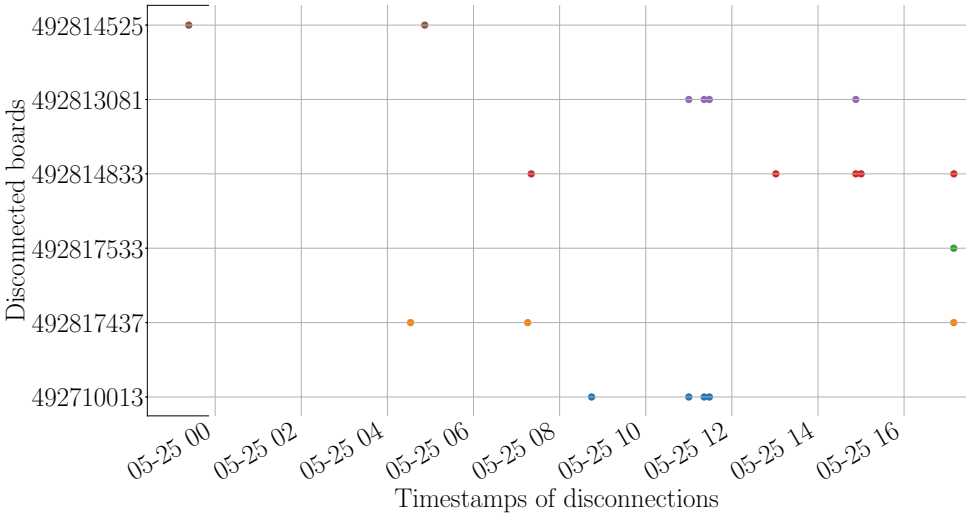
Table 4.6: Information about the experiment.

Table A.3 summarizes the results. The recovery percentage is 99.83%, which is very close to 100%. The average distance is 5.03 seconds. It is worth noting that bridges don't have a better recovery percentage than Mesh nodes. This probably means that the reason for some messages getting lost isn't related to a node being a Bridge or a Mesh node.

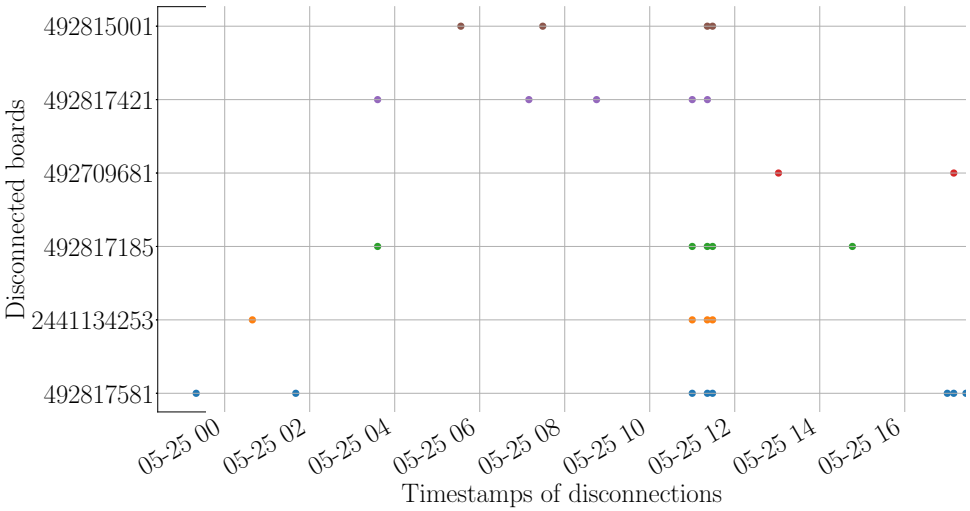
We observe the number of discontinuities and disconnections that took place during the experiment. Both are close to each other, but not always equal. Sometimes one is higher and sometimes one is lower. This means that the discontinuity parameter is probably not a reliable measure of the state of the network, as the factors that influence it can be multiple. These include reboots of the board due to Wi-Fi disconnections, reboots due to hardware or software failure, or multiple reboots in a short period of time which can lead to no discontinuity observed.

Surprisingly, the Wi-Fi station disconnections don't seem to have a significant effect on the total percentage of retrieved messages. This means that the disconnections are probably due in part to the fact that PainlessMesh reorganizes the mesh to keep it stable, and sometimes changes the station connection to another bridge. We observe in table A.4 that not all the boards stay connected to the same bridge for the whole duration of the experiment. Moreover, we observe that no disconnections take place in hub 5 (which contains one bridge). This hub is placed 60 cm apart from hubs 1 and 2 which also contain bridges. It seems that boards in hub 5 exclusively connect to the bridge in hub 5, and that boards in the other hubs share their connection time between the bridges in hubs 1 and 2. The conclusion we can extract from this is that PainlessMesh manages properly the network to maximize its stability, while minimizing the loss of messages, and that for doing so, it switches the station connection of some boards towards other boards when its necessary. PainlessMesh also optimizes the connections by assigning the closest bridges to each Mesh board when possible.

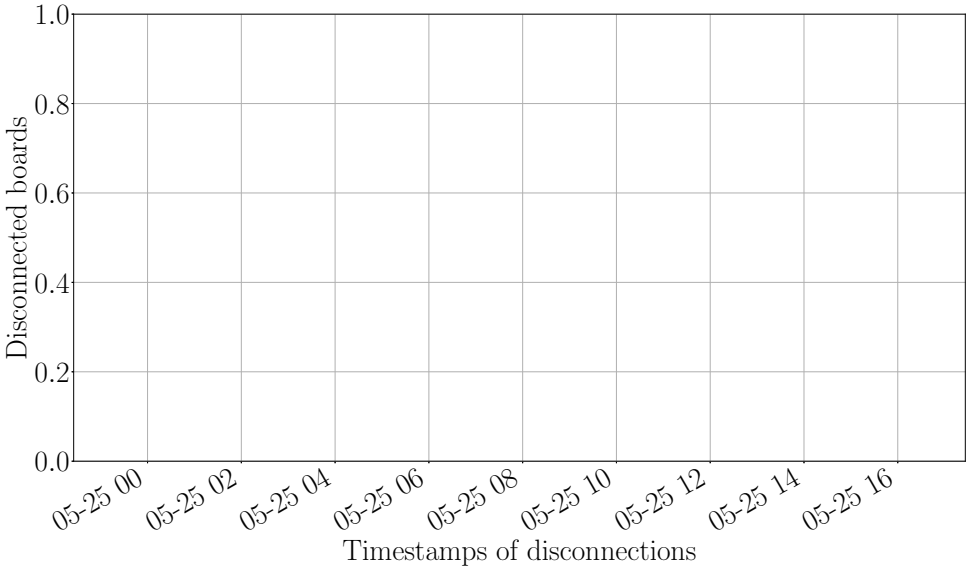
Now we analyze the timestamps at which each board disconnected in figures 4.11 and 4.12. We observe in several occasions that multiple boards of the same hub disconnect at the same time or close in time. Moreover, the distribution of the disconnections is not homogeneous. There are periods when almost no disconnections occur, and periods in which many disconnections occur in a row. Again, these disconnections are probably triggered by PainlessMesh as a part of its schedule to keep the network stable, and no significant loss of messages result from these disconnections according to our results.



(a) Hub 1, with bridge node in the first row from the bottom (2441135817).

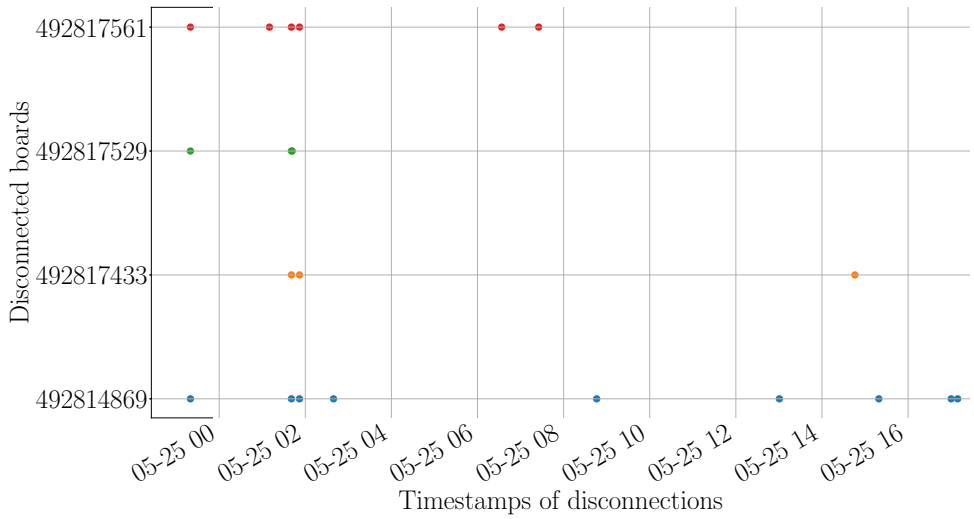


(b) Hub 2, with bridge node in middle row (492817461).

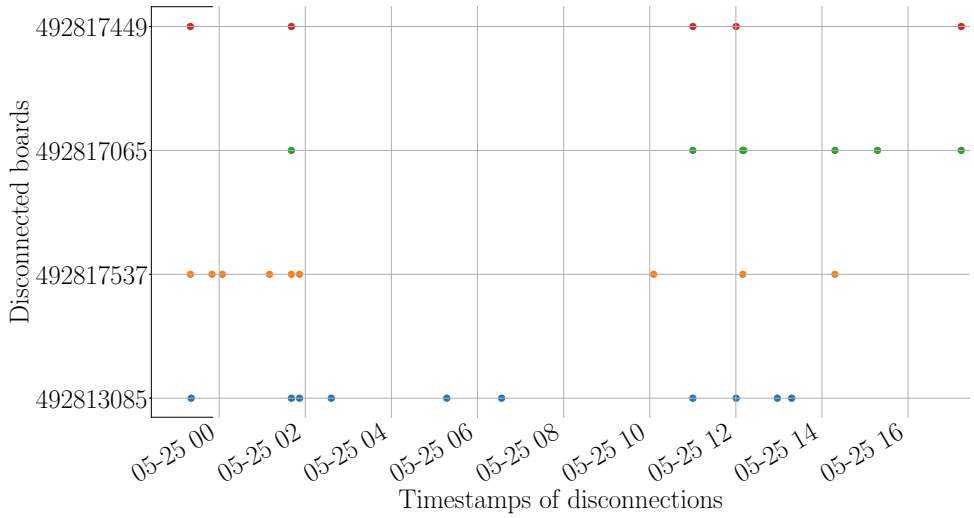


(c) Hub 5, with bridge node in the first row from the top (492817125). No disconnections observed.

Figure 4.11: Hubs containing bridges.



(a) Hub 3



(b) Hub 4

Figure 4.12: Hubs not containing bridges.

4.4.4.4 Test 4 - 2 minute reboot time test

This test aims to determine if a change in the reboot waiting time could yield an even better overall performance of the network. We set it to two minutes as indicated in table 4.7. This will allow PainlessMesh to reconfigure the network if necessary, but will force a reboot if that process takes too long.

characteristics	
Initial date	5/22/2021, 11:08:20 AM
Final date	5/23/2021, 11:08:20 AM
Total boards	27
Bridges	2441135817,492817461,492817137
Reboot time after disconnection	2 minutes
Hubs containing bridges	1, 2 and 5

Table 4.7: Information about the experiment.

Table A.5 shows the results obtained in this test. As we can see, they are not significantly better than in test 3 (see section 4.4.4.3). The percentage of retrieved message is very close to 99.8%, and the time distance between two messages is 5.01 s. However, these numbers are much better than the ones in the second experiment (see section 4.4.4.2), where no reboots took place. The fact that the difference between rebooting the boards immediately and waiting for 2 minutes to reboot them is insignificant means that PainlessMesh is able to quickly integrate nodes that join the network, and assign them optimal connections.

In table A.6, we analyze which boards connect to which bridges, and the amount of time they spend on each one. There is a Bridge in hub 1, hub 2 and hub 5. We recall that hubs 1 and 2 are close to each other, and that the fifth is 60 cm apart from the rest. We observe that boards in hubs 1 and 2 connected exclusively to bridges 1 and 3, which was expected. They seem to have spent the majority of the time connected to one of the two, but not the full time. Some of them spent 90% of the time on one of the two, and 10% on the other. Boards in hub 5 spent the majority of the time connected to the bridge in hub 5, with small occasional exceptions. This reaffirms that some sort of rearrangement of the connections takes place during the experiment, as we saw in the third experiment (see section 4.4.4.3).

A last observation that we made is that the distribution of the work among the three bridges is surprisingly close, with bridge 1 taking 33.67% of the work, bridge 2 taking 29.87%, and bridge 3 taking 36.44%. This confirms again that PainlessMesh tries to keep the network fairly distributed and stable.

4.4.4.5 Test 5 - Long duration test

The long duration test was designed to test the implementation for a longer period of time (more than ten days). In table 4.8 we summarize the characteristics of this experiment. We chose to reconnect the boards as soon as they lose their connection.

characteristics	
Initial date	4/15/2021, 12:00:00 AM
Final date	4/26/2021, 5:13:28 PM
Total boards	29
Disconnected boards	492805861
Bridges	2441135817,492817137
Reboot time after disconnection	instantaneous
Hubs containing bridges	1 and 5

Table 4.8: Information about the experiment.

Table A.7 shows the results obtained after running the network for eleven and a half days. We observe an average percentage of recovered messages of 99.06%. The average time between the receipt of two messages is 5.05 s, 1% higher than the desired 5 s rate.

4.4.5 Final thoughts on the test results

After testing different configurations, we can conclude that there exists disconnection events that affect all Mesh nodes. A part of these events are probably caused by PainlessMesh's maintenance routine, although this hypothesis needs further investigation in order to be

confirmed. However, if we let PainlessMesh run alone, the network stops working after a few hours, and yields very bad recovery percentages. The best way we found to keep the network functioning and to reduce the number of lost messages was to add as many bridges as possible, and to reboot the boards at most two minutes after they've lost their station connection. This technique offers a close to 100% message recovery over a larger time scale test of eleven and a half days, with 29 boards. A small proportion of messages are probably lost due to hardware and software limitations (something PainlessMesh's library already warns about). Care should be taken to chose a reliable router access point to minimize the number of times bridges lose their station connection.

Chapter 5

Summary

Let's conclude by reiterating the steps we took to achieve the goal of this project. In Chapter 1, we introduced JUNO. We saw an overview of how the detector works, which main parts it is made of, and the function of each of its electronic systems. We identified a potential problem, which was the possible overheating of the equalizers when they would be subject to workload for a period of 6 years. To solve this problem, we decided to monitor continuously the temperatures from a remote computer.

In Chapter 2, we introduced the hardware requirements to achieve our monitoring goal, as well as the choices we had to make. We needed temperature sensors placed near the equalizers, as well as some electronic device to read and transmit the read values. We chose TMP100s sensors and ESP32 motherboards for the capabilities they offer at an affordable price. Regarding the communication protocols, we chose $I2C$ to link the sensors to the ESP32 for its simplicity to wire multiple devices and acknowledge data receipt. We chose to transmit the values over the air instead of cables to avoid adding more cables to the electronic system.

In Chapter 3, we proposed a solution to the problem. We started by explaining in detail how I^2C was implemented in our electronic system, and how the temperatures were retrieved. After that, we presented our strategy for the temperature transmission based on a Wi-Fi mesh. We showed how a mesh network could be useful for our particular application, and which were the best tools and protocols to gather the data in an accessible object.

In Chapter 4, we presented some experiments to put our implementation to the test. Firstly, we logged the temperatures measured by the sensor for a period of 7 days, and observed that they remained mostly constant, while being at a predictable range from the temperatures measured by the thermocouple. Secondly, the wireless range of the ESP32s was assessed. Thirdly, a "small-scale" test was designed to measure certain parameters of the network. We made sure that the message rate didn't exceed the maximum rate that our hardware could support. Additionally, we analyzed the behaviour of the network for an extended period of time, and found out the percentage of messages that successfully arrived to the remote computer, as well as the average time they took to arrive.

There are additional experiments and investigations that can be done to improve this work and these findings. A conversion model between the thermocouples and the TMP100 sensors could be established to better track the temperatures remotely. More work could be done to better understand the origin of the board's disconnections, and to know what part of them comes from PainlessMesh, and what part comes from other factors. More detailed studies about Wi-Fi range and router stability would also be very useful.

Based on our conclusions from the experiments, we believe that the solution proposed in this work is suitable for the objectives we want to meet. Using TMP100s and ESP32s allows to keep track of the temperature of the equalizers after determining a conversion function between the measurements of the sensors and the ones of the thermocouple. The Wi-Fi mesh network implementation transmits the temperatures reliably if we accept a small percentage (less than 2%) of lost messages.

Appendix A

Table for the different tests

A.1 Test 1 - Assessment test

Hub number	Board ID	NB of discontinuities	% of retrieved messages (assuming 1 msg every 5s is 100%)	Average message distance (seconds)
1	2441135817	0	99.72%	5.0
1	492710013	0	91.24%	5.37
1	492817437	0	98.14%	5.09
1	492817533	0	89.44%	5.5
1	492814833	0	91.37%	5.36
1	492813081	0	97.79%	5.05
1	492814525	0	97.38%	5.07
2	492817581	0	92.82%	5.32
2	2441134253	0	90.75%	5.42
2	492817185	0	92.89%	5.3
2	492709681	1	96.55%	5.08
2	492817421	0	94.96%	5.24
2	492815001	0	99.45%	5.02
2	492817461	0	98.76%	4.98
3	492814869	0	83.92%	5.86
3	492817433	0	90.48%	5.45
3	492817529	2	86.47%	5.67
3	492817561	0	90.96%	5.41
4	492817537	0	90.13%	5.47
4	492817065	3	86.54%	5.7
4	492817449	0	85.37%	5.76
5	492814845	0	88.27%	5.57
5	492814837	0	92.89%	5.31
5	492814801	0	91.86%	5.37
5	492817229	0	93.86%	5.26
5	492814989	0	89.86%	5.48
5	492817137	0	95.1%	5.19

Total		6		
Average			92.55%	5.34

Table A.1: Discontinuities, percentage of retrieved messages, and average distance, for each board.

A.2 Test 2 - Increased bridge test

Hub number	Board ID	NB of discontinuities	% of retrieved messages (assuming 1 msg every 5s is 100%)	Average message distance (seconds)
1	2441135817	0	99.88%	5.01
1	492710013	0	97.85%	5.11
1	492817437	0	45.29%	5.23
1	492817533	0	68.34%	6.71
1	492814833	0	95.65%	5.23
1	492813081	0	45.25%	5.24
1	492814525	0	45.23%	5.24
2	492817581	0	4.76%	5.0
2	2441134253	0	97.86%	5.11
2	492817185	0	21.36%	8.94
2	492709681	0	89.63%	5.12
2	492817421	0	95.73%	5.22
2	492815001	0	70.26%	6.34
2	492817461	0	1.11%	5.21
3	492814869	0	4.76%	5.0
3	492817433	0	62.42%	7.35
3	492817529	0	4.76%	5.0
3	492817561	0	32.58%	14.08
4	492817537	0	15.36%	15.43
4	492817065	0	37.18%	6.37
4	492817449	0	15.4%	15.39
5	492814845	0	4.74%	5.02
5	492814837	0	27.82%	16.49
5	492814801	0	4.76%	5.0
5	492817229	0	27.86%	16.47
5	492814989	0	4.76%	5.0
5	492817137	0	4.72%	5.05
5	492817125	3	17.74%	5.19
Average			39.58%	7.26

Table A.2: Discontinuities, percentage of retrieved messages, and average distance, for each board.

A.3 Test 3 - Memory logging disconnections

Hub number	Board ID	NB of discontinuities	NB of recorded disconnections	% of retrieved messages (assuming 1 msg every 5s is 100%)	Average message distance (seconds)
1	2441135817	0	0	99.84%	5.01
1	492710013	5	7	99.85%	5.01
1	492817437	4	4	99.82%	5.01
1	492817533	1	2	99.91%	5.0
1	492814833	5	8	99.81%	5.01
1	492813081	6	4	99.83%	5.01
1	492814525	3	8	99.84%	5.01
2	492817581	8	8	99.77%	5.01
2	2441134253	5	4	99.84%	5.01
2	492817185	5	8	99.77%	5.01
2	492709681	2	3	99.86%	5.01
2	492817421	5	6	99.82%	5.01
2	492815001	5	4	99.83%	5.01
2	492817461	0	0	99.88%	5.01
3	492814869	9	11	99.69%	5.02
3	492817433	4	5	99.83%	5.01
3	492817529	3	6	99.87%	5.01
3	492817561	6	8	99.74%	5.01
4	492813085	11	19	99.61%	5.02
4	492817537	10	11	99.74%	5.01
4	492817065	9	16	99.69%	5.02
4	492817449	5	11	99.82%	5.01
5	492814845	0	0	99.92%	5.0
5	492814837	0	0	99.92%	5.0
5	492814801	0	0	99.92%	5.0
5	492817229	0	0	99.92%	5.0
5	492814989	0	0	99.92%	5.0
5	492817137	0	0	99.85%	5.01
Average				99.83%	5.01

Table A.3: Discontinuities, disconnections, percentage of retrieved messages and average distance, for each board.

Hub number	Board ID	Bridge 1 in hub 1 (2441135817)	Bridge 2 in hub 5 (492817137)	Bridge 3 in hub 2 (492817461)
1	2441135817	100.0 %	0.0 %	0.0 %
1	492710013	97.51 %	0.0 %	2.49 %
1	492817437	98.49 %	0.0 %	1.51 %

1	492817533	98.49 %	0.0 %	1.51 %
1	492814833	70.1 %	0.0 %	29.9 %
1	492813081	97.52 %	0.0 %	2.48 %
1	492814525	33.54 %	0.0 %	66.46 %
2	492817581	54.61 %	0.0 %	45.39 %
2	2441134253	54.44 %	0.0 %	45.56 %
2	492817185	66.14 %	0.0 %	33.86 %
2	492709681	21.69 %	0.0 %	78.31 %
2	492817421	57.76 %	0.0 %	42.24 %
2	492815001	4.73 %	0.0 %	95.27 %
2	492817461	0.0 %	0.0 %	100.0 %
3	492814869	38.67 %	0.0 %	61.33 %
3	492817433	5.75 %	0.0 %	94.25 %
3	492817529	4.87 %	0.0 %	95.13 %
3	492817561	10.24 %	0.0 %	89.76 %
4	492813085	47.06 %	21.87 %	31.07 %
4	492817537	49.05 %	16.54 %	34.41 %
4	492817065	53.86 %	6.29 %	39.84 %
4	492817449	53.85 %	1.08 %	45.08 %
5	492814845	0.0 %	100.0 %	0.0 %
5	492814837	0.0 %	100.0 %	0.0 %
5	492814801	0.0 %	100.0 %	0.0 %
5	492817229	0.0 %	100.0 %	0.0 %
5	492814989	0.0 %	100.0 %	0.0 %
5	492817137	0.0 %	100.0 %	0.0 %

Table A.4: Distribution of the percentage of messages transmitted by each Bridge, for each board.

A.4 Test 4 - 2 minute reboot time test

Hub number	Board ID	% of retrieved messages (assuming 1 msg every 5s is 100%)	Average message distance (seconds)
1	2441135817	99.91%	5.0
1	492710013	99.81%	5.01
1	492817437	99.87%	5.01
1	492817533	99.78%	5.01
1	492814833	99.78%	5.01
1	492813081	99.89%	5.01
1	492814525	99.93%	5.0
2	492817581	99.88%	5.01
2	2441134253	99.95%	5.0
2	492817185	99.9%	5.0

2	492709681	99.8%	5.01
2	492817421	99.94%	5.0
2	492815001	99.81%	5.01
2	492817461	99.91%	5.0
3	492814869	99.68%	5.02
3	492817433	99.63%	5.02
3	492817529	99.69%	5.02
3	492817561	99.62%	5.02
4	492813085	99.58%	5.02
4	492817537	99.59%	5.02
4	492817065	99.55%	5.02
4	492817449	99.57%	5.02
5	492814845	99.86%	5.01
5	492814837	99.94%	5.0
5	492814801	99.91%	5.0
5	492817229	99.9%	5.01
5	492814989	99.91%	5.0
5	492817137	99.91%	5.0
Average		99.8%	5.01

Table A.5: Percentage of retrieved messages and average distance, for each board

Hub number	Board ID	Bridge 1 in hub 1 (2441135817)	Bridge 2 in hub 5 (492817137)	Bridge 3 in hub 2 (492817461)
1	2441135817 (Bridge 1)	100.0 %	0.0 %	0.0 %
1	492710013	69.27 %	0.0 %	30.73 %
1	492817437	91.35 %	0.0 %	8.65 %
1	492817533	99.68 %	0.0 %	0.32 %
1	492814833	79.24 %	0.0 %	20.76 %
1	492813081	94.69 %	0.0 %	5.31 %
1	492814525	38.61 %	0.0 %	61.39 %
2	492817581	0.0 %	0.0 %	100.0 %
2	2441134253	100.0 %	0.0 %	0.0 %
2	492817185	85.1 %	0.0 %	14.9 %
2	492709681	15.73 %	0.0 %	84.27 %
2	492817421	100.0 %	0.0 %	0.0 %
2	492815001	9.02 %	0.0 %	90.98 %
2	492817461 (Bridge 3)	0.0 %	0.0 %	100.0 %
3	492814869	20.42 %	12.89 %	66.69 %
3	492817433	3.79 %	31.77 %	64.44 %
3	492817529	3.82 %	31.74 %	64.44 %
3	492817561	9.85 %	31.74 %	58.41 %
4	492813085	0.05 %	51.05 %	48.91 %

4	492817537	3.9 %	47.14 %	48.96 %
4	492817065	2.08 %	12.92 %	85.0 %
4	492817449	16.27 %	39.27 %	44.45 %
5	492814845	0.0 %	85.84 %	14.16 %
5	492814837	0.0 %	100.0 %	0.0 %
5	492814801	0.0 %	100.0 %	0.0 %
5	492817229	0.0 %	93.41 %	6.59 %
5	492814989	0.0 %	98.81 %	1.19 %
5	492817137 (Bridge 2)	0.0 %	100.0 %	0.0 %
Average		33.67%	29.87%	36.44%

Table A.6: Distribution of the percentage of messages transmitted by each Bridge, for each board.

A.4.0.1 Test 5 - Long duration test

Hub number	Board ID	% of retrieved messages (assuming 1 msg every 5s is 100%)	Average message distance (seconds)
1	2441135817 (Bridge)	99.49%	5.03
1	492710013	99.24%	5.04
1	492817437	99.26%	5.04
1	492817533	99.19%	5.04
1	492814833	99.22%	5.04
1	492813081	99.18%	5.04
1	492814525	99.07%	5.05
2	492817581	98.97%	5.05
2	2441134253	98.93%	5.05
2	492817185	98.97%	5.05
2	492709681	99.03%	5.05
2	492817421	99.05%	5.05
2	492815001	98.98%	5.05
2	492817461	99.03%	5.05
3	492814869	98.69%	5.07
3	492817433	98.75%	5.06
3	492817529	98.62%	5.07
3	492817561	98.73%	5.06
4	492813085	98.76%	5.06
4	492817537	98.72%	5.06
4	492817065	98.89%	5.06
4	492817449	98.74%	5.06
5	492814845	99.23%	5.04
5	492814837	99.36%	5.03

5	492814801	99.32%	5.03
5	492817229	99.29%	5.04
5	492814989	99.33%	5.03
5	492817137	99.33%	5.03
5	492817125 (Bridge)	99.4%	5.03
Average		99.06%	5.05

Table A.7: Percentage of retrieved messages and average distance, for each board.

Bibliography

[1] Fengpeng An et al. "Neutrino physics with JUNO". In: *Journal of Physics G: Nuclear and Particle Physics* 43.3 (Feb. 2016), p. 030401. DOI: 10.1088/0954-3899/43/3/030401. URL: https://doi.org/10.1088/0954-3899/43/3/030401.

[2] Barbara Clerbaux et al. *Automatic test system of the back-end card for the JUNO experiment*. 2020. arXiv: 2011.06823 [physics.ins-det].

[3] *IEEE 1588-2019 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. URL: https://standards.ieee.org/content/ieee-standards/en/standard/1588-2019.html (visited on 04/12/2021).

[4] *Jiangmen Underground Neutrino Observatory*. URL: http://juno.ihep.cas.cn/ (visited on 03/29/2021).

[5] D. Pedretti et al. "The Global Control Unit for the JUNO Front-End Electronics". In: Proceedings of International Conference on Technology and Instrumentation in Particle Physics 2017 (2018). Ed. by Zhen-An Liu, pp. 186–189.

[6] D. Pedretti et al. "Nanoseconds Timing System Based on IEEE 1588 FPGA Implementation". In: *IEEE Transactions on Nuclear Science* 66.7 (July 2019), pp. 1151–1158. DOI: 10.1109/tns.2019.2906045. URL: https://doi.org/10.1109/tns.2019.2906045.

[7] A. Aloisio et al. "The Reorganize and Multiplex (RMU) front-end trigger optical bridge for the JUNO experiment". In: *2017 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*. IEEE, Oct. 2017. DOI: 10.1109/nssmic.2017.8533064. URL: https://doi.org/10.1109/nssmic.2017.8533064.

[8] *ESP32-PICO-KIT V4 / V4.1 Getting Started Guide - ESP32 - — ESP-IDF Programming Guide latest documentation*. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-pico-kit.html (visited on 04/01/2021).

[9] *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. en-US. URL: /index.html.

[10] *espressif/crosstool-NG*. original-date: 2016-08-19T06:13:15Z. Mar. 2021. URL: https://github.com/espressif/crosstool-NG.

[11] *espressif/esp-idf*. en. URL: https://github.com/espressif/esp-idf.

[12] *espressif/arduino-esp32*. original-date: 2016-10-06T06:04:20Z. Apr. 2021. URL: https://github.com/espressif/arduino-esp32.

[13] *Comprendre la durée de vie du stockage flash*. fr. URL: https://www.ni.com/fr-be/support/documentation/supplemental/12/understanding-life-expectancy-of-flash-storage.html.

[14] *Wi-Fi Driver - ESP32 - — ESP-IDF Programming Guide latest documentation.* URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html#wi-fi-protocol-mode.

[15] *File:2.4 GHz Wi-Fi channels (802.11b,g WLAN).svg.* en. URL: https://en.wikipedia.org/wiki/File:2.4_GHz_Wi-Fi_channels_(802.11b,g_WLAN).svg.

[16] *ESP32 Useful Wi-Fi Library Functions (Arduino IDE) — Random Nerd Tutorials.* en-US. URL: https://randomnerdtutorials.com/esp32-useful-wi-fi-functions-arduino/.

[17] Ralph Droms. *Dynamic Host Configuration Protocol.* en. URL: https://tools.ietf.org/html/rfc2131.

[18] *TMP100 data sheet, product information and support — TI.com.* URL: https://www.ti.com/product/TMP100#design-development#all (visited on 04/13/2021).

[19] Scott Campbell — DIY Electronics — 52. *Basics of the I2C Communication Protocol.* en-US. Feb. 2016. URL: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/ (visited on 04/03/2021).

[20] *What is an Open Collector or Open Drain in a Transistor ?* URL: http://www.learnerswings.com/2014/06/what-is-open-collector-or-open-drain-in.html (visited on 04/03/2021).

[21] *I2C - learn.sparkfun.com.* URL: https://learn.sparkfun.com/tutorials/i2c/all (visited on 04/03/2021).

[22] *myOMRON Europe: Services & Support.* URL: https://www.myomron.com/index.php?action=kb&article=1581.

[23] *Arduino - Wire.* URL: https://www.arduino.cc/en/reference/wire.

[24] *Visual Paradigm Online.* URL: https://online.visual-paradigm.com/.

[25] *Home · Wiki · painlessMesh / painlessMesh.* en. URL: https://gitlab.com/painlessMesh/painlessMesh/-/wikis/home.

[26] *ESP32 Wi-Fi Mesh support with ESP-MESH protocol.* en-US. URL: https://iot-industrial-devices.com/esp32-wi-fi-mesh-support-with-esp-mesh-protocol/.

[27] Yoppy Chia et al. "Performance Evaluation of ESP8266 Mesh Networks". In: *Journal of Physics: Conference Series* 1230 (July 2019), p. 012023. DOI: 10.1088/1742-6596/1230/1/012023.

[28] *TaskScheduler - Arduino Reference.* URL: https://www.arduino.cc/reference/en/libraries/taskscheduler/.

[29] J. Postel and J. K. Reynolds. *Telnet Protocol Specification.* en. URL: https://tools.ietf.org/html/rfc854 (visited on 04/16/2021).

[30] *ESP Telnet - Arduino Reference.* URL: https://www.arduino.cc/reference/en/libraries/esp-telnet/.

[31] *Arduino_JSON - Arduino Reference.* URL: https://www.arduino.cc/reference/en/libraries/arduino_json/.

[32] *Arduino FreeRTOS Queue Structure: Receive Data from Multiple Tasks.* en-US. URL: https://microcontrollerslab.com/arduino-freertos-structure-queue-receive-data-multiple-resources/.

[33] *MQTT Specification*. URL: https://mqtt.org/mqtt-specification/.

[34] The HiveMQ Team. *MQTT Client and Broker and MQTT Server and Connection Establishment Explained - MQTT Essentials: Part 3*. en. URL: https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment.

[35] *Eclipse Mosquitto*. en. Jan. 2018. URL: https://mosquitto.org/.

[36] Takanori Suzuki. *takanorig/mqtt-bench*. original-date: 2015-03-31T16:07:32Z. Mar. 2021. URL: https://github.com/takanorig/mqtt-bench.

[37] Biswajeeban Mishra. "Performance Evaluation of MQTT Broker Servers". In: July 2018, pp. 599–609. ISBN: 978-3-319-95170-6. DOI: 10.1007/978-3-319-95171-3_47.

[38] *PubSubClient - Arduino Reference*. URL: https://www.arduino.cc/reference/en/libraries/pubsubclient/.

[39] Nick O'Leary. *knolleary/pubsubclient*. original-date: 2009-02-02T11:07:58Z. Apr. 2021. URL: https://github.com/knolleary/pubsubclient.

[40] steve. *Paho Python MQTT Client-Understanding The Loop*. en-US. Feb. 2017. URL: http://www.steves-internet-guide.com/loop-python-mqtt-client/.