BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

DYNAMIC MESH NETWORK IMPLEMENTED IN MICROPYTHON ON TOP OF ESP-NOW PROTOCOL

DYNAMICKÁ MESH SÍŤ V MICROPYTHONU VYUŽÍVAJÍCÍ ESP-NOW PROTOCOL

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. JINDŘICH ŠESTÁK**
AUTOR PRÁCE

SUPERVISOR — **Mgr. KAMIL MALINKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Master's Thesis Specification

Student:	**Šesták Jindřich, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Computer Networks
Title:	**Dynamic Mesh Network Implemented in Micropython on Top of ESP-NOW Protocol**
Category:	Security

Assignment:

1. Study existing secure IoT mesh protocols (802.11s, Bluetooth mesh, ESP-WIFI-MESH, painlessMesh), focus on comparison of various approaches.
2. Design a self-organizing mesh network protocol that allows nodes to be connected without the need for a predefined structure (at least 10 nodes). The protocol will support two modes: stand-alone and connected to an external network.
3. Implement the proposed protocol and demo project. The implementation should be available for the micropython port on the ESP32 series of MCUs.
4. Test and evaluate its effectiveness. Demonstrate a function of the protocol on a selected use case such as automatic light control for large spaces. Evaluate your results and suggest possible improvements/extensions.

Recommended literature:

- ESP-WIFI-MESH https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/mesh.html
- ESP-BLE-MESH https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/esp-ble-mesh/ble-mesh-index.html
- PainlessMesh https://gitlab.com/painlessMesh/painlessMesh/-/wikis/home
- 802.11s https://www.ieee802.org/802_tutorials/06-November/802.11s_Tutorial_r5.pdf
- Micropython for esp32 https://docs.micropython.org/en/latest/esp32/quickref.html
- ESP-NOW https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_now.html
- The Raft Consensus Algorithm https://raft.github.io/

Requirements for the semestral defence:

- Items 1 and 2

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Malinka Kamil, Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 18, 2022
Approval date:	November 3, 2021

Abstract

The goal of this thesis is to create a dynamic mesh network using ESP32 microcontrollers for IoT and sensor networks. The mesh consists of several nodes interconnected in a tree structure and is able to overcome node failures. This is fulfilled by creating a new mesh solution that is able to operate with and without an Internet connection. The use of MicroPython enables asynchronous operations to be run in a non-blocking manner. The project is built on top of two protocols, proprietary ESP-NOW and common WiFi. The solution brings possibilities for quick mesh application development, but it is limited by memory consumption. The functionality was tested by the creation of a demo application with three practical scenarios for home use.

Abstrakt

Cílem této práce je vytvořit fungující dynamickou mesh síť na ESP32 microkontrolerech pro využití v IoT a senzorových sítích. Mesh síť se skládá z několika uzlů mezi sebou propojených do stromové struktury a je schopna se vypořádat i z pádem kterýholiv z uzlů. To je zajištěno vytvořením nového mesh řešení, které je schopno fungovat s připojením k Internetu i bez připojení. Použití MicroPython umožňuje asynchronní zpracování neblokujícím způsobem. Projekt je postaven pomocí dvou komunikačních protokolů, proprietárního ESP-NOW a běžné WiFi komunikace. Řešení přináší možnosti rychlého vývoje mesh aplikací, ale je silně limitováno pamětí mikrokontrolerů. Funkčnost řešení byla otestovaná pomocí vytvořené demo aplikace se sadou třech testovacích scénářů pro domácí použití.

Keywords

Mesh network, ESP32 microcontroller, ESP-NOW protocol, IoT, Espressif, MicroPython, sensor networks, mesh application

Klíčová slova

Mesh síť, ESP32 mikrokontroler, ESP-NOW protocol, IoT, Espressif, MicroPython, senzorové sítě, mesh aplikace

Reference

ŠESTÁK, Jindřich. *Dynamic mesh network implemented in micropython on top of ESP-NOW protocol.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Kamil Malinka, Ph.D.

Rozšířený abstrakt

Cílem této práce je navrhnout nové řešení pro mesh sítě na mikročipech ESP32 pro senzorové a IoT sítě a především pro rychlý a snadný vývoj mesh aplikací. Požadavkem na řešení je, aby jej bylo možné uplatnit jak v samostatném módu bez přístupu k další síti, tak v módu připojeném k Internetu. Tato navržená mesh síť se skládá z několika uzlů mezi sebou propojených do stromové struktury a je schopna se vypořádat i z pádem kteréhokoli z uzlu.

Práce byla zadaná firmou Espressif Systems (Czech) s.r.o. (dále jen Espressif) a ta si vytyčila určité požadavky. Jedním z nich je použití bezdrátového proprietárního protokolu ESP-NOW založeného na Management WiFi rámcích. Druhým požadavkem bylo použití MicroPythonu. MicroPython je vysokoúrovňový jazyk a nabízí velmi snadný a rychlý vývoj aplikací. Současně je cílem této práce zjistit jeho limity a možnosti ve světě vestavěných systémů na mikročipech ESP32. Jsem si vědom, že pro vývoj takto náročných systému jsou vhodnější progr, ve kterých již existují řešení mesh systému.

V rámci práce byly prostudovány aktuální řešení mesh sítí a byly použity pro přehled vlastností a následný výběr konkrétních požadavků pro mé řešení. Stromová topologie byla vybrána pro lepší přehlednost a menší počet spojení mezi uzly. Spolu s tímto krokem je nezbytné mít v síti centrální kořenový uzel. V kombinaci s touto topologií je vhodné použít směrování zpráv směrem k cíli namísto primitivního broadcastu za účelem snížení počtu zpráv na síti.

Mé řešení je postaveno na dvou protokolech, ESP-NOW a běžné WiFi. Výše zmíněný ESP-NOW protokol, je vyžádán v zadání. Pro velikost zpráv pouze 250 bytů je použit pouze ke sběru informací o ostatních uzlech a jako základ pro samotné formování stromu. V jeho režii je kromě jiného i příprava a přidávání nových mikročipů do mesh sítě. Tento proces se nazývá Provisioning a je častým problémem IoT zařízení. V mé práci jsem navrhl nový protokol Mesh Protected Setup (MPS) určen speciálně pro přidávání nových uzlů. Po pouhém stisknutí tlačítka je MPS proces aktivován a dojde k předání tajného mesh klíče novému uzlu. Ten pomocí tohoto klíče může komunikovat s ostatními uzly v mesh síti.

Pod pravomoc ESP-NOW protokolu spadá i volba kořenového uzlu. V návrhu řešení je popsáno jak by měla volba kořenového uzlu probíhat, avšak z důvodů komplikaci s firmwarem, je tento proces pouze simulován a kořenový uzel je definován staticky v konfiguračním souboru.

Druhým protokolem je WiFi komunikační protokol. Řešení využívá vlastnosti mikročipů ESP32, které disponují dvěma WiFi rozhraními. Jedno rohraní je pro roli stanice a druhé funguje jak přístupový bod (AP) nebo-li WiFi router. WiFi protokol slouží především pro tvorbu stromové struktury. Díky jeho možnosti přenášet až 1500 bytů dat je vybrán i pro přenos dat uživatelem definované aplikace. WiFi spojení mezi jednotlivými uzly se začíná formovat teprve až se zvolí kořenový uzel. Ten jako první pošle své WiFi údaje synovským uzlů a ty se připojí k jeho AP rozhraní a tím vznikne hrana mezi uzly ve stromové struktuře.

Mesh síť se umí vyrovnat s pádem kteréhokoli z uzlů. K detekci pádu otcovského uzlu dojde do 10 sekund, což je následováno kompletním restartem uzlu. Detekce pádu synovského uzlu je složitější a je na ni třeba 26 až 30 sekund. Po pádu synovského uzlu je změna o struktuře propagovaná a následníci mrtvého uzlu mohou být opět zařazeni do stromové struktury na jiném místě.

Pro ověření funkčnosti vytvořené mesh sítě byla implementována i jednoduchá demonstrační aplikace. Ta každému uzlu náhodně vybere barvu LED diody a po stisku tlačítka

je tato barva přeposlána ostatním uzlům sítě, které začnou svítit stejnou barvou. Tato aplikace funguje i jako vzor pro tvorbu uživatelských projektů pro fungování na této mesh síti.

V práci bylo ověřeno, že řešení funguje a mesh síť je opravdu schopna se vyrovnat s pádem některého z uzlů. Programování v MicroPython je velice snadné, ale není bez problémů. Kvůli špatné kompatibilitě firmwaru na mikročipy ESP32-Buddy a verzích MicroPython se mi nepodařilo použít 4 MB externí PSRAM paměti. Zpřístupnil jsem pouze 111 KB RAM paměti pro haldu v MicroPythonu. Tento limit značně omezuje výsledky práce. Mým testováním jsem ověřil plnu funkčnost a stabilitu na 6 uzlech při provozu 24 hodin. Ve scénářích jsem uzly odpojoval a zase připojoval abych ověřil schopnost vyrovnání se s pádem uzlu a průběžně měnil barvy LED diody pomocí demonstrační aplikace. Bohužel už při 7 uzlech docházelo v důsledku neefektivního využívání paměti k chybám a samovolnému restartu uzlů v důsledku nedostatku prostředků pro zpracování velkého množství zpráv. Předpokládám, že opravením chyby kompatibility firmwaru by mesh síť mohla fungovat na mnohem více uzlech. Avšak řešení v MicroPython nemůže dosáhnout kapacity jako aktuální řešení v jazyce C, jehož možností je propojení až 1000 uzlů.

I přes problémy s množstvím uzlů, je řešení vhodné pro rychlý návrh a odzkoušení aplikace pro tvorbu prototypů finálních řešení na jiných platformách. Návrh mesh sítě popisuje nové způsoby volby kořene a vytvořený MPS protokol pro provisioning je snadný způsob dynamického přidávání nových uzlů do sítě. Až na nedostatečný počet uzlů, je zadání splněno. Řešení je portovatelné na mikrokontrolery rodiny ESP32 z důvodu použití proprietárního protokolu ESP-NOW a limitu paměti.

Dynamic mesh network implemented in micropython on top of ESP-NOW protocol

Declaration

I hereby declare that this Diploma's thesis was prepared as an original work by the author under the supervision of Mgr. Kamil Malinka Ph.D. and I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Jindřich Šesták
May 16, 2022

Acknowledgements

I would like to thank my supervisor Mgr. Kamil Malinka Ph.D. for his help in this project. My thanks also belong to Mr. Sergei Silnov who was my external consultant from company Espressif for his comments and remarks and engagement through out the whole project.

Contents

Chapter 1

Introduction

Smart devices and smart homes are increasingly popular nowadays. A smart device is a device connected to a network that can be managed remotely. Among these devices belong smart electronic devices like washing machines, heat control, and others. Many technologies focus on developing smart homes and connecting devices together. Networks of many devices are called mesh networks. Another use of the mesh network is for sensor monitoring. Such technology can be used for better irrigation of crop fields, temperature control in warehouses, or effective light control in offices.

Currently, there are many solutions for mesh networks connecting smart devices. These solutions are not compatible and often use special and expensive devices with new specialized standards that programmers have to learn. Some solutions are developed on cheap microcontrollers and use a common WiFi connection. However, these solutions are not versatile enough.

In this thesis, there are proposed new mesh network protocols for connecting small and inexpensive microcontrollers from the ESP32 family. This solution promises the autonomous function of devices online and offline, allowing devices to function in remote areas without the Internet connection.

The goal of this thesis is to design a new mesh network protocol that maintains microcontrollers ESP32 and connects them. These connected devices can run applications such as light control in warehouses. It aims to develop a single approach of managing devices connected to the Internet as well as a stand-alone network of devices in remote areas without an Internet connection. The proposed mesh should be able to operate on its own and even overcome failures of some microcontrollers and continue working. It attempts to create a platform for quick and easy prototype development of mesh applications.

In the chapter 3 there is a description of a microcontroller ESP32 and ESP-NOW communication protocol which are used for mesh creation. Mesh networks, principles, and existing solutions for ESP32 microcontrollers are described in chapter 5.1. This chapter offers a summary of three existing solutions and a debate on specific aspects of each solution, which is followed by my own proposed mesh design, including explanations of several key parts and overall functionality. Based on the design, the implementation in MicroPython is described in detail in its own chapter 7. Testing and evaluation of the designed network can be found in chapter 8.

Chapter 2

Motivation

In recent years, there is a boom in smart home devices and the Internet of Things in general. Many companies are trying to develop their functioning protocols. However, due to lack of respect and late publishing of the standard, no solution is following it. Therefore, these solutions are not compatible. Furthermore, these solutions for ESP32 are not versatile enough as they often offer mesh networks only in environments with WiFi AP or only without it.

The motivation for this project is lack of easy and flexible mesh network solution. MicroPython programming language is very popular and it is expected that the programming community will have interest in this project for use in homes.

The assignment from the company specifies the use of **MicroPython** as the company aims to meet the possibilities and limitations of MicroPython on **ESP32** boards. MicroPython should offer easier reprogramming and improvements for more specific use-cases. The company also requires to use the proprietary ESP-NOW protocol. Because this protocol is currently supported only on ESP8266 and ESP32 microchips family, the development aims only for ESP32 boards and portability on a different platform is not currently possible.

The solution should be automatic and **self-organising**, meaning that the mesh will form its connection without prior configuration. Dynamic mesh networks should be able to act on changes in the mesh. Meaning the addition of nodes in the existing mesh is possible and the mesh will reorganise on node failures, which is called **self-healing**.

Right now, there are at least three mesh network solutions working on microcontrollers of family ESP32 [34]. First, ESP Bluetooth Low Energy Mesh is based on Bluetooth technology. In this mesh, nodes are connected to as many devices as they possibly can. The mesh is without any structure and uses flooding as the only way of transmitting messages. PainlessMesh is a library in C++ language that offers small and fast deployment of the mesh using a WiFi interface. Nodes create few connections to other nodes. And the mesh uses routing instead of flooding to reduce number of packets. The third solution is ESP-WIFI-MESH, which also uses a WiFi interface in mesh and routes packets. This solution is more reliable and faster. These solutions are described in detail in section 5. This thesis is note-worthy because it aims to develop single mesh network protocol that can manage mesh networks connected to WiFi access points or a stand-alone mesh network without an Internet connection.

A Mesh network is a network in which every node communicates with each other. This can be achieved either by flooding through messages by broadcast or by unicast routing. The decision to use the **routing** reduces the number of packets in the network. In a completely connected mesh network, there would be too many routes between two nodes. In order

to reduce this number, a structure is created in the mesh. That structure takes form of a **tree topology**, in which there is only one path between every two nodes. This changes the meaning of routing a little and as such it is better to use switching when there is only one path to choose. A Mesh network that routes the traffic needs a root node, which manages the mesh and is often connected to the Internet. However, this created an issue with the root node election.

Our solution uses a combination of two technologies. **ESP-NOW** protocol [28] is used to collect information about nodes in the mesh. Prior to **WiFi** connection and transmitting of data, the mesh is formed based on the collected information from ESP-NOW. The mesh requires a root node, which can be elected automatically using an index based on link quality. After the root node is elected, it manages and directs the further forming of the mesh. In the process of formation, nodes connect to each other through above mentioned WiFi. Node is connected to only a subset of nodes it sees and the aim is to form connections with nodes with the best signal.

This project brings another solution to mesh networks for ESP32 microcontrollers. With the use of MicroPython, it aims to become more popular for community projects and spread to more users. A new way of forming the mesh is presented. Additionally, this solution can work either with a connection to the Internet or without it, while there is no need for manual reconfiguration. The mesh is formed without any prior setup except key credentials.

This work aims to develop a universal solution mainly for home use for IT enthusiasts and hobbyists. But it could be used for quick and easy development of mesh applications for mock-ups and prototyping.

Chapter 3

Espressif tools

This chapter contains descriptions of tools and devices developed by a company Espressif Systems (Czech) s.r.o. (from now on only Espressif) [35] needed for this thesis. The company develops low-cost SoC[1] microcontrollers and IoT solutions. These microcontrollers contain many peripheral interfaces and GPIOs ideal for various scenarios and complex applications. This chapter first overviews an ESP32 microcontroller 3.1. Then there is a description of an ESP-Now protocol that provides connectivity between ESP32 microcontrollers in the section 3.2.

3.1 ESP32 microcontroller

ESP32 is a series of low-cost and low-power SoC that has wireless connectivity through WiFi and Bluetooth. This series of microcontrollers was introduced in 2016. It is a successor of the former microcontroller ESP866. The newer ESP32 has bigger memory, better CPU, and better management of wireless interface than the former ESP8266. Many variants of ESP32 boards have been released. There are two different versions of the ESP32 microcontroller in Figure 3.1 for demonstration. Different versions can have different devices, peripherals, and some support embedded flash, but they are code-compatible and share the same SDK[2]. In Figure 3.2 there is a function diagram of ESP32. It overviews and shows what modules are there in ESP32 and what peripherals microcontrollers can operate.

3.1.1 Specification

The ESP32 used in the development and implementation of this thesis is specific ESP32-Buddy [33]. ESP32-Buddy has an additional 4 MB of SRAM, otherwise, the specification of core elements of the ESP32 boards family is similar. Full specification can be found in ESP32-Series Datasheet [34]. Important features are:

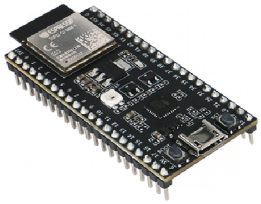
- **Wifi** supports standards **802.11 b/g/n**, 2.4 GHz and up to 150 Mbps
- **Bluetooth v4.2** with BLE[3] support

[1] SoC - System on Chip is an integrated circuit containing CPU, RAM, storage, and peripheral interfaces and consumes a little energy.

[2] SDK - Software Development Kit is one package or collection of software tools for developing certain applications.

[3] BLE - Bluetooth Low Energy is Bluetooth mode that consumes a little energy. That ensures longer duration of the device connected to battery.

- Processor **Xtensa single-/dual-core 32-bit** LX6 microprocessor, operating at 240 MHz
- 448 KB **ROM**, 520 KB **SRAM**, 16 KB **SRAM** in RTC
- 34 programmable **GPIOs**
- 12-bit **ADC** with up to 16 channels
- two 8-bit **DAC**
- four **SPI**, two **I2S**, two **I2C**, three **UART** interfaces, PWM modules, Hall sensor
- **Cryptographic hardware** acceleration: AES, SHA-2, RSA, ECC, RNG



(a) ESP32-S2-DevKitM-1 with internal WiFi antenna. Source:[4]

(b) ESP32 CAM with connected camera and SD card port. Source:[5]

Figure 3.1: There are many different versions of ESP32 boards for different purposes, but purchase of microchip only is also available.

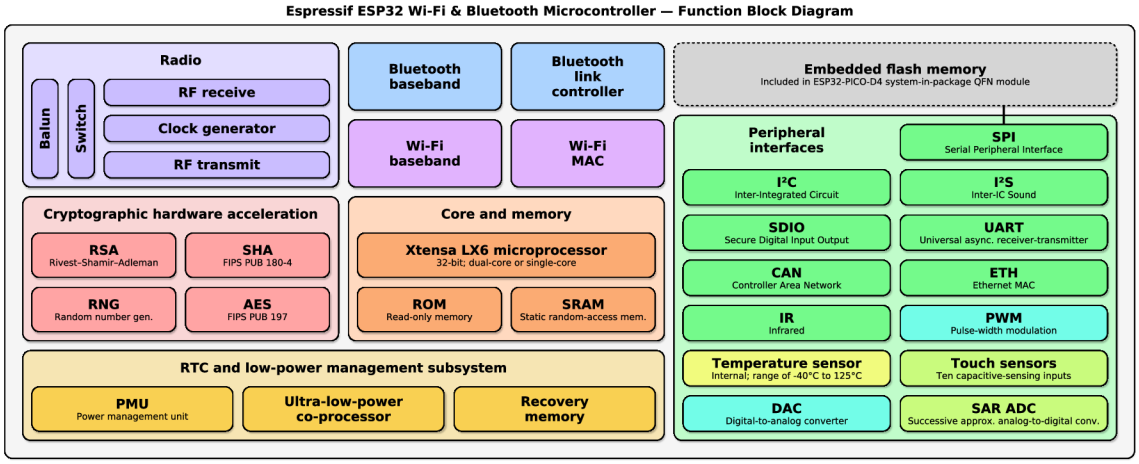
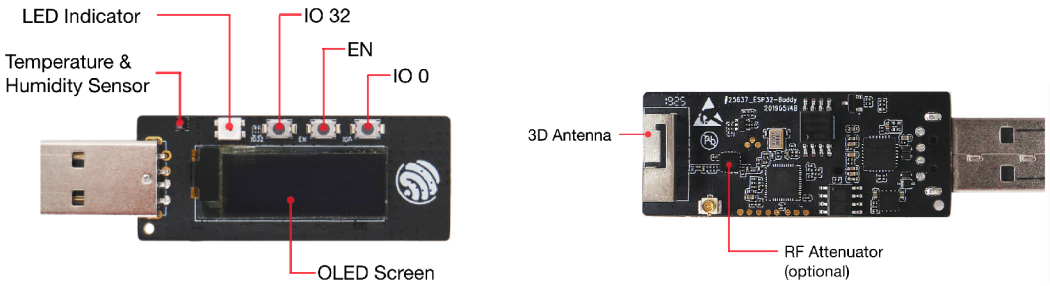


Figure 3.2: Function diagram of ESP32 microcontroller includes modules for CPU and RAM, cryptography, peripheral interfaces, wireless communications and a low power management subsystem. Source:[6]

[4] https://docs.espressif.com/projects/esp-idf/en/latest/esp32s2/hw-reference/esp32s2/user-guide-devkitm-1-v1.html

[5] https://www.nabto.com/guide-to-iot-esp-32

In Figure 3.3 there are the front side and backside of the ESP32-Buddy microcontroller. In addition to the previous specification, ESP32-Buddy has useful peripherals such as a temperature and humidity sensor, LED diode, an OLED screen and 16 MB of flash memory for storage [32]. In addition, it is provided with three pushable buttons [33]. ESP32 WiFi module supports layer two of ISO/OSI communication model using MAC addresses. MAC addresses are permanently written into part of the memory and are unique for every ESP32 board. This address is often used as a unique identifier and this project also uses it as an identifier for each board.



(a) ESP32-Buddy front side

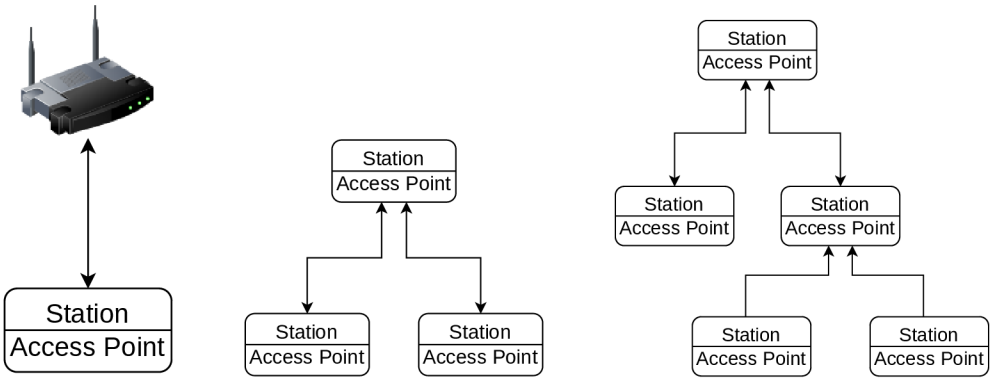
(b) ESP32-Buddy back side

Figure 3.3: ESP32-Buddy board with a description. Source:[33]

3.1.2 WiFi module

The WiFi module consists of two interfaces, one station and one access point [34]. The WiFi can be set in one of these modes or both simultaneously. The station mode or interface can connect to the other network, for example, to a WiFi router. Important is that each board can have only one connection through the station interface to the other network. On the other hand, ESP32 can act as an access point, meaning that it will create its own WiFi network and other WiFi devices can connect to it [25]. It is called Soft access point because ESP32 does not need to be connected to the Internet and therefore does not have to provide an Internet connection. It is possible to combine these two approaches and ESP32s can act as stations and as access points at once and can create the structure of wireless networks. There is a limit of maximum connected devices to the access point interface to only 10 nodes, which is mentioned in question 20 of Wi-Fi Frequently Asked Questions [37]. In Figure 3.4 there are shown these three modes of WiFi interface operation.

[6] https://commons.wikimedia.org/wiki/File:Espressif_ESP32_Chip_Function_Block_Diagram.svg



(a) Station mode can connect to other networks.

(b) Acces point mode form its own network.

(c) Combination of two WiFi interfaces.

Figure 3.4: ESP32 supports three modes of WiFi functionality: station mode, access point mode and combination of both.

3.2 ESP-NOW protocol

ESP-NOW is a low-power connectionless communication protocol that allows multiple devices to communicate with each other without the need for an access point (WiFi router). The ESP-NOW protocol uses IEEE 802.11 Management Frames [3]. These frames tend to manage networks and can be sent to devices that are not connected to any access point. Among Management frames belong such frames as Beaconning message [3], which the access point sends to announce its presence and SSID to other devices. Pairing between devices with ESP-NOW needs to be done before their communication can start. After pairing, the communication can be secure, peer-to-peer, and persistent.[29] The protocol operates on the second layer of the ISO/OSI communication model and uses MAC addresses for addressing.

3.2.1 Frame format

ESP-NOW is a wireless communication protocol that uses WiFi antennas for transmitting small packets. Application data transmitted through ESP-NOW is encapsulated in a special Management frame called a Vendor-Specific Action frame. Due to the small size of packets in combination with connectionless communication and therefore lower transmitting time, the protocol is ideal for smart light and sensor networks and other IoT networks.

The format of a Vendor-Specific Action data frame can be seen in Figure 3.5. The MAC header of the packet is not full as defined by 802.11 standards (figure 3.6) but contains only the first 24 bytes, which represent the following fields: Frame Control, Duration, Destination MAC address, Source MAC address, Basic Service Set ID[7], Sequence Control. The rest of the fields of the vendor-specific frame consists of the following values [28]:

- **The Category Code** field is set to a value *127*, that indicates vendor-specific category.

[7]Basic Service Set ID is set to broadcast MAC address *0xff:0xff:0xff:0xff:0xff:0xff*.

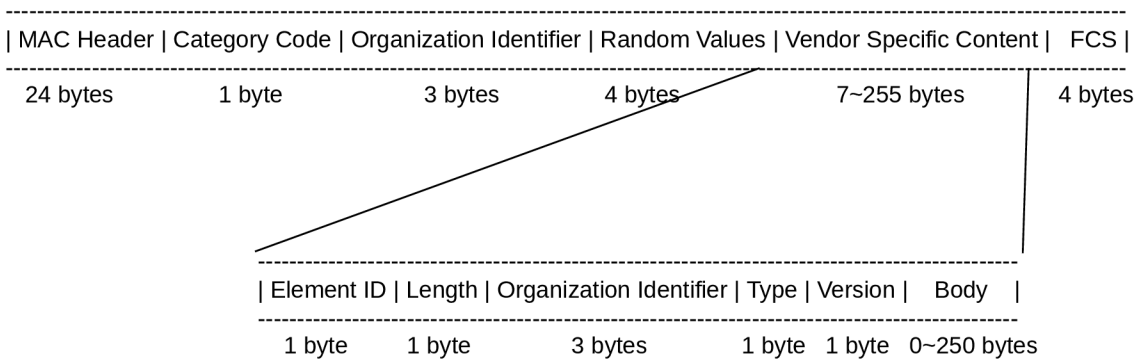


Figure 3.5: The format of vendor-specific action frame and of Vendor Specific Content field. Source:[28]

- **The Organisation Identifier** field has a unique identifier value *0x18fe34* set by Espressif.
- **The Random Value** field contains randomly generated bytes to prevent relay attacks.
- **The Vendor Specific Content**
  - The Element ID has value of *221* that indicates a vendor specific element.
  - The Length is a length of Organisation Identifier, Type, Version and Body.
  - The Organisation Identifier field has value *0x18fe34* (see upper field).
  - The Type has value *4*, that is reserved for ESP-NOW.
  - The Version field is set to the version of ESP-NOW protocol.
  - The Body field contains data. Users can define their own data structures to be sent through ESP-NOW.
- **The Frame Check Sequence** (FCS) is an error-detecting code calculated from the whole frame.

Frame Control	Duration	Address 1	Address 2	Address 3	Sequence Control	Address 4	QoS Control	HT Control	Frame Body	FCS
2 Bytes	2 Bytes	6 Bytes	6 Bytes	6 Bytes	0-2Bytes	6 Bytes	0-2Bytes	0-4Bytes	<Variable>	4 Bytes

Figure 3.6: 802.11 Generic Frame body.

3.2.2 Functionality

The MAC address of the destination device must be registered before sending any data through the ESP-NOW protocol. This process of registration is called adding a peer. The same applies to broadcast communication. Address 0xff:0xff:0xff:0xff:0xff:0xff must be also registered before sending broadcast messages. On the other hand, to receive data through the ESP-NOW protocol, there is no need to register the MAC address of the sender. The

maximum number of paired devices with encrypted communication is six. The total number of all peer devices (encrypted and unencrypted) is twenty.

For the proper working of the protocol, it is necessary to register callback functions with sending and receiving functions. However, these callback functions run in high-priority WiFi tasks and it is recommended to not do time-consuming operations in these callback functions [28]. ESP-NOW can transmit at most 250 bytes of data in one packet. When there is a need to transmit more data in more packets, it is desirable to wait for the callback function of the previous sending frame to end.

For packet security, ESP-NOW uses CCMP[8] protocol for packet encryption [1]. However, the sender and receiver must have the same pre-shared key configured. The device operates with Primary Master Key PMK and several Local Master Keys. Each key is 16 bytes long [30]. PMK encrypts LMK with symmetric block cypher AES-128. There can be a maximum of six LMK keys, meaning that there are at most six encrypted connections between peers. If the LMK key for the peer is not set, encryption of the vendor-specific action frame is not supported. Broadcast and multicast communication are always unencrypted.

[8]CCMP - Counter Mode Cipher Block Chaining Message Authentication Code Protocol or Counter Mode CBC-MAC Protocol or CCM mode Protocol is the standard encryption protocol in WiFi Protected Area II (WPA II) standard.

Chapter 4

MicroPython

MicroPython is an implementation of a Python3 programming language optimised to run on microcontrollers. Some of the core Python libraries are part of this language, but it also includes modules that allow low-level hardware access to the programmer. Because microcontrollers have limited resources of memory and RAM, there are tips and pieces of advice on how to get the most of the resources[1].

4.1 Overview

The main advantage of MicroPython is the same as that of Python, its simplicity. It is easy to learn and write code. Fans of this project have emerged and created a whole community dedicated to spreading and evolving MicroPython. Officially, there is only support for a specialised microchip PyBoard, but the community has created support for ESP32 and ESP2866 microcontrollers.

The whole installation of MicroPython includes a Python compiler to bytecode and a runtime interpreter. The program is compiled to bytecode and then executed. There is an option to install only the runtime interpreter of MicroPython, then it is necessary to precompile the program to bytecode on other devices. Programs can be written in a file and saved on a device or there is an interactive REPL[2] available.

A framework ESP-IDF is needed for the installation of MicroPython on ESP boards. Detailed installation of MicroPython is provided as a guide by Espressif company [10]. MicroPython runs as a task under the ESP-IDF. The ESP-IDF framework is built on top of the FreeRTOS operating system. RTOS (Real-Time Operating System) is an operating system with a scheduler of tasks according to their priority with critically defined time constraints. Only one task can run at a time on one CPU, other tasks must wait.

4.2 Libraries and modules

MicroPython includes many original Python3 libraries and modules such as *os* or *time*. But due to memory and RAM constraints, some modules are only a subset of original Python3 modules and some had to be re-implemented for microcontrollers [8]. A new library *machine*

[1] Tips and advice, how to save RAM and memory https://docs.micropython.org/en/latest/reference/constrained.html.

[2] REPL - Read evaluated print loop is an interactive command-line environment for straight execution of a program.

allows access to hardware devices of the microchip and its peripherals that communicate with the outside world. Classes implemented in this library are for example *Pin*, *ADC* and *SPI*. For communication with other devices, there is a library *bluetooth*

The essential library for this project is the *network* library. This module implements class *WLAN* that enables communication with the network and configuration for station mode, access point mode and a combination of both on the WiFi interface. In station mode, the board can connect to other networks. Using access point mode, ESP32 can create its network [14]. Below 4.1 is a small example of the code of using a network library to configure the WiFi interface in mode and to connect the device to a WiFi network.

```
def wlan_connect(ssid='MYSSID', password='MYPASS'):
    import network
    wlan = network.WLAN(network.STA_IF)
    if not wlan.active() or not wlan.isconnected():
        wlan.active(True)
        print('connecting to:', ssid)
        wlan.connect(ssid, password)
        while not wlan.isconnected():
            pass
    print('network config:', wlan.ifconfig())
```

Listing 4.1: Function to connect to a WiFi network using *network* library. Source:[14]

4.3 Asyncio

Programming language Python can be used with module *asyncio* for asynchronous programming. In MicroPython this module is called *uasyncio* which implements concurrency. Concurrency is defined as the ability of tasks to run in an overlapping manner [26].

One example of the concept of concurrency is parallelism on multi-core processors. In parallelism, multiple operations are executed at the same time, but each runs on a different core. Python implements parallelism through module multiprocessing.

Another example of concurrency is threading, which is also part of Python. Threading is more similar to asyncio than multiprocessing. In this implementation of concurrency, multiple threads take turns executing their instructions on a single-core processor. Usually, one process opens multiple threads. One thread runs for some time and the processor decides when to switch the content to another thread. Threads have some overhead because they need more memory and context switching consumes time and CPU instructions.

In the implementation of concurrency using asyncio (or uasyncio) library, coroutines are scheduled to run in an overlapping but non-blocking manner using cooperative multitasking [26] on single-core processors. A big difference to threading is that the programmer decides when and where tasks should wait and yield resources like CPU to other tasks. That means that the task can not be interrupted in the middle of computing unless it wants to (the programmer wants to). In this case, programmers do not have to worry about thread safety by using locks and mutexes to avoid race conditions or deadlocks [5]. In Figure 4.1 there can be seen different approaches to computing on single-core processors.

Asynchronous concurrency is ideal for I/O operations when a programmer knows that there is an operation that waits for something to happen. For example, when a program ini-

[3] https://devopedia.org/asynchronous-programming-in-python

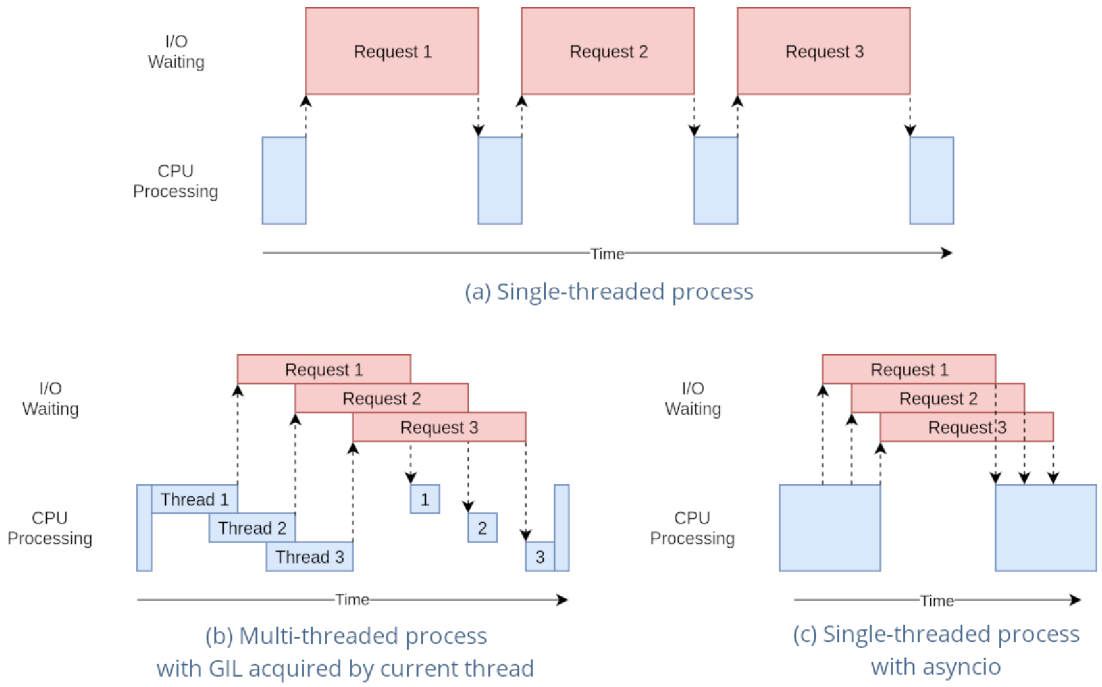


Figure 4.1: Comparison of different computing concepts. Source:[3], originally from [5]

tiates HTTP requests to the server, it does not have to wait for a response. It can continue working on some other tasks and when the program receives a response from an HTTP server, it can be scheduled to continue working. The programmer defines coroutines and events for which coroutines have to wait. The controller and manager of tasks is an asynchronous event loop. The event loop maintains lists of tasks in their states (i.e. tasks ready to run, tasks waiting, ...) and schedules tasks to continue executing as can be seen in Figure 4.2.

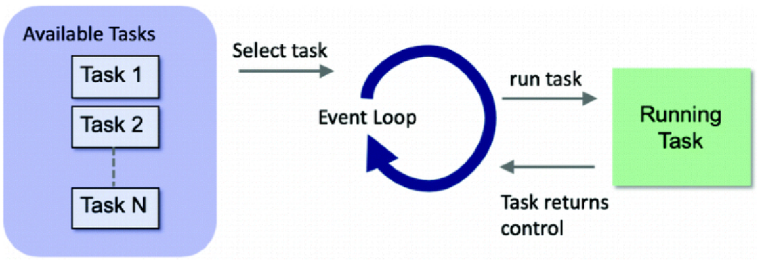


Figure 4.2: Asynchronous event loop manages asyncio tasks and select task to be run. The event loop continues until there are no more tasks to run. Source: [18]

Asynchronous computing uses new keywords. Keyword *async* for asynchronous function definitions and keyword *await* for giving up CPU to other tasks. Await keyword signals to the event loop that it is safe to switch the context to another coroutine [18]. It works similarly to yield in Python, when it does not return, it just yields an actual object and then continues computing (i.e. generator in Python).

Chapter 5

State of the art

This chapter looks closely at a mesh network with a connection to ESP32 boards. First, there is a section about mesh networks as a whole. There is also a short view into the WiFi standard IEEE 802.11s [2]. Later on, three existing solutions are presented and compared to get a better view of different approaches. Although there is an existing standard, neither of these solutions follows it, mainly because of its complexity and delays in its publishing. Therefore, the solutions are vendor specified and not compatible. This project focuses only on wireless technologies, so from now on the terms, Wireless Mesh Network and Mesh Network will be interchangeable if it is not said otherwise. Very popular solutions of mesh network protocols and technologies are ZigBee and LoRaWan, etc., but these are not covered in this project, because they have their own proprietary IEEE radio standards, other than Bluetooth or WiFi 802.11.

5.1 Mesh network

A wireless mesh network is a network of nodes that communicate without a physical connection like optical cable or Ethernet cables UTP[1]. A node in networking terminology is a device that can transmit and receive data [39]. Each node can communicate with every other node in the mesh. Nodes use wireless communication which allows connection many-to-many without said wires. Cables are not too expensive, but the price of switches and routers increases with the number of available ports, so wireless networks are cheaper and scalable [15]. In addition, with wireless communication, it is much easier to add and remove a device to a network without the need to physically connect a new device. Some mesh technologies can extend the range of WiFi access points and offer WiFi connections to clients that are far away. Or they can work as sensor networks, where many nodes across a large area send sensor updates to the mesh.

5.1.1 Topology

In a full (true) mesh topology, nodes connect to as many nodes as they can, but this is usually harder than other implementations. On the other hand, in the partial mesh topology, nodes connect only to a small subset of available nodes, so there are not so many connections.

[1]UTP - Unshielded Twisted Pair is a type of wiring when two conductors are twisted together for better performance.

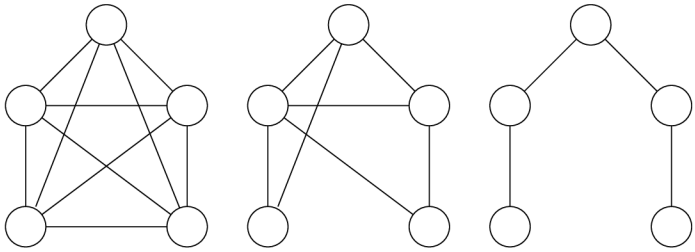


Figure 5.1: Topologies of meshes with 5 nodes: full mesh, partial mesh, tree topology.

Networks can be viewed as a graph, where nodes are vertices and connections between nodes are edges in a graph. In graph theory, a full mesh topology represents a complete graph where every node is interconnected with every other node. Partial mesh is a sub-graph of the full mesh. A special type of partial mesh is a tree topology. The tree is a spanning tree of a complete graph that contains all the nodes but only the lowest number of edges to form a fully connected graph[2]. Different mesh topologies can be seen in Figure 5.1.

Several same nodes form a mesh, but there can also be different types of devices participating in the mesh. In the mesh network, it is important to know that all nodes are equal in means of capability, therefore every node can efficiently participate in the relaying of frames and computing.

Meshes can be peer-to-peer or infrastructure. In peer-to-peer meshes, nodes communicate directly with other nodes within their range and relay messages further in the mesh. Any of the nodes that are in the range of other networks can transfer data to an external network (i.e. Internet).

Infrastructure meshes have a special node called a root node, which is a central point that bears more responsibility than other nodes. The root node can manage the mesh network and decide if and where to add a new node, what to do with the failed node, route traffic to the external network and more. However, every node of the mesh network should be able to take over the role of the central node without any difficulties. That means that even if there is a central point, a single point of failure is eliminated. A single point of failure is one point in the network that relies on this point. If the point fails down, the whole network stops working. This ensures the availability of the mesh network and its resilience [6].

5.1.2 Abilities of mesh

Mesh networks have the capability of self-organising and self-healing, meaning that they can reorganise and operate on their own without the intervention of administrators. Meshes can overcome the failures of nodes during runtime without manual configuration. Nodes can be added to the mesh during runtime to enlarge the mesh's range across the area. Self-organising and self-healing are important to create dynamic networks where nodes can fail or be added to the mesh. The mesh network has to solve fail of the root node in the infrastructure mesh and replace him with another node as can be seen in Figure 5.2.

Transmission of messages in mesh networks is done either by flooding or routing. Flooding uses broadcast as a way of relaying messages and is used in peer-to-peer mesh networks. This type of communication is simple to implement. Every node just forwards the frame

[2]Connected graph is an undirected graph in which there is a path from one node to another for all nodes and vice versa.

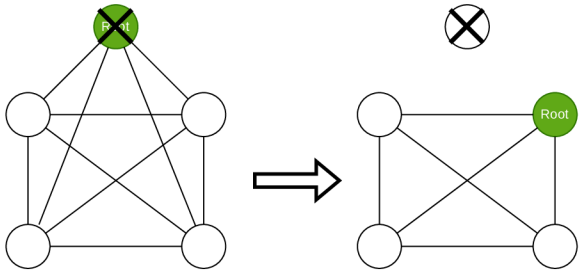


Figure 5.2: Self-healing can overcome failures of nodes and even a failure of a root node. In infrastructure mesh, a new root node must be appointed.

to its neighbours. This ensures that the frame will be delivered to the destination through multiple paths. With flooding, there is a risk of loop and broadcast storms, which overload network resources and the network becomes congested[3]. This issue can be overcome by controlling the flood of frames using the sequence number in frames [22].

Routing propagates the frame along a path by hopping from one node to the other until the frame reaches a destination node. It is harder to manage a mesh network with routing because there has to be a routing algorithm that computes all paths to destination nodes. And when some node fails down, the routing algorithm must recompute all paths. However, because the mesh has many connections between nodes, it is ensured that there is always some path between nodes in the mesh. In Figure 5.3 there are shown diagrams of flooding and routing. Routing algorithms have to find a new route after a direct link becomes unavailable.

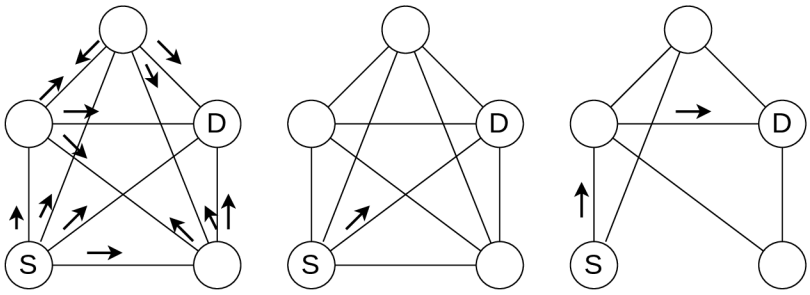


Figure 5.3: Flooding, routing and routing in partial mesh. Flooding uses broadcast and frame floods the network. Message is delivered to destination from several paths. When nodes use routing method, every node knows best path to the destination and sends only one frame.

5.1.3 Standard 802.11s

Standard IEEE[4] 802.11 is a family of norms and standards for the transmission of radio signals in wireless networks. For common WiFi communication standards, IEEE 802.11a/b/g/n are used. In table 5.1 there is a comparison between these versions of

[3]Network congestion causes for example packet loss, decrease throughput and packet delays.

[4]IEEE - Institute of Electrical and Electronics Engineers is the international association for education and technical advancement of telecommunications and other fields.

the 802.11 standards. Standard 802.11a uses 5GHz radio frequency, 802.11b and 802.11g operate on 2,4GHz frequency, and standard 802.11n uses both 5 and 2,4GHz.

Table 5.1: Comparison of 802.11 standards

Version	Release date	Frequency	Max. Rate
802.11a	1999	5GHz	54Mbit/s
802.11b	1999	2.4GHz	11Mbit/s
802.11g	2003	2.4GHz	54Mbit/s
802.11g	2009	2.4/5GHz	600Mbit/s

There are many amendments to these standards to provide better security and efficiency for WiFi devices. IEEE 802.11s defines the principles of how wireless devices should interconnect and communicate to form a wireless local mesh network [6]. Due to the complexity of mesh networks, another consideration had to have been solved and this delayed the final version of the 802.11s standard for several years. During this time, many vendors came up with their own proprietary solutions based on their terminology and technology. Because of that, there is not a unified mesh network technology [17] and solutions are not compatible.

Types of nodes

As is stated in [6] there are several types of mesh nodes:

- Station(STA) is a node that does not participate in mesh organising or frame forwarding. This node connects to the mesh through Mesh AP.
- Mesh Point(MP) node supports the mesh network and participates in mesh organising. It interconnects to another MP via peer links.
- Mesh access point(MAP) combines MP with the functionality of the access point. Therefore, it supports communication with stations.
- Mesh Portal(MPP) is a node through which the mesh network is connected to other networks.

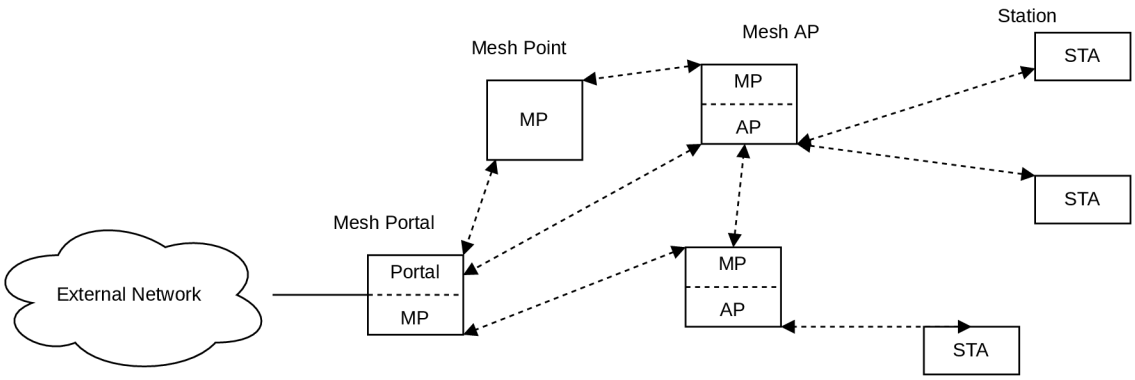


Figure 5.4: The 802.11s network architecture and different types of nodes in mesh network.

In Figure 5.4 there can be seen the structure of the mesh network and the roles of node types. Mesh Portal node needs to implement L3 routing protocol to correctly route packets

from the mesh network to the outside network and vice versa. In infrastructure mesh, the Mesh Portal node is a candidate to be a root node.

Mesh formation

Mesh Point sends beacon messages for new nodes to be able to see a mesh network. In the beacon frame, there are two characteristics of the mesh. Mesh ID is the identification number or name of the mesh network. Mesh profile is another value that tells new nodes what routing protocol and metrics the mesh uses. After a new Mesh Point receives this beacon frame, it will establish a peer connection with the closest Mesh Point already in the mesh if it can participate in the routing protocol and metrics and begins to support the mesh.

Routing

According to the standard 802.11s mesh network should support two routing protocols. The first and default one, the Hybrid wireless mesh protocol, combines the approach of flexible on-demand route discovery and efficient pro-active routing. It uses airtime as a metric. The second and optional routing protocol is Radio Aware Optimised Link State Routing Protocol. More information can be found at [6].

Note that routing in mesh networks may be difficult. Many connections from many routes and therefore maintaining routing tables is memory and CPU demanding. Nodes can frequently fail down or be added to the mesh and change the topology of the mesh, after that re-computation of routes is needed.

5.2 ESP-WIFI-MESH

ESP-WIFI-MESH is a proprietary networking protocol developed by an IT company Espressif. As stated in documentation [31], the protocol is built atop the WiFi protocol and allows nodes to spread over large areas. Nodes are situated in a single WLAN. This protocol is autonomous and implements self-organising and self-healing features.

Traditional WiFi requires that the stations are in the range of WiFi access point or router (AP) to be able to communicate with other networks. This protocol is used to offer WiFi connections for the nodes outside the range of AP. ESP-WIFI-MESH network can span over large areas as the nodes can connect and relay messages to AP and back.

5.2.1 Topology

ESP-WIFI-MESH combines multiple WiFi networks to form a single WLAN. ESP32 board uses a combination of both WiFi modes (see 3.1.2), to act simultaneously as stations and as access points, thus enlarging the span of the WLAN network. Nodes and their connections form a tree and therefore there has to be a root node on top of the tree. It is important to emphasise that only the root node has complete information about the mesh. The depth of the tree can be limited. The limitation of the tree depth together with the number of allowed child nodes indicates how many nodes there can be in the mesh.

Only the root node connects the external network to the mesh and relays packets in and out of the mesh. The root node has several child nodes. Leaf nodes are nodes in the maximum depth layer of a tree and they can not have any child nodes. They can only transmit or receive packets. Intermediate parent nodes are nodes between the root node

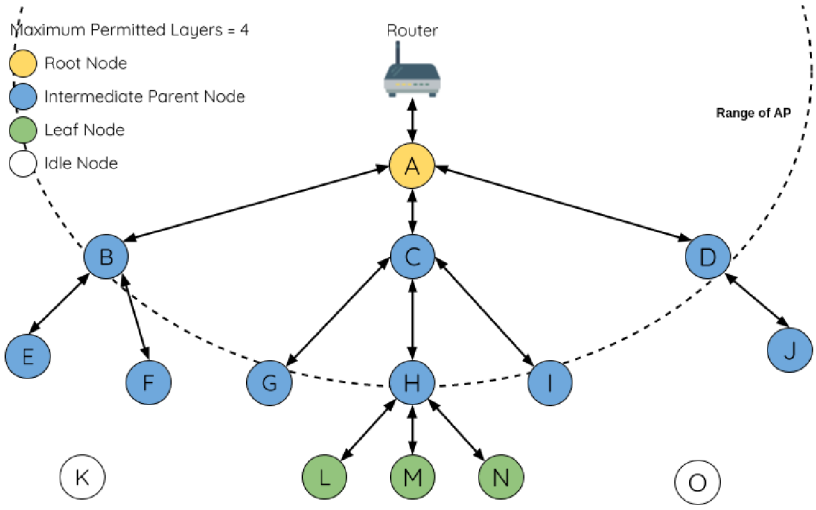


Figure 5.5: ESP-WIFI-MESH topology enlarges WiFi internet connection beyond the range of AP (router). Root node is connected to the AP. Leaf nodes are at maximum depth. Idle nodes are yet to be added to the mesh. Changed, originally from [31].

and child nodes. They are connected to their parent node (or root node in the first layer) and can have multiple child nodes. Idle nodes are nodes that have not joined the mesh yet. Tree mesh topology of ESP-WIFI-MESH can be seen in Figure 5.5.

5.2.2 Root election

The root node can be elected in two different ways. There is an option for the user to define a designated root node. The designated root node directly connects to a WiFi AP and the election process is skipped. Every node in the mesh must abandon the root election process to prevent the election of a new root node, thus causing root node conflict. Nodes must wait for the designated root node to propagate itself using beacon frames.

The automatic root election process depends on WiFi AP RSSI[5] send through the beacon frames of the WiFi router. The election process composes of several iterations. In the first iteration, every node propagates the RSSI value of the WiFi AP it receives to the neighbouring nodes. Every node then compares other RSSI values received from neighbours and selects the best one. In the second iteration, nodes propagate the best-selected RSSI value together with the ID of the node RSSI value originated from and on and on. After the last iteration, nodes check their vote percentage and if some node has a count of votes above some threshold, it is selected as the root node. For automatic formation, the WiFi AP is necessary, but after the mesh is formed, the mesh network does not need WiFi AP anymore. [31]

5.2.3 Mesh formation

Nodes send their WiFi beacon frames to inform other nodes about themselves. New idle nodes listen to these beacons. Idle nodes can have multiple possible parent choices. Firstly,

[5]RSSI - Received Signal Strength Indication measures power presented in the signal. Values are in dBm (Decibel-milliwatts) units.

it takes into consideration the depth of the possible parents and chooses the shallowest one. If the depth of the two best possible parent nodes is the same, then it prefers the one with the fewest child nodes. According to ESP-WIFI-MESH documentation, the first decision criteria ensures that the mesh minimises the depth of a tree and the second criteria balance the child nodes between nodes on the same tree level.

5.2.4 Routing

Because the mesh has a structure of a tree, it would be ineffective to broadcast packets in the whole tree, instead of routing hop-by-hop is used. Each node in the mesh has information about its descendants (sub-tree) and where they are in the tree. The routing table consists of sub-tables related to each child node with their subnet. Node also knows its parent node. If there is a record of destination MAC addresses in the routing table, the node sends a packet to the correct child node. If no MAC address in his routing table matches the destination MAC address of a packet, the node forwards the packet to his parent and hopes the parent knows the destination MAC address. This continues until the packet reaches the root node. Because the root node is on the top of the tree, it knows the whole topology.

5.3 PainlessMesh

The PainlessMesh library offers an easy solution to the creation of a simple mesh network on ESP826 and ESP32 boards. The library is written in C++ language. The mesh can function without any planning, central node or WiFi router. The PainlessMesh library uses JSON in its messages because JSON is human readable and therefore easy to debug. Self-organising and self-healing features are implemented in the mesh. Another useful feature is time synchronisation is also implemented. The PainlessMesh library was developed in 2016 as a GitLab project [7].

5.3.1 Topology

The PainlessMesh library constructs a star-like network topology, as can be seen in Figure 5.6, where nodes are equal. The composition of several stars together forms a tree structure. Nodes can act as a station and as an access point simultaneously (see 3.1.2). Nodes can connect to only one access point[6] interface of another node. The root node is optional. And nodes are equal. Nodes in the centre do not have any advantage over peripheral nodes (unlike in ESP-WIFI-MESH where the root node has more information about the network). Nodes exchange their lists of connections and subconnections of their child nodes with other nodes, thus every node knows the whole topology. Topology updates are sent every 3 seconds.

5.3.2 Root election

By default, there is no need for a root node and a mesh can form and function without it. Nodes can instantly self-organise into the mesh. However, optionally there is a function with which a static setting of a root is possible. The formation of the mesh can be sped up by telling all the other nodes that there is a root node in the mesh. The setting of the root

[6] Access point references an interface of ESP32 that is able to form a downstream connection. The access point in ESP-WIFI-MESH refers to a WiFi router access point.

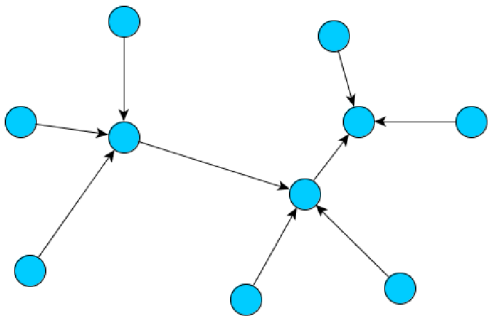


Figure 5.6: PainlessMesh topology forms a star. Source: [7].

node is for topological reasons. Because of the nature of mesh formation, nodes can form several little independent clusters or sub-meshes and not a single mesh, which is shown in Figure 5.7. Documentation [7] explains that nodes then randomly disconnect and try to form different connections with other nodes to form one single mesh containing all nodes. This process is time demanding and random, therefore there is no prognosis of how long it can take.

5.3.3 Mesh formation

Node serves as an access point for other nodes to connect to and simultaneously acts as a station and connects to the other's node access point interface. The limit of a station (child) nodes of one access point interface on ESP32 boards is 10 as it is said in the FAQ forum in question 20 [37]. Nodes that are not in the mesh scan for the available access point. The connection to an access point takes into consideration two factors. Firstly, a node can connect to the node that is not present in the list of connections. It ensures that there are no loops in the mesh. Therefore, there is no danger of a broadcast storm occurring. Secondly, it will connect to the node with the best RSSI signal.

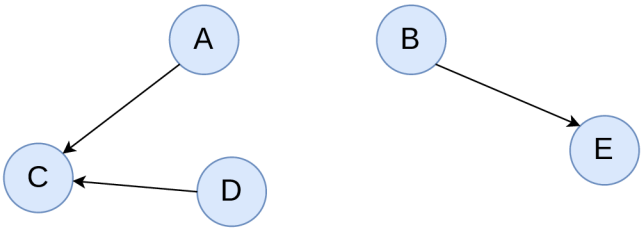


Figure 5.7: Wrong choices of upstream connection can lead to formation of several separate small mesh networks.

5.3.4 Routing

Every node informs its neighbours about all its connections and subconnections of its child nodes every 3 seconds. Eventually, every node will know the whole topology. Because the mesh is in the form of a star or a tree topology, there exists only one possible path between two nodes. This makes routing simple. Routing is similar to the routing process in the ESP-WIFI-MESH protocol, but every node knows the complete topology of the network.

5.4 ESP-BLE-MESH

Another mesh networking solution provided by the company Espressif is ESP-BLE-MESH [7]. As said in documentation [27], this solution is optimised for large-scale networks. ESP-BLE-MESH protocol is based on Bluetooth Low Energy radio technology and is built on top of Bluetooth Mesh [40]. Many different devices with different types of Bluetooth standards can be part of the mesh. Contrary to the previous two solutions, the nodes in this mesh have a free WiFi interface, but they can only use station mode. The foregoing statement means that a node can be part of the mesh while simultaneously being connected to a WiFi router.

5.4.1 Topology

Compare with previous technologies, ESP-BLE-MESH forms a fully connected peer-to-peer mesh without a root node. Nodes are connected to many other nodes within their range. This mesh can contain thousands of nodes. Nodes sent heartbeat messages to let other nodes determine the topology of the subnet. Actually, the topology is much more complex and contains four types of nodes [23]. Nodes are alive stations, but low power nodes are in a sleep mode and come alive once in a while. Friend nodes store messages for low power nodes and when the low power node comes alive, the friend node sends all the messages designated to the node. Relay nodes forward messages further to the mesh.

5.4.2 Mesh formation

After the nodes are up and running, they do not form a mesh instantly. There is a need for provisioning. In documentation [27] there is a guide on how to establish a new mesh. Provisioning is the process of adding a new device to the BLE mesh. This process requires a provisioner device, such as a common smartphone with *nRF Mesh* application, that provides data to a new node that allows it to become part of the mesh. The provisioner device sets the network cryptographic key for the mesh. After provisioning, there can be a phase of configuration, when the application key can be set.

5.4.3 Routing

ESP-BLE-MESH does not use routing in the mesh. It uses flooding instead. Because nodes have many to many connections, there are several paths through which data flows. The key to managed flooding is that each node sends one message to all the other nodes in range, but only special nodes called relays can re-transmit the message further. A message floods the mesh and it is ensured that every node receives the message. This solution is supposed to transfer a small amount of data. The main use is for sensor or control networks. Special node called mesh gateway is used to transmit data between Bluetooth and non-Bluetooth network [23].

5.5 Summary

This section sums up and compares the previously mentioned solutions. There were no information about the exact maximum number of nodes in the meshes, except 1000 nodes

[7] ESP-BLE-MESH - Espressif Bluetooth Low Energy Mesh

on the ESP-WIFI-MESH commercial webpage [36]. The theoretical number of nodes in the mesh is limited by the memory resources on the chip. In ESP32 forum [13], it is said, that the estimated theoretical limit is about 6 levels with 6 child connections, which together gives about 6^5 nodes which is 7776 nodes. With each level, there is also a need to retransmit the packets, which can in ESP-WIFI-MESH up to 30 ms of delay. [13]. But with this many nodes, the delay of the message will rapidly increase due to the occupation of the WiFi channel. And this can also lead to packet loss.

Table 5.2: ESP-WIFI-MESH vs. PainlessMesh vs. ESP-BLE-MESH

Features	ESP-WIFI-MESH	PainlessMesh	ESP-BLE-MESH
Technology	WiFi	WiFi	Bluetooth
Topology	Tree	Star(tree)	Full mesh
Additional device	WiFi router/—	—	Provisioner (smartphone)
Root node &	Always	Optional	—
Election criteria	RSSI/manual	Manual	—
Nodes' Knowledge	Subtree	Whole	—
Routing	Subtable	Complete table	Flooding

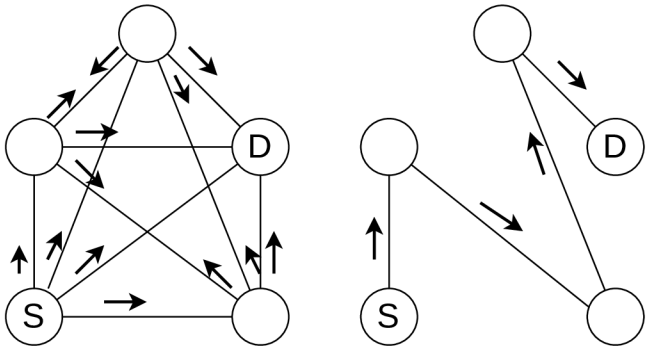
In table 5.2 there is a comparison of these three mesh protocols. ESP-BLE-MESH uses Bluetooth radio transmission as only one of the three, thus having a free WiFi interface that can operate only in station mode. As the other two solutions use WiFi modules to form a mesh, in neither of the three solutions, the nodes do not have a free access point interface of the WiFi module. Therefore, they can not provide an Internet connection to client devices (laptops, smartphones, etc.).

ESP-BLE-MESH needs to provision (add) nodes to the mesh manually using an additional device. PainlessMesh can function on its own, but it is recommended to manually set the root node in the mesh for quicker and more efficient mesh formation. For ESP-WIFI-MESH to organise autonomously, there is a need for WiFi access point routers to automatically select a root node. After that, a WiFi router is no longer needed. When there is no WiFi router, a manual configuration of a root node is required.

Although ESP-WIFI-MESH can work both with and without a router, it can not autonomously elect a root node without a router. PainlessMesh does not need a root node, but setting the root node is recommended for better network formation. Moreover, PainlessMesh can not communicate with other networks like the Internet, although it could when the root node is set manually because it would have a free station interface to connect to router WiFi. ESP-BLE-MESH is a good solution for sensor networks, but transmitting large packets via Bluetooth is not recommended.

All three above solutions are developed for ESP boards. ESP-BLE-MESH is a really simple full mesh network, without additional complexity. However, due to its simplicity and use of flooding, it is quite inefficient, because every node must receive and forward all the messages. The destination node will receive the packet several times from multiple nodes which ensure delivery. This blocks a lot of bandwidth and floods the network with many packets. On the contrary, ESP-WIFI-MESH and PainlessMesh have some internal structure and use routing which generates fewer packets and nodes process only messages destined for them. Although routing needs more memory for the routes and the computation of the routes needs some overhead, I think it is still more efficient than flooding. For example in a full mesh with N nodes, the flooding would always generate N-1 messages as every node

except the receiver would retransmit the message. But the maximum height of a non-full binary tree is N. This means that even in a very simple structure mesh with nodes connected in line (a tree with max one child node), the worst-case scenario is when the node at the beginning wants to send data to the node on the end is still N-1 packets. But every other node that wants to communicate with the end node would trigger fewer packets as the path between nodes would be shorter. And the direct neighbour would trigger only one packet. an example of five nodes can be seen in Figure 5.8



(a) Using flooding every node resends the packet again generating additional load.

(b) Using routing even in line of nodes structure generates worst case scenario the same amount of messages.

Figure 5.8: Flooding in full mesh generates N-1 messages in the network. Routing can generate the same amount for worst case of communication when two nodes on opposite ends of line of structure wants to communicate. But for other possible communication, there are less packets generated

It is possible to have an unstructured mesh that uses routing, but due to many routes, it would generate a bigger computation load to store them and decide on the best route. Also in structure mesh, it is possible to use flooding. A node would send packets both upstream and downstream. It could be useful when information on the nodes is inconsistent and the node doesn't have a complete topology. To deliver the message node would send it every possible way and hope that some other node may know the correct path.

The topology would require selecting only one root node in order to avoid problems shown in PainlessMesh 5.7 and thus the creation of multiple meshes. This one root node has a free station network interface available and can connect to the router WiFi access point and offer an Internet connection to the mesh. The root node can be viewed as a single point of failure, but it is not true as meshes should have the self-healing ability to overcome any node failure and reorganise.

Chapter 6

Mesh Protocol Design

In this chapter, there is a description and requirements of a new mesh network protocol. The goal is to create one mesh protocol that can function in two modes autonomously without a predefined structure. The first mode is when the mesh is connected to the WiFi access point. The other one is the stand-alone mode when the mesh does not have access to external networks. There is a deep look into the features of a newly designed mesh network protocol. The functioning of a mesh network is presented with processes for self-healing and self-organising the mesh network. Neither of the foregoing mesh network solutions is versatile enough for the mesh to be able to self-organise itself with and without WiFi routers autonomously.

The goal of this section is to define a new mesh protocol. This project uses ESP32-Buddy boards and therefore the mesh is wireless. Mesh network consists of nodes of ESP32-Buddy boards and connects them together into one object. Nodes connect to each other and one node can communicate with every other node in the same mesh.

6.1 Specification

Key points of specification from the assignment were created by the company Espressif together with my supervisor. The mesh has the following requirements:

- Two modes of the mesh have to be supported:
  - **Stand-alone mode** supports the mesh in remote places without the presence of a WiFi access point router of another network. For example, on farms or in warehouses.
  - **Connected mode** of mesh operates with a WiFi access point in the range of the mesh network. Mesh can communicate with devices outside the mesh network, either in the same LAN network or in the outside world.
- The network of nodes should be able to self-organise itself **autonomously** without previously predefined structure with both modes.
- Use of **ESP32** boards and **MicroPython** are required
- The mesh should be build on top of the **ESP-NOW** protocol.
- The mesh network has to be able to manage at least **10** nodes.

The mesh protocol has been designed and some new specifications were set to aim for better mesh performance, list of the new specifications is below in items. Together with the ESP-NOW protocol operating in broadcast, the WiFi socket connection will be established for one to one communication. Tree topology is chosen in order to create fewer connections with WiFi protocol and because the ESP32 microchips support a maximum of 10 connections to WiFi AP interface [37]. In an unordered mesh, it would create an undefined and random connection. Tree topology is more transparent as there is only one path between nodes and additional computation is not needed, so instead of routing it just switches the packet to the peer on a path to the destination. It is also better to manage and understand as the root node can manage the tree.

With the topology there is no need to flood WiFi packets and routing is supported to reduce the traffic in the network. That said, WiFi uses routing. But ESP-NOW uses flooding because it is connections less and doesn't need prior configuration or settings. Using ESP-NOW protocol, later on, the WiFi connections can be established. Due to only 250 bytes of payload in ESP-NOW protocol, it is not used for topology or application messages which can grow in size. Hence it serves as a base layer for getting the information about the nodes in the area. WiFi protocol with the ability to transport up to 1500 bytes on ESP32 will be sufficient to transport big application data and topology messages. A basic summary is below:

- **Tree** topology with root node where every node knows **whole** topology.
- **ESP-NOW** protocol is used for management. Mesh uses **WiFi** only for topology updates and application data. They work simultaneously.
- Combination of **flooding** in ESP-NOW protocol and **routing** in WiFi.
- **Self-healing** ability will deal with failures of nodes already presented in the mesh.

Operations and approaches to both Connected and Stand-alone modes are the same in order to create one transparent solution. The state diagram of the nodes in the mesh can be seen in Figure 6.1. Nodes can be in four different states while being inside or outside the mesh. Node is considered in the mesh network while in *idle* or in *tree* state. Otherwise, the node is outside the mesh.

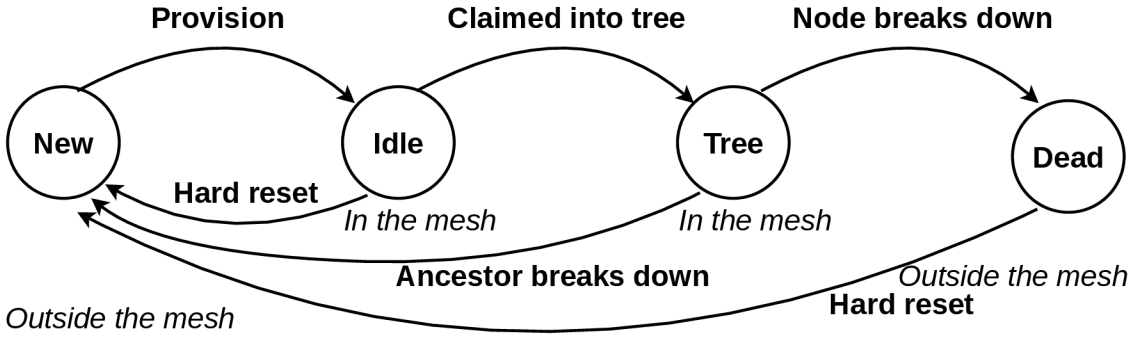


Figure 6.1: State diagram of nodes and transitions between states.

6.2 Overview

The goal is to create one mesh protocol that can function in two modes autonomously, connected to the WiFi AP and Stand-alone without the Internet connection. This versatile approach is lacking in existing solutions. As for any mesh network, it should implement self-organising features for autonomous organising. We added a harder requirement for the mesh to be able to self-heal.

For a collection of information about nodes and for adding nodes into the mesh network, the proprietary ESP-NOW protocol is used. This protocol is built on top of IEEE 802.11 Management Frames. For topology updates and application data transmission, a common WiFi protocol is used. This means that for periodic messages the mesh uses low power protocol ESP-NOW and for bigger data transmitting the WiFi is used. With these two wireless protocols, the mesh uses both broadcast flooding in ESP-NOW and unicast routing for WiFi.

Key stages in creating a new mesh are listed here:

- Provision process is needed to distribute the key to the nodes.
- Collection of information about the nodes in the same mesh.
- Root node election in order to create a tree.
- Creation of links in the tree represents the creation of connection using WiFi interfaces on the parent node and child node.

For easy and not time demanding provisioning of nodes, we proposed the Mesh Protected Setup (MPS) 6.4 method of adding nodes to already installed mesh. There is still a need to manually set the key credentials for message signing and encryption but only on the first node. The addition of nodes to the mesh is done by pressing the button on ESP32 boards.

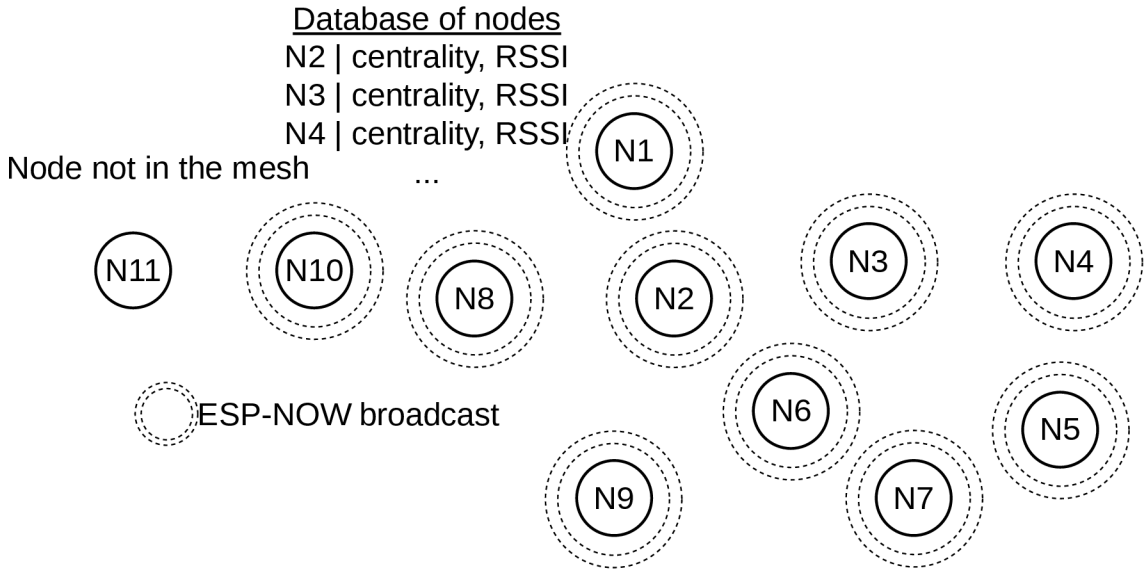


Figure 6.2: Nodes are in mesh, but in idle state. They collect database of all the nodes in the mesh. Root node is selected as the one most in the centre of the mesh. Only ESP-NOW communication is active. No application can run.

Nodes with key credentials send periodic updates through broadcast. Receiving nodes update their node database and retransmit these advertisements. This way nodes collect information about all the nodes with credentials, ergo nodes in the mesh. If nodes didn't receive advertisements about certain nodes for some amount of time, it considers him disconnected and wipes out the record from the database.

The advertisement contains nodes centrality value (viz. equation 6.1) and RSSI signal to WiFi AP if it is to be connected mesh. When nodes are idle, they collect information about neighbours which is seen in Figure 6.2. They can be also used for provisioning new nodes into the mesh using mentioned MPS protocol.

Nodes then enter the root election phase where they need to settle on one root node. After this is done, WiFi protocol is enabled. The root configures its AP network interface and sends its AP identifications to child nodes still using ESP-NOW. Child nodes receive this identification and connect through their STA network interface to the root node. The root and child node has established a WiFi connection or link in the topology and the root node can start sending topology updates and application data to the child node. The child nodes then start their own AP network interface and send identifications about their WiFi AP network interface to their children. And this way at least one WiFi connection on all nodes is created and this forms the tree structure as can be seen in Figure 6.3.

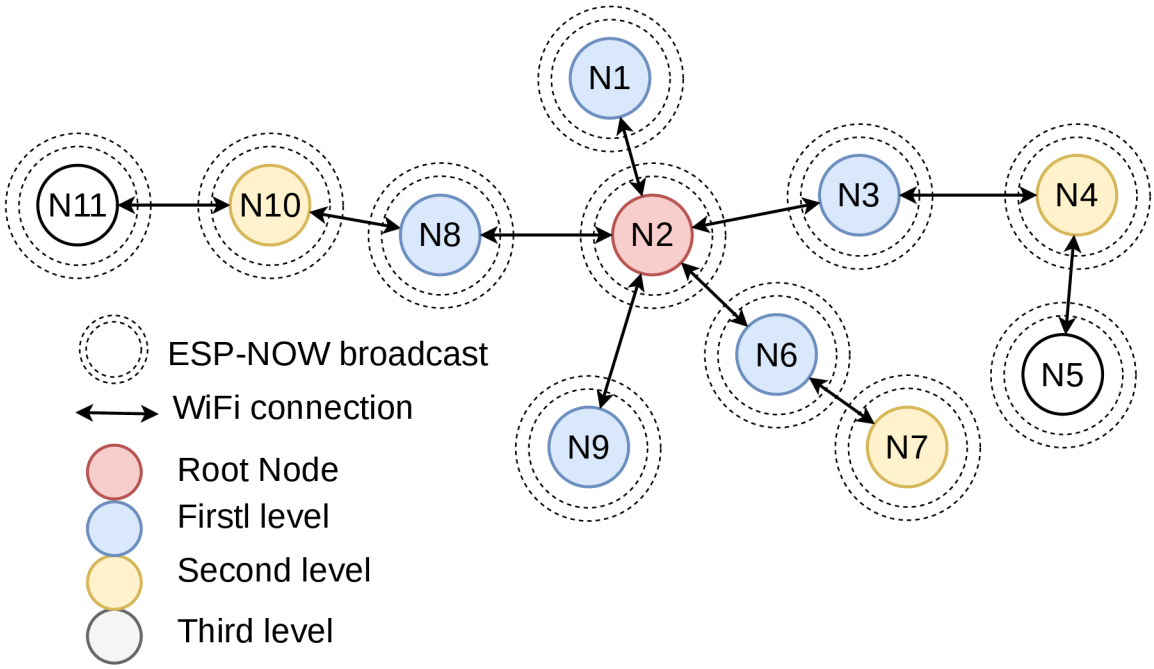


Figure 6.3: Nodes create tree topology of WiFi connections between themselves. They still use ESP-NOW broadcast to be able to add new nodes. The node is fully functioning when it is included in tree and its WiFi connections are active, then the application can run.

Using the WiFi root node sends periodic topology updates to his children. The children then update their topology and send this updated topology to their children. This way, the topology is propagated to the mesh. Every node knows the whole topology and therefore is able to route the generated traffic in the right direction. When a node is inside tree topology, the application can run on the node.

6.3 Topology

Full mesh network topology has a lot of possible routes which ensures quick delivery. However, it is also impractical because a high number of connections can be hard to manage, set up, and monitor. I have decided that the tree topology would be better to manage and organise. In tree topology, there are fewer packets generated as the message is routed between connections. Tree topology forms fewer connections, is structured, and probably better to understand and debug. In case of a failure of a node, the mesh can self-heal itself and reorganise the tree to keep the full functionality. The mesh should overcome even the failure of the root node. The idea to create a tree has been inspired by ESP-WIFI-MESH solution 5.2.

6.4 Mesh Formation

Mesh formation is divided into two stages where the new foreign node is transformed into the idle state (provisioning) and then the idle node can be added to the tree topology, see figure 6.1.

The mesh needs an identifier in order to divide two meshes in the same area. But there is a problem with delivering this key to other nodes safely on the network.

The provisioning of the new node is done by a newly proposed Mesh Protected Setup or MPS 6.4 method. There is still a need to manually set the key credentials for message signing and encryption but only on the first node. The addition of nodes to the mesh is done by pressing the button on ESP32 boards. Using broadcast handshake both new node and node with credentials register each other with predefined LMK security key for encryption in ESP-NOW protocol. Then they securely exchange key credentials in a secure ESP-NOW unicast. They are registered only for this exchange process due to the limit of registered devices, therefore one node in the mesh can one by one send credentials to all the other nodes.

The message in ESP-NOW is considered valid and accepted for further processing if and only the HMAC [20] hash sign of the message matches the computed sign by the key credentials. Otherwise, messages are dropped. The MPS process was inspired by the WiFi WPS method on WiFi access points, where a device can be connected to a WiFi router using pressing the WPS button on both the device and the router. After the MPS process, the node is present in the mesh, but it isn't in the tree topology, it is in the idle state.

Nodes send advertisements broadcasts and collect the database in order to select the root node. After the root election, the root node selects random nodes from the close neighbours, that are not in the tree (currently only the root node is in the tree) and sends them his WiFi AP interface credentials. Nodes receive this and connect to the root node AP interface as stations. The root is informed by this addition. The root updates the tree topology and starts to send periodic topology updates to his child nodes. When child nodes receive the tree topology update, they know they are in the tree topology and they can also select close neighbours and send them their own WiFi AP interface credentials. This way the tree topology is formed

[0]WPS - WiFi Protected Setup is a standard which allows a device to be temporally connected to the network without the knowledge of the pre-shared key.

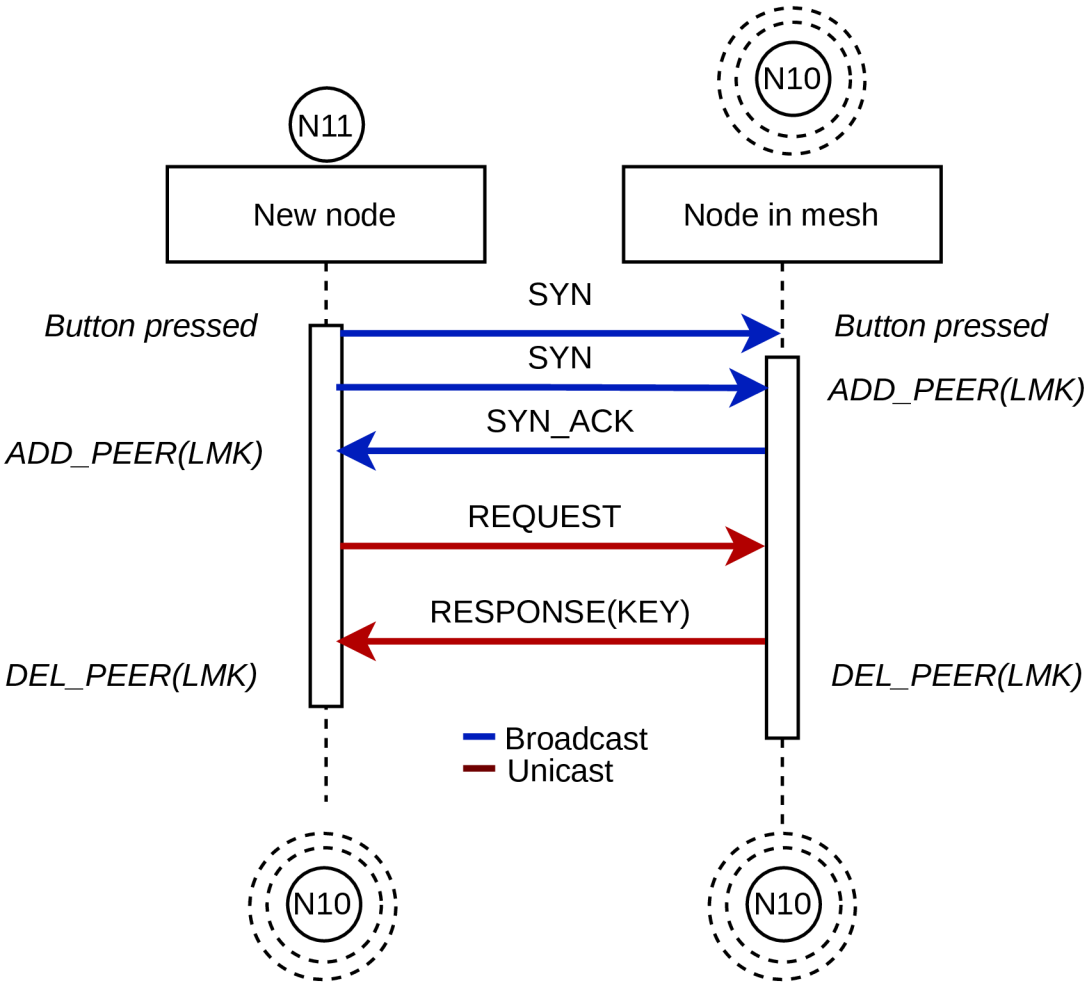


Figure 6.4: Mesh Protected Setup (MPS) for exchanging security key for message signing uses broadcast for registering a peer. Secure key is transferred via ciphered unicast. After this exchange new node can process and send messages signed by mesh security key.

6.5 Root Node election

The root node should be the node most in the centre of the mesh in Stand-alone mode. This is because, from the centre node, a shallower tree is formed than from an edge node. If the mesh is supposed to be in connected mode, it elects the root node based on RSSI value to the WiFi router AP because only the root node has a free station network interface in order to connect to the router. As all nodes periodically advertise themselves and their neighbours. Later on, all nodes have the same information as all other nodes in the mesh. This is called convergence when all nodes have the same content in the database of nodes. The advertisement messages are sent through ESP-NOW broadcast and contain these values:

- **Node ID** is a node identification that is represented by MAC address.
- Value of a function that determines the node's **centrality** (how much in the centre of the mesh the node is) for root node election in Stand-alone mode. It consists of neighbours and RSSI to the neighbours of the node. According to how many neighbours

and how close (RSSI value) they are to the node, it is possible to determine the node the most in the centre of the mesh. This value can lead to some local minimums, but at least it is sufficient to elect a root node in every situation. A function definition is as follows, where x_i is one of the visible neighbours from the set of visible neighbours X:

$$centrality = \sum_{x_i \in X} \frac{1}{\sqrt{|RSSIx_i|}} \tag{6.1}$$

- **RSSI** value to the WiFi access point (AP). If WiFi AP is not present in the range or the mesh operates in Stand-alone mode, the value is zero. The use of this value is to select the root node in **Connected** mode.
- Flag that determines if the **root is already elected**. It is used for new nodes to inform them to not start root election. It tells them to just wait until they are incorporated into the tree topology.

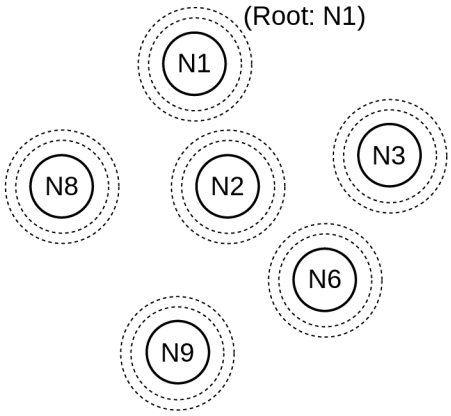
Only messages from provisioned nodes are processed. After there are no new additions to the node database, the node can trigger the root election. It searches through database values and values of itself and selects the best record. If the node is the best record, it triggers the root election process 6.5a, when it sends the root propagation packet of itself, otherwise, it does nothing. When the node receives the root propagation, it compares the received root candidate with its database and propagates the best one further into the mesh 6.5b. If there are two equal candidates, the process selects the one with a lower MAC address. Every node can consequently propagate better and better candidates until every node has the same proposed root node with the best values, so the root is elected. During this process, the database of nodes would not allow writing in order to not change the values during the root election.

The root node election is non-preemptive, meaning that even when a better root candidate appears, it cannot trigger a new root election or depose the existing root. Once the root node is elected, it stays the root node until it breaks down.

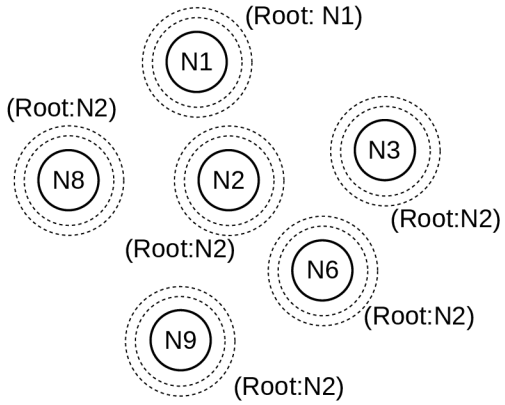
6.6 Routing

Every node forms its own routing table that is composed of topology updates from the parent node. These updates are the same and all the nodes have the same topology. Although, there can be some inconsistency with propagation delays into lower layers. For example, when a new node connects to the parent, the parent informs the root node about the topology change. The root node in the next periodic update starts to propagate a new tree topology. But there can be delays until all the nodes receive the new topology. But meanwhile, they can receive packets from the new node. In order to not drop the packet, the child node sends a packet, which it knows no route to, to his parent node and hopes that the parent already has the new topology. This process can continue until the packet reaches the root node, which knows the topology at the moment and therefore has the route.

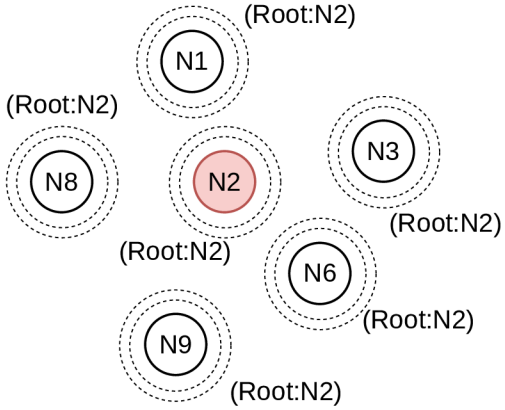
The routing is not much of routing because there are no multiple routes to the destination. Tree structure guarantees that there is only one possible path between two nodes. Therefore nodes can compute their routing table as subtrees. They know about their own



(a) Node N1 triggers root election and claims it should be the root.



(b) Other nodes respond with suggestion node N2 as better candidate.



(c) Node N1 accepts because N2 is better candidate and root can be established.

Figure 6.5: Close up on the central nodes. Root election process is the same in both modes. In Stand-alone mode it depends on centrality value. In Connected mode it depends on RSSI to WiFi router value.

children and parent node. From topology, they can see what descendants nodes are connected to what child node and mark as the next-hop its direct child node. For other nodes (nodes that are not descendants and are not in topology), it marks the next hop as the parent node.

6.7 Self-healing

Self-healing ability allows a mesh to continue working whenever some node breaks down. The mesh should overcome even root node failure. Be aware that topology breaks only on a WiFi level because ESP-NOW still broadcast advertisement messages. In this design, it is proposed that when some node breaks down, other nodes behave differently depending on the relation to the broken node. When the child node breaks down, the parent will notify the root node about the topology change. The whole subtree (all the descendants) behind the broken node is dropped and the parent erases his connection to a dead child.

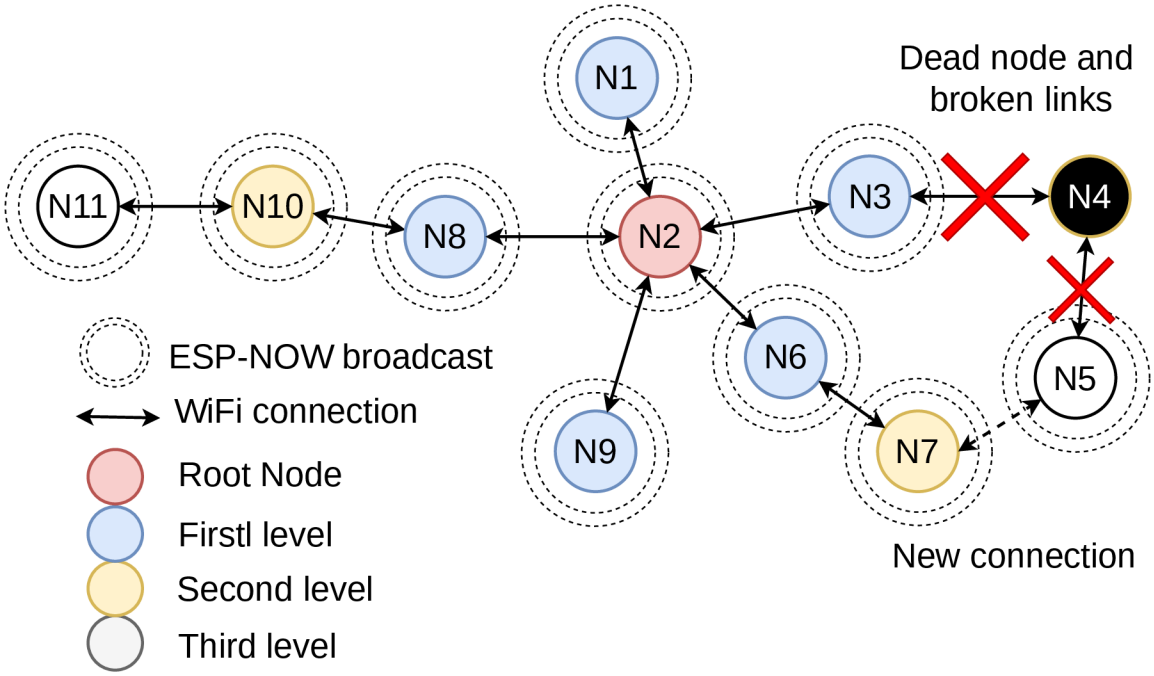


Figure 6.6: When node N4 breaks down, the parent node N3 notifies the root node and all the descendants are dropped from the topology. The child node N5 resets its STA interface and waits to be claimed by different parent node N6.

The root node propagates the new topology to the tree and other nodes will be notified about the change.

From the opposite point of view, when a parent node breaks down, the child loses its connection through WiFi to the mesh. Therefore it has to reset its STA interface in order to be able to connect to the different parent nodes, so it waits until some node claims it back into the tree.

With that said when the leaf node breaks down, it triggers only topology updates. When the intermediate node breaks down, its children reset their STA interfaces and wait to be claimed by a different parent node meanwhile the parent node of the dead node informs the root about the change. an illustration of the break of the intermediate node can be seen in Figure 6.6. In the worst case, when the root node breaks down, consequently every node disconnects and resets its AP interface. With no root node in the mesh, it will trigger a new root election.

Chapter 7

Implementation

This chapter focuses purely on the implementation of the mesh protocol in MicroPython on ESP32-Buddy boards. The use of MicroPython was defined by the assignment from the company in order to explore its possibilities and limitations. It also tries to demonstrate the use of MicroPython for quick prototyping even on microcontrollers. The use of the required proprietary ESP-NOW protocol limits the portability of this solution to only ESP32 and ESP826 boards.

The use of MicroPython is not memory efficient compared to the ESP-IDF framework which uses C and C++. Therefore boards with bigger memory are required. In my project, I used ESP32-Buddy that have 4MB of SPIRAM. Unfortunately, due to problems with compatibility with MicroPython, I was forced to use the firmware with only 111KB of RAM memory for MicroPython heap, which is approximately 36x less than the board is capable of. I used *build-GENERIC* type of firmware and version of MicroPython from specific commit[1].

MicroPython is an implementation of a Python3 programming language optimised to run on microcontrollers. Some of the core Python libraries are part of this language, but it also includes modules that allow low-level hardware access to the programmer like library machine. For this project, the essential libraries are espnow for ESP-NOW protocol operations and library network for directing and managing WiFi network interfaces.

Centre point of implementation is asynchronous event loop which is implemented in uasyncio module. In this implementation, coroutines (==tasks) are scheduled to run in an overlapping but non-blocking manner using cooperative multitasking on single-core processors, similar to threading. But the biggest advantage over threading is, that in asyncio the programmer himself decides when and where should one task yields its resources like CPU to the other tasks. Furthermore, the task cannot be interrupted in the middle of computing unless it wants to, therefore there is no need to worry about locks, mutexes, race conditions and deadlocks, unlike threading. The asynchronous event loop manages tasks and schedules tasks to be run.

7.1 Concept

ESP32-Buddy boards belong to the ESP32 family, but are specially designed for ESP-WIFI-MESH development 5.2. They have 4 MB of SPI RAM for bigger memory. They

[1]Commit *b67384616b87fb56b126fd9befe23225d184091d*, but due to the re-base of *glenn-g20* git branch, the commit is no longer available. This commit is saved on the medium supplied to this thesis.

also have already connected peripherals. More information about hardware point of view of the ESP32-Buddy boards can be found in Section 3.1.

- LED on pin 25.
- Left button on pin 32.
- Reset (middle) button.
- Right button on pin 0.
- OLED display uses I^2C communication with data channel (SDA) on port 18 and clock channel (SCL) on port 23.

The main functionality and logic of the mesh network are implemented in two key parts (cores). I learned how to write big and complex projects in MicroPython from the bachelor thesis of Bc. Josef Kolař on Coordination of MicroPython-based IoT by means of NODE-RED [19]. I have been inspired by the thesis and my cores have synchronous blocking start functions which just schedules main task _run as is demonstrated in example of code 7.1. The main task waits for the right events and schedules additional critical tasks. A more detailed description of important parts of the code follows.

```
class EspNowCore:
    def start(self):
        self.loop.create_task(self._run())

    async def _run(self):
        self.esp.add_peer(self.BROADCAST)
        self.loop.create_task(self.on_message()) # Receive messages
        await self.added_to_mesh() # Wait for MPS.
        ...
```

Listing 7.1: Core classes have blocking start function which schedules asynchronous entry point function _run which then register further tasks.

ESP-NOW core is a base layer and can run on its own. WiFi core waits for concrete events to take place in ESP-NOW core and after the event, it proceeds and schedules coroutines into the asynchronous loop and enables WiFi interfaces for communication. User-defined application imports WiFi core module and have to execute WiFi core's start. The start function executes both cores and the user application can be run as needed. When a node is connected to the tree topology, the application can communicate with other nodes. Module hierarchy and its functions can be seen in Figure 7.1.

7.2 ESP-NOW Core

Class EspNowCore implements procedures and functions related to the base layer of the mesh protocol. This class takes care of communication based on the ESP-NOW protocol. It is the first step in creating communication on the WiFi layer. This core is called from within WiFi core. These important sections are further described:

- Add broadcast MAC address into ESP-NOW communication.

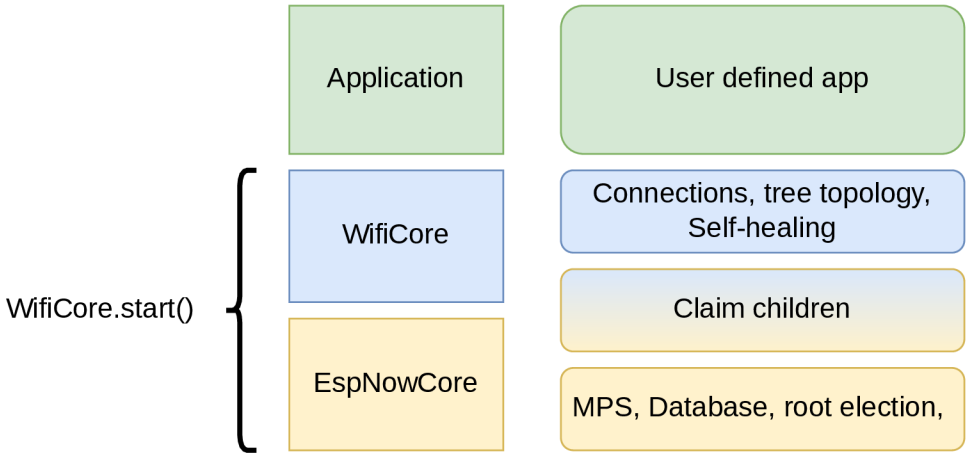


Figure 7.1: The project consists of three major modules. Each module is responsible for subset of working mesh. For mesh to start working, user simply has to init WifiCore and run its start function which takes care of everything mesh related.

- Wait until it has a key for message HMAC-SHA256 signing. Serves as an integrity and authentication check for messages.
- Distribute the key through Mesh Protected Setup process 6.4. Activate by a button pressed for 4,25-8,5 seconds and run for 45 seconds.
- Send periodic beacon advertisement 6.5 every 5 seconds. Manage database of nodes. Retransmit information about other nodes every 13 seconds.
- Root node election is only simulated but not implemented. The root node is set statically in the configuration file.
- Send AES-128 encrypted node's WiFi AP SSID and password to child nodes. This function is triggered by WiFi core but uses ESP-NOW protocol.

As the first step node has to add_peer address into ESP-NOW communication. Every device adds broadcast MAC address *0xff:0xff:0xff:0xff:0xff:0xff* in order to send and to listen to every other node in the area.

Then the core waits until it has the key credentials needed for message signing and verification. The key credential is important to distinguish two independent meshes that can overlap (neighbours in apartments). Messages are signed using HMAC algorithm [20] with SHA256 from the official library hashlib, which is supported by hardware acceleration, to verify the integrity and authenticate messages to be sure that message was generated by some node in the mesh with the same key.

For the receive function to be lightweight, the messages are only received in the loop function and they are registered in another coroutine for future processing as can be seen in the illustration below 7.2. This way it is ensured that no time demanding action takes place in the critical receive function.

On one node it is necessary to set key credentials into a configuration file and upload the file to one board. Other nodes can be added during runtime using buttons through Mesh Protected Setup (MPS) protocol described in the previous chapter 6.4. Button has to be pressed for 4,25s to 8,5s and this allows MPS process only for 45 seconds. The new

node tries to obtain the key credential. The nodes register each other for encrypted unicast communication in the ESP-NOW protocol. For this secure communication, there is a need for predefined Primary Master Key (PMK) and Local Master Key (LMK), these keys are in the configuration file for all the nodes the same. PMK encrypts LMK using AES-128 and LMK encrypts messages with CCMP method [28].

```
async def on_message(self):
    """
    Wait for messages. Light weight function to not block recv process.
    Further processing in~another coroutine.
    """
    while True:
        buf = await self.esp.read(250)
        ...
            self.loop.create_task(\
                self.process_message(msg, digest, msg_len))
                # Process message in~another coro.
           ...
```

Listing 7.2: on_message function only reads the ESP-NOW packet and creates new coroutine for each packet. This way in the function there are no time demanding operations. This principle is the same in WiFi receiving messages.

Nodes send periodic beacon advertisement every 5 seconds with function advertise. Neighbour nodes on first contact immediately resend advertisements from other nodes to inform all the mesh. The nodes retransmit other nodes every 13 seconds to reduce the load on the network. After the node didn't receive advertisements about any node for 26 seconds (twice of 13 seconds to give a second chance), it considers him dead and wipes the record from the database.

The root node election is not automatic. It has to be statically set in the configuration file. For the root node election, the nodes have to scan the network for the RSSI signal of neighbouring nodes. But in the RTOS system responsible for real-time action, the procedure for scanning the WiFi runs on the same thread as the process of receiving messages. In other words, when a node scans the network, it cannot receive messages and they are dropped. According to number 3 Frequently Asked Question forum on ESP32 [37], the scanning procedure for 13 channels takes 2,04 seconds. This scan should be run every time before the advertisement is sent (every 5 seconds). This means that scanning would block 40,8% of the time and the percent of messages would be dropped. A workaround could be that instead of root election based on RSSI to neighbours, only the number of directly visible neighbours would be used. This could be done using TTL or AGE field in advertisement packets. This workaround was not implemented yet.

In order to create WiFi connection, the child node must receive WiFi credentials of the parent node. Parent node therefore sends these WiFi AP interface credentials encrypted with AES Cipher Block Chaining from cryptolib library [9] mode through function claim_children. The child node then informs the WiFi core and further functionality is executed there. The ESP-NOW core remains its full functionality and cores run in parallel using asyncio.

7.3 WIFI Core

The second centre part of my program is class WifiCore which implements the creation of WiFi connections between nodes and thus creates tree topology. This is the main part of the mesh. This class takes care of and implements these key processes:

- Connect to parent node's WiFi AP interface, create socket connection.
- Start its own WiFi AP and open socket for future children. Randomly try to claim a node that is not in the tree topology. It uses EspNowCore.claim_children function
- Parent nodes send topology periodically to children.
- Detection of dead node and self-healing.
- Routing of messages.

Node accepts only first WiFi credentials from node and tries to connect to his WiFi AP interface and creates a socket connections. Socket connection is created with asyncio functions asyncio.open_connection. Credentials are sent using EspNowCore.claim_children function. The parent has static open port number 1234. Node checks and tries to claim new children nodes every 7 seconds.

After that, the node can configure its own AP interface and open a socket on port 1234 for child nodes to be able to connect to it with asyncio.start_server. Then it randomly selects nodes that are not in the tree topology yet and sends them his WiFi AP credentials using EspNowCore.claim_children function.

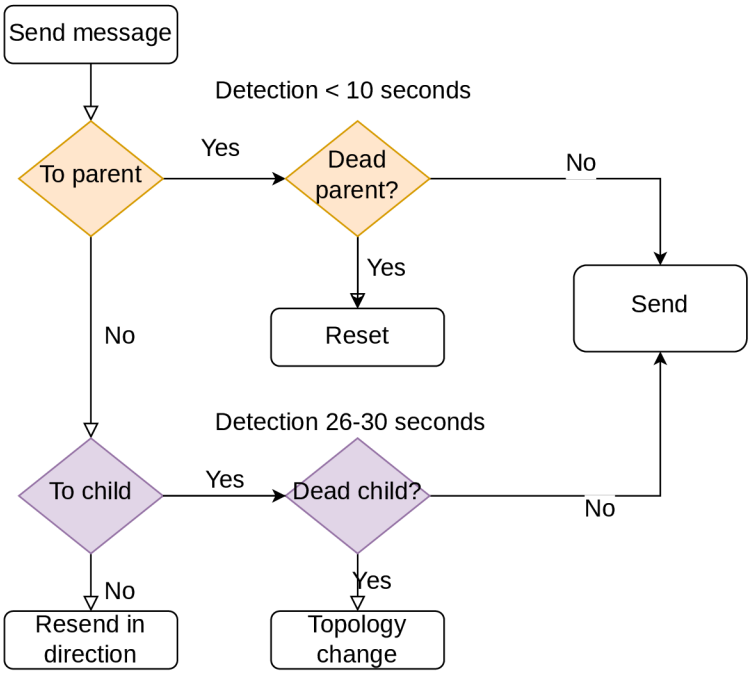


Figure 7.2: The project consists of tree major modules. Each module is responsible for subset of working mesh. For mesh to start working, user simply has to init WifiCore and run its start function which takes care of everything mesh related.

Nodes send topology in periodic intervals every 7 seconds to their child nodes. The root node sends to his children, and the children send to theirs and so on. This could take time until tree change is detected. For this purpose, the topology changes (new and dead nodes) are propagated immediately. The values of 7 seconds for topology propagation together with periodic advertisements in EspNowCore and neighbour advertisement every 13 seconds, were experimentally chosen. The intention of these values was to select prime numbers in a hope that the intervals would interleave one another as little as possible so the burst on the node would not be as high as it would with even numbers.

Child nodes are notified of the dead server port of their parent after approximately 10 seconds and they reset themselves in order to be able to create a new parent connection on the same IP address and the same port number. Otherwise, a new socket connection would not be able to open. Unfortunately detecting dead child nodes is not an easy task. This cannot be detected using sockets on the parent side in MicroPython nor using a listing of connected devices to the WiFi AP interface. Instead of that, I use information from the base layer of EspNowCore. With sending messages to child nodes, I test that the is_peer_alive function. If peer is in a database of nodes in EspNowCore.neighbours I know that the peer has been active. If the MAC address is not listed there, it indicates that the peer didn't send an advertisement for 26 seconds. Therefore in theory dead child node should be detected after 26 seconds, but due to switching of coroutines and awaiting instruction and delays, it takes about 30 seconds to detect a dead child node. The flow diagram of detecting a dead peer is shown in Figure 7.2. Every descendant node of the dead node is purged from the tree topology with the aim to claim them on a new position in the tree and a new topology is propagated.

With every topology change, the routing table (or switching table) is updated. Nodes process messages only destined to them and broadcast messages. Other messages are resent in the right direction to the destination node. The user can define an application in which the messages are destined to specific nodes and mesh would be able to route the packets to the specific destination. The application can use two functions for sending packets listed below in 7.3

```
    async def send_to_nodes(self, msg, nodes=None):
        """ Send to directly connected nodes. Used for broadcast messages"""
        if nodes is None:
            nodes = [self.parent] + list(self.children_writers.keys())
        for node in nodes:
            writer = self.get_writer(node)
            await self.send_msg(node, writer, msg)

    async def send_to_all(self, msg):
        """ Send directly to every node in the mesh."""
        nodes = self.tree_topology.root.get_all() + \
                [self.tree_topology.root.data]
        nodes.remove(self.id)
        for node in nodes:
            msg.packet["dst"] = node
            await self.resend(msg, msg.packet)
```

Listing 7.3: Two functions for sending the application data are defined.

7.4 Demo Application

For purpose of verifying and testing that the mesh network works and nodes are interconnected, I developed a small application. This application controls the LED and changes the colour of the diode. This application serves as a demo and a basic template can be seen in listing 7.4

As ESP32-Buddy has three pushable buttons, one is for hard reset and one for MPS procedure, the board has one button left. I use this button on pin 32 to trigger a change of colour. The interrupt is registered to this button press and interrupt executes function blink. In initialisation, every board randomly selects its colour for all the time. When this button is pressed, the node sets its colour to its init colour and sends the order to every other node to set their colour of LED to the same. This means that we can see how the nodes change their colour in real-time as the orders flow through mesh and route to every other node.

Class for application has to define App.process function to process the application packets. It is also necessary that before application communication, the WifiCore class needs to be initialised and the function WifiCore.start needs to be run. The app can function without a WiFi connection yet established but the information would not be sent to the nodes.

```
class BlinkApp:
    def __init__(self):
        self.wificore = WifiCore()
        ...

    def start(self):
        self.wificore.start()
        ...

    async def process(self, AppMsg): # Function must be named like this.
        ...

    async def blink(self):
        ...
        msg = AppMessage(self.wificore.id, "ffffffffffff", {"blink":colour})
        await self.wificore.send_to_nodes(msg)
        ...
```

Listing 7.4: Source code of a demo application BlinkApp class for LED blinking. For user defined application, user has to init WifiCore and run its start function. The function process has to be defined.

I used USB-HUB station with 7 USB ports to connect and run several boards at the same time as is represented in Figure 7.3. I was provided two of these station and overall 14 ESP32-Buddy boards from Espressif company. For development purposes I used the USB-HUB station and for testing I used more real scenarios.

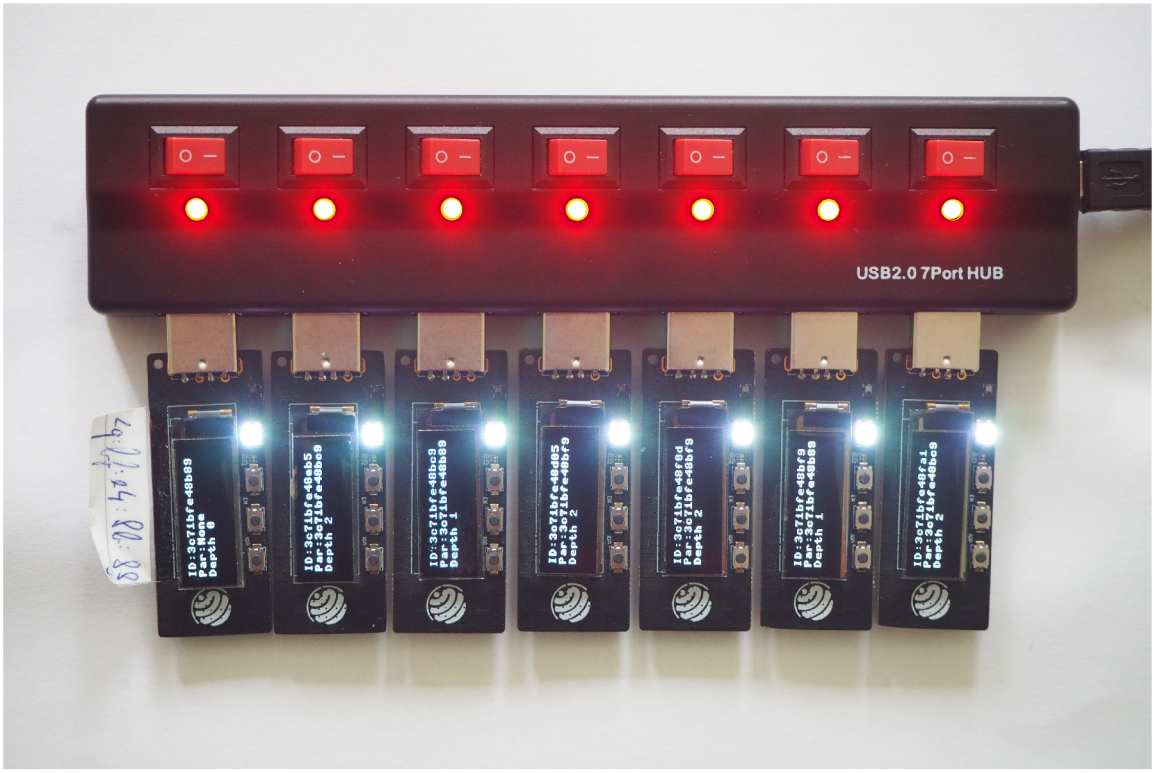


Figure 7.3: USB-HUB station with 7 ports used for running the mesh network on ESP32-Buddy boards with demo application during development. LEDs with the same colour indicates that the mesh is working and the colour was propagated to all the nodes.

7.5 Other useful modules

For the purpose of this project, a few additional modules were created to support the overall functionality of both cores. Among these modules are:

hmac.py which implements HMAC class for message signing. It uses the SHA256 hash algorithm. This module was not written for this thesis. It was taken from GitHub repository [21]. Nodes process messages that are signed with the same mesh key.

messages.py module defines classes for messages both in ESP-NOW and WiFi communication and defines functions for packing and unpacking messages to send them properly. Messages for ESP-NOW have a strictly defined structure and are packed using struct library to pack messages into bytes. It uses less space than WiFi messages. To the packed ESP-NOW messages, EspNowCore adds a sign with mesh key credential which is always 32 bytes long. WiFi messages are packed using JSON for serialisation which was inspired by PainlessMesh solution [7]. There is defined one class for user application which is AppMessage. The initialisation takes three parameters. Source and destination MAC address and the message itself. The message has to be a dictionary or an already packed JSON object.

net.py file implements two classes that overshadow original *network.WLAN* and *espnow.ESPNow* classes. They implement only some subset of original functions and are used in both cores. Original classes are still accessible through properties of these overshadowing classes, but in the project, I use only the newly defined functions to overshadow low-level implementation.

oled_display.py is a module for manipulation and writing to OLED display. This module was taken from the tutorial HowToElectronics [4]. The OLED display is used by WifiCore to print basic information about the position of the node in the tree. MAC address together with parent node is printed. It also prints the depth of the node in the tree. Demonstration of use od OLED display is shown in Figure 7.4.

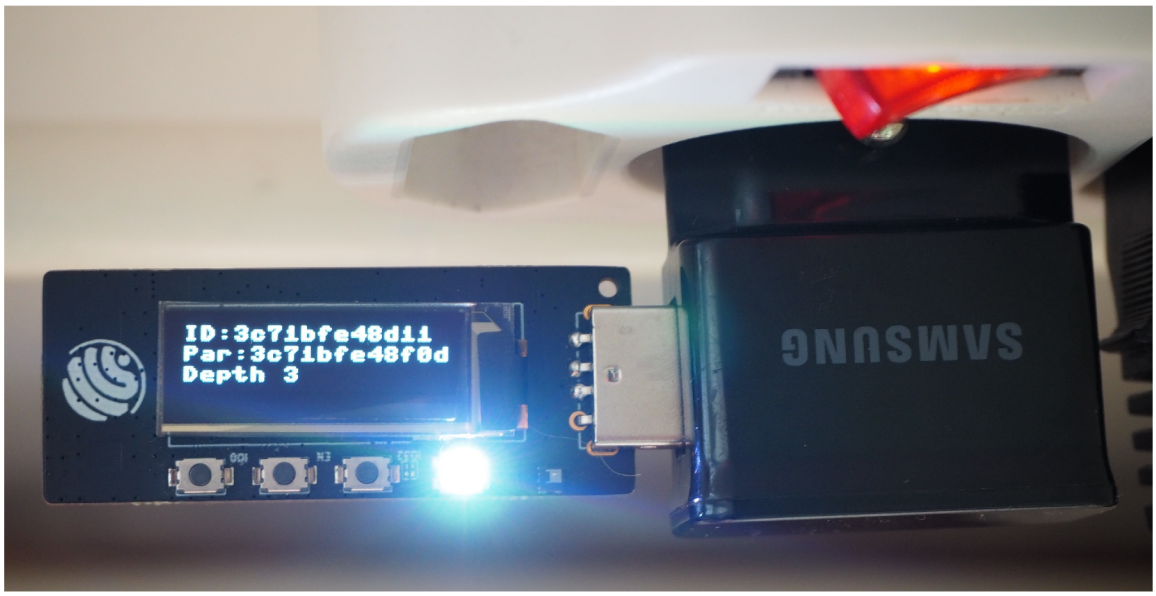


Figure 7.4: OLED display on ESP32-Buddy boards is used to show MAC address of the node (*ID*), MAC address of the parent node (*Par*) and depth in the tree the node is (*Depth*).

pins.py file defines numbers of GPIO pins and initialisation of LED diode. It also offers the function to initialise the button on ESP32 and register an interrupt handler. These handler functions are used for processing the push button interrupt in the MPS process and in the BlinkApp application.

tree.py is a module for Tree and TreeNode classes. The Tree class represents the tree topology and is saved on each node in the tree. It also supplies functions that can create a Tree instance from a JSON object.

7.6 Use and deployment

How to use this project was shown in Demo application 7.4. For automatic initialisation and the start of the project it is useful to use boot.py and main.py that executes at the boot of the device. A basic overview of the deployment of the user programs to microcontrollers can be found at the official website of MicroPython [10] or on Random Nerd Tutorials [24]. The big advantage is that when the board reset itself, it automatically reads instructions from boot.py and main.py and can start working again without the need to manually reset or upload the program again.

After 29 seconds of no new addition through MPS into the mesh, the root is elected which takes some amount of time. Then at worst every 7 seconds new layer of child is added. In total 29+7*L seconds where L is the height of the tree. Be aware that connection to WiFi AP of parent node takes unknown amount of time.

Chapter 8

Testing and limitations

Users must be aware, that importing modules consumes memory. When the module is compiled into the bytecode, the bytecode is stored in RAM and the compilation itself requires memory. After importing all the needed modules and initialising EspNowCore, WifiCore and BlinkApp, the memory usage is around 56KB out of 111KB. This size could be reduced by memory profiling and improvements [11] like using module *array*[1] or *memoryview*[2].

During the implementation phase, I used the USB-HUB station as shown in Figure 7.3. But when nodes are so close to each other, it could lead to misbehaviour. In order to test real-life scenarios, I distributed nodes around the house and tried to run the mesh with more real-life use cases. The distribution should simulate the distances and signals of nodes. ESP32-Buddy board can be plugged into any USB port, I use mobile chargers, laptop and USB-HUB station.

In the testing scenarios, I defined the maximum number of child nodes for each node as two, in order to demonstrate tree structure. ESP32 boards can maintain at most 10 connections on their AP interface.

8.1 Tests

Six nodes for 24 hours

First use case was with six nodes that run over one day. Firstly, all six nodes were plugged into the USB-HUB station and debug outputs in REPL consoles were watched. The debug prints in the configuration file can enable such information as received messages and send messages and additional actions. After 15 minutes with a stable and working mesh, the nodes were turned off and distributed around the house. Only two nodes were left in the USB-HUB and every other node was stationed in a different room. One node was always connected to the laptop in order to monitor debug prints in the REPL console.

For the next 24 hours, the tree structure and stability of nodes were mapped. During this time no node experienced node failure and the mesh network was stable. The tree structure was marked down as it was, and after the night it was checked with the current tree structure again. If the mesh had failed or collapsed, it would have reconstructed itself very likely in a different constellation because claiming of children and forming of the tree structure is randomised. This way, it can be said with certainty, that the mesh was stable

[1] https://docs.micropython.org/en/v1.10/library/array.html

[2] https://docs.micropython.org/en/v1.10/library/builtins.html?highlight=memoryview#memoryview

and didn't re-organise itself. Nodes were intentionally turned down in order to verify the process of self-healing as can be seen in Figure 8.1. The mesh managed to repair itself even with multiple turned down nodes. And also when multiple nodes turned back on. Throughout the day, the change of LED colour was triggered on several nodes to ensure the nodes are really connected and resending packets.

When a node is turned down, it has to stay down for at least 26-30 seconds for the mesh to self-heal. After the node goes down, its child nodes discover its parent is dead after less than 10 seconds and they restart. But the child nodes can function on the ESP-NOW layer, but in order to get back to the WiFi tree structure, they have to wait for about 26-30 seconds because the grandparent discovers the dead parent after 26-30 seconds and only after that time the parent is erased from the whole tree together with its descendants. Therefore the child nodes are free to be claimed by other nodes.

When the root node was turned down, after 8 seconds all the nodes restarted themselves and waited to be claimed. When a node detects a dead parent node, it tears down its connection to its child nodes and based on that the child nodes restart themselves immediately (they do not have to wait for additional 8 seconds to detect their dead parent node). Because root election is manually set, they did not elect a new root node, but the root node had to be turned back on.

```
[OnTopologyPropagate]
 '3c71bfe48b89'
        '3c71bfe48d11'
                '3c71bfe48bb9'
                '3c71bfe48d05'
        '3c71bfe48ebd'
                '3c71bfe48eb5'
```

(a) Tree structure

```
[OnTopologyPropagate]
 '3c71bfe48b89'
        '3c71bfe48ebd'
                '3c71bfe48eb5'
```

(b) *3c71bfe48d11* turned off for at least 30 seconds

```
[OnTopologyPropagate]
 '3c71bfe48b89'
        '3c71bfe48ebd'
                '3c71bfe48eb5'
                '3c71bfe48bb9'
        '3c71bfe48d05'
```

(c) Descendants reclaimed after 26-30 seconds

```
[OnTopologyPropagate]
 '3c71bfe48b89'
        '3c71bfe48ebd'
                '3c71bfe48eb5'
                        '3c71bfe48d11'
                '3c71bfe48bb9'
        '3c71bfe48d05'
```

(d) *3c71bfe48d11* turned on and claimed

Figure 8.1: Demonstration of self-healing when in full tree topology one node is turned down, mesh self-heals and claims dead node's descendants. Then the dead node is turned back on and claimed back into the tree.

Seven nodes for 24 hours

Testing with seven nodes was a little bit complicated. The mesh at first worked successfully. But some nodes randomly restarted themselves and deformed the mesh. When connected to the REPL interface on my laptop, the Guru Meditation Error [38] mainly caused by **Load-Prohibited** exception was printed in the REPL. This exception occurs when a pointer to an object is corrupted 8.2. Using the backtrace information, I was able to repair most errors,

but not all and the problem remains in lower frequency. Another Guru Meditation Error was **IlegalInstruction** which indicates wrong instruction in Program Counter Register.

```
[SEND] drained and done
Guru Meditation Error: Core  0 panic'ed (LoadProhibited). Exception was unhandled.

Core  0 register dump:
PC      : 0x400da389  PS      : 0x00060030  A0      : 0x80083434  A1      : 0x3ffccab0
A2      : 0x00000002  A3      : 0x3ffec3e0  A4      : 0x3fffd2b0  A5      : 0x0000000e
A6      : 0x0000001e  A7      : 0x00000000  A8      : 0x00000008  A9      : 0xdd304e01
A10     : 0x3f4048d0  A11     : 0x3ffccaf0  A12     : 0x3ffccae0  A13     : 0x3f402670
A14     : 0x00000010  A15     : 0x00000008  SAR     : 0x0000000a  EXCCAUSE: 0x0000001c
EXCVADDR: 0xdd304e19  LBEG    : 0x400827c1  LEND    : 0x400827c9  LCOUNT  : 0x00000027

Backtrace:0x400da386:0x3ffccab0 0x40083431:0x3ffccae0 0x40084983:0x3ffccb20 0x400dca91:0x
50 0x400e3179:0x3ffccc70 0x400845f9:0x3ffccc90 0x400dc6e8:0x3ffccd30 0x400e3049:0x3ffccd6
0845f9:0x3ffcceb0 0x400dc6e8:0x3ffccf50 0x400e3049:0x3ffccf80 0x400e3179:0x3ffccfa0 0x400
0x3ffcd0d0 0x400dc6e8:0x3ffcd170 0x400e3049:0x3ffcd1a0 0x400e3072:0x3ffcd1c0 0x400f03d7:0


ELF file SHA256: 009e7c06444b5b69

Rebooting...
ets Jun  8 2016 00:22:57
```

Figure 8.2: Guru Meditation Error Load Prohibited occurs when seven and more nodes resonance together and cause packet burst on some node.

The problem could be caused by low memory size. With seven boards starting at once, they are synchronised and send messages at the same time like a burst. This means that the boards receive many messages at once, and there is probably low memory to save all of them and process them at once. This is probably caused by my implementation of creating a task for each message in order to have the receive message function as light-weight as possible 7.2 or by the wrong MicroPython implementation.

Errors and restarts occur randomly throughout the life of the mesh, but mainly in the beginning. Later when the mesh is formed, the restarts occur very sparely. From the second half-hour of functioning, the errors occur minimally (I noticed on average two errors in one hour). This means that the mesh found its stable state and the bursts of messages were lower.

When the restarts happen after the tree is formed, there is a problem with dead node detection and self-healing, because, in order to detect a dead child node, it has to stay dead for at least 26 seconds. But nodes restart themselves immediately. Which leads to wrong or time-long detection of dead connection. When the node is not dead for 26 seconds, the information about its failure is known from the socket instance. But detecting a dead child node through the socket takes from 2:20 to 2:40 minutes according to my measures. This means that the failed node recovers, and its descendants reset themselves, but they all must wait for 2:30 minutes in order to be wiped from the tree topology and to be claimed again in a different place in the tree.

Eight and more nodes

With more than seven nodes, the problems with Guru Meditation Error grow stronger and stronger. With eighth nodes, the problem is so severe, that the mesh and nodes cannot find

a calm state and they experience Guru Meditation Error throughout the whole life of the mesh very often. The mesh is stable for a while, nodes form a tree, but then some of them restart. I measured that the full eighth node tree was working at most for 15 minutes until some of the nodes break down.

I tried to test the mesh with ten nodes as was requested in the assignment but unfortunately, with this many nodes, the mesh is not stable at all and nodes reset themselves very often and randomly. Overall with ten nodes, there was definitely more time spent on rebuilding the tree and reconnecting due to the errors than the mesh spent time functioning with all the nodes as it should.

8.2 Comparison with existing solution

The ESP32-Buddy boards are not commercially available. They were produced only for the development of the ESP-WIFI-MESH 5.2 on ESP32 boards. Even though there are set exact intervals to trigger some actions, they occur around the time and not exactly in a given time, because delays and asyncio operations can postpone some actions. I run several time measurements and the time results are that:

- 6-8 seconds from turning on the board to starting the program (importing modules and libraries, initialisation of network and ESP-NOW interfaces).
- 35 seconds until the root election process is simulated.
- 7,5-8 seconds until the node sends the first claim to the possible child node.
- 5-8 seconds until the child connects to the parent's WiFi and opens the socket connection(time to connect to the WiFi AP can vary quite a lot).
- 26-30 seconds until the dead child node is detected using ESP-NOW advertisements database of nodes.
- 8-10 seconds until the dead parent node is detected using sockets.
- 2:20-2:40 minutes until the dead child node is detected when the child immediately restarts itself. It is detected by socket timeout on server side (parent node).

For the child to be considered dead, it must stay dead for 26 seconds in order to be wiped out from the tree. If the child fails down and comes back up right away without the needed 26 seconds period, the mesh notices the wrong or failed WiFi connection thanks to the socket after much longer.

From the results above, the mesh elects its root after 41-43 seconds. This time could be reduced to only 6-8 seconds because the election is only simulated and is statically set in the configuration file. With every 12,5-16 seconds a new layer (level) of nodes can be added to the tree. I measured the per-hop latency of WiFi messages to an average of 174 milliseconds.

Let's assume that my design can take up to 100 nodes in order to make a comparison to ESP-WIFI-MESH 5.2 and see the results in Table 8.1. There is a comparison with time measures. It tells us that MicroPython cannot measure in speed point of view, at least not normal MicroPython (there are special code emitters to speed up execution[3]). But be

[3] https://docs.micropython.org/en/latest/reference/speed_python.html - special Native or Viper code emitters can speed up some suitable code several times.

aware that this is merely a simulation and times for building the mesh and the per-hop latency would probably increase with more load on the node, due to the switching of tasks and scheduling. My solution can effectively work with only six nodes, due to the memory limits and ESP-WIFI-MESH has 36x more memory than my boards.

Table 8.1: Lets assume 100 nodes with max 6 children for each node and max 6 layers in the mesh (depth) for comparison with existing solution ESP-WIFI-MESH [31].

Category	MicroPython Mesh	ESP-WIFI-MESH
Build time	103.5-123 s	< 60 s
Healing Time	26-30 s	< 5 s
Per Hop latency	174 ms	10-30 ms

I have computed rate of bits/seconds and number of messages send by each node.

$$rate(bit/second) = 117,2 + 31,9 + (74,8 + 34,3 * N) * C \tag{8.1}$$

$$rate(packets/minute) = 20,5 + 4,6 * N + 8,5 * C \tag{8.2}$$

Where N represents the total number of nodes in the mesh and C represents the number of child nodes. With a working mesh with 10 nodes and a max of 2 children, the rate is *984,7 bps* and 83,5 packets per minute on each node. Together the 10-mesh produces *9847* bps and *835* packets per minute. Low rates are because the ESP-NOW advertise packets are small (52B) and topology propagation of 260 Bytes is sent only 8,5x in one minute. These values are for mesh working only, additional load is added by the user application and messages it sends.

8.3 Limits and improvements

The biggest limitation of this solution is the memory. ESP32-Buddy microcontrollers have 4MB of RAM but due to the wrong compatibility with MicroPython, only 111KB of RAM is accessible. The modules themselves without any runtime activity take about 55KB of memory. Probably due to the RAM limits, the solution is stable only for a maximum of six nodes in the mesh. With seven nodes the mesh is formed after some time of restarts. For more than 8 nodes the mesh is not stable and the formation and repairs of the mesh consume more time than the mesh working itself with all the nodes.

The root election is set statically because there is a problem with WiFi scanning networks, which takes between 2 and 2,5 seconds. Even though in MicroPython WiFi scan is defined in another thread, in RTOS it runs in the same thread as receiving incoming packets, therefore, it blocks the receiving. I came up with a solution to overcome this issue, that the root election would not be based on the RSSI signal of neighbours but simply on a number of visible neighbours (using TTL or Age flag). I am aware that this type of root election can lead to electing some locally optimal root, but it is a better solution in most cases than a random root election, because it tries to form a shallower tree.

Detection of dead connection needs to be rethought and improved in order to know dead peer in a shorter time and to be able to detect dead peer even after it boots up and starts propagating itself again in ESP-NOW protocol.

The LMK and PMK key for secure ESP-NOW communication during the MPS process of exchange signing credentials must be predefined in the JSON configuration file because these values have to be the same on both devices.

From the low-level implementation of ESP-NOW and network interfaces, they must operate on the same channels. This means that when the mesh is in connected mode to the Router, all the nodes are on the same channel as other users' devices. Even though node sends not many packets, when there are more nodes, they send many little packets and this could lead to delays in transmitting due to CSMA/CA[4] because devices would always wait for a free channel.

ESP32 boards can offer a maximum of 10 stations to be connected to one AP interface. This means that the maximum number of child nodes is limited to 10 nodes. Also randomly claiming children can lead to the isolation of some nodes, because the parent node can claim whichever node he wants. But some nodes can see only this particular parent node. When the parent randomly chooses nodes, it could leave the node isolated without any other node to claim it because no one else sees it directly (a little similar to PainlessMesh root problem 5.7). This is probably not going to happen in home uses, but it is possible in use cases with bigger distances (farms, warehouses,...).

A little improvement would be inserting a Web server like microdot [16] on at least one node in order to set mesh key and WiFi router credentials dynamically during the runtime, instead of changing the configuration file and uploading it to the board.

There haven't been any power consumption measurements to know how long can device operate on a battery, but WiFi operations consume a lot of energy. The WiFi communication is a power bottleneck [12], as the Rf antenna is used constantly. This can be improved by implementing NTP[5] protocol for devices to come alive at certain intervals and sleep in the meantime.

[4]Carrier-sense multiple access with collision avoidance (CSMA/CA) is a method for wireless devices to avoid collisions on the channel by beginning transmission only after the channel is sensed to be idle. When the node transmits the packet, it transmits the data in its entirety.

[5]Network Time Protocol is used for time synchronisation.

Chapter 9

Conclusion

The goal of this thesis is to create a dynamic mesh network using ESP32 microcontrollers for IoT and sensor networks that are capable of working with or without an Internet connection. The new mesh network protocol was designed and implemented in MicroPython as was requested by the assignment from the company Espressif. I have chosen this mesh should form a tree structure for better management. Together with the tree, there is a need for a root node. Mesh can overcome node failures.

In this work, the current solutions were studied and based on the information a new mesh network was designed. It supports both modes, stand-alone and connected to an external network. A firmware with MicroPython together with an asynchronous loop primarily for network I/O operations was created. Furthermore, a demo application has been created to demonstrate the functionality of the solution. It is important that user-defined applications can be created, according to the demo, to run on the mesh network for sensor and management purposes.

To test the effectiveness and functionality, the tree testing scenarios have been completed with mesh deployment in a home environment. During the implementation on ESP32 boards, I encountered problems with the compatibility of firmware and versions of MicroPython. It results in a low capacity of memory. This limited the use case scenarios to be able to work without trouble only on 6 nodes, not 10 as it is requested in the assignment. Further optimisations can make this implementation more efficient. However, MicroPython solution probably cannot achieve robustness with 1000 nodes of other solutions using C or C++ languages.

Collaboration with an external consultant was very beneficial and throughout this project, I learned how to develop complex network systems from scratch. This work was also presented at the Student's conference of innovations Excel@FIT in Brno. I have won two diplomas. One from a professional panel for the creation of practical useful results and the second one is from the company Espressif for extraordinary work. For this conference, I have written a scientific paper and prepared a presentation in front of an audience.

I would like to continue my work and improve it because it could be useful for IT hobbyists and I made it publicly available on the platform GitHub. A big improvement would be solving problems with compatibility on firmware and MicroPython, in order to use full memory capacity. My work could be improved by implementing the proposed automatic root election process. Another improvement would be creating an algorithm for adding child nodes for the purpose of not creating isolated nodes.

Bibliography

[1] IEEE Standard for information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 6: Medium Access Control (MAC) Security Enhancements. *IEEE Std 802.11i-2004*. 2004, p. 1–190. DOI: 10.1109/IEEESTD.2004.94585.

[2] IEEE Standard for Information Technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications Amendment 10: Mesh Networking. *IEEE Std 802.11s-2011 (Amendment to IEEE Std 802.11-2007)*. 2011, p. 1–372. DOI: 10.1109/IEEESTD.2011.6018236.

[3] IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*. 2016, p. 1–3534. DOI: 10.1109/IEEESTD.2016.7786995.

[4] Admin. *MicroPython: Interfacing 0.96„ OLED display with ESP32*. Mar 2021. Available at: https://how2electronics.com/micropython-interfacing-oled-display-esp32/.

[5] Anderson, J. *Speed Up Your Python Program With Concurrency – Real Python*. Available at: https://realpython.com/python-concurrency.

[6] Conner, W. S., Kruys, J., Kim, K. and Zuniga, J. C. *IEEE 802.11s Tutorial* [Overview of the Amendment for Wireless Local Area Mesh Networking]. 2006. Available at: https://www.ieee802.org/802_tutorials/06-November/802.11s_Tutorial_r5.pdf.

[7] Coopdis, E. v. L. *PainlessMesh* [release/1.4.9]. GitLab, 2019 [Online]. Available at: https://gitlab.com/painlessMesh/painlessMesh/-/wikis/home.

[8] Damien P. George, P. S. *MicroPython Documentation*. 2021. Available at: http://docs.micropython.org/en/v1.10/micropython-docs.pdf.

[9] Damien P. George, P. S. *Cryptolib – cryptographic ciphers*. May 2022. Available at: https://micropython-glenn20.readthedocs.io/en/latest/library/cryptolib.html?highlight=aes.

[10] Damien P. George, P. S. *Getting started with MicroPython on the ESP32*. Apr 2022. Available at: https://docs.micropython.org/en/latest/esp32/tutorial/intro.html.

[11] Damien P. George, P. S. *Maximising MicroPython speed*. Apr 2022. Available at: https://docs.micropython.org/en/latest/reference/speed_python.html.

[12] Engineers, L. M. *Insight into ESP32 sleep modes & their power consumption*. Last Minute Engineers, Dec 2020. Available at: https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/.

[13] Espressif Systems, E.-M. f. *Mesh size limts*. 2018. Available at: https://esp32.com/viewtopic.php?t=5919.

[14] glenn20. *MicroPython port to the ESP32 - Glenn20/micropython*. Available at: https://github.com/glenn20/micropython/tree/espnow-g20/ports/esp32.

[15] Gomez, C. and Paradells, J. Wireless home automation networks: A survey of architectures and technologies. *IEEE Communications Magazine*. 2010, vol. 48, no. 6, p. 92–101. DOI: 10.1109/MCOM.2010.5473869.

[16] Grinberg, M. *The impossibly small web framework for Python and MicroPython*. 2019. Available at: https://github.com/miguelgrinberg/microdot.

[17] Henry, J. and Burton, M. *802.11s mesh networking*. 2011.

[18] Hunt, J. *Concurrency with AsyncIO*. Springer International Publishing, 2019. 407–417 p. ISBN 978-3-030-25943-3.

[19] Kolář, J. *Koordinace IoT na bázi MicroPythonu pomocí Node-RED*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/21632/.

[20] Krawczyk, H., Bellare, M. and Canetti, R. *HMAC: Keyed-hashing for message authentication*. RFC Editor, 1997. DOI: 10.17487/RFC2104. Available at: https://www.rfc-editor.org/info/rfc2104.

[21] Mazzella, D. *Micropython package for doing fast elliptic curve cryptography, specifically digital signatures*. 2021. Available at: https://github.com/dmazzella/ucrypto.

[22] Ramakrishnan, Karthik. *An Improved Model for the Dynamic Routing Effect Algorithm for Mobility Protocol*. Ontario, Canada, 2004. Master's thesis. University of Waterloo.

[23] Robin Heydon, V. Z. Mesh profile. *Bluetooth® Specification*. Mesh Working Group. 2017.

[24] Santos, S. *Getting started with MicroPython on ESP32 and ESP8266*. May 2019. Available at: https://randomnerdtutorials.com/getting-started-micropython-esp32-esp8266/.

[25] Santos, S. and Santos, R. *Esp32 useful Wi-Fi library functions (Arduino IDE).* Available at: https://randomnerdtutorials.com/esp32-useful-wi-fi-functions-arduino/#1.

[26] Solomon, B. *Async IO in Python: A Complete Walkthrough – Real Python.* Available at: https://realpython.com/async-io-python.

[27] Systems, E. *ESP-BLE-MESH.* Available at: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/esp-ble-mesh/ble-mesh-index.html.

[28] Systems, E. *ESP-Now.* Available at: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_now.html.

[29] Systems, E. *ESP-Now Overview: Espressif Systems.* Available at: https://www.espressif.com/en/products/software/esp-now/overview.

[30] Systems, E. *ESP-NOW User Guide.* Available at: https://www.espressif.com/sites/default/files/documentation/esp-now_user_guide_en.pdf.

[31] Systems, E. *ESP-WIFI-MESH.* Available at: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/esp-wifi-mesh.html.

[32] Systems, E. *ESP32-Buddy.* Available at: https://docs.espressif.com/projects/esp-mdf/en/latest/hw-reference/esp32-buddy.html.

[33] Systems, E. *ESP32 Buddy User Guide.* Available at: https://github.com/espressif/esp-mdf/tree/master/examples/development_kit/buddy.

[34] Systems, E. *ESP32 Series Datasheet.* Available at: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[35] Systems, E. *Espressif's AWS IOT Expresslink solution.* Available at: https://www.espressif.com/.

[36] Systems, E. *Mesh Development Framework - ESP-WIFI-MESH.* Available at: https://www.espressif.com/en/products/sdks/esp-wifi-mesh/overview.

[37] Systems, E. *Wi-Fi:Frequently asked questions.* Available at: https://docs.espressif.com/projects/espressif-esp-faq/en/latest/software-framework/wifi.html.

[38] Systems, E. *Fatal Errors.* May 2022. Available at: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/fatal-errors.html#id3.

[39] Waqas, A. and Kashif, A. M. *An investigation of Routing Protocols in Wireless Mesh Networks (WMNs) under certain Parameters.* Karlskrona, Sweden, 2009. Master's Thesis. Blekinge Institute of Technology, Karlskrona Campus, Sweden.

[40] Woolley, M. *Bluetooth Mesh networking.* Bluetooth SIG, 2020. Available at: https://www.bluetooth.com/wp-content/uploads/2019/03/Mesh-Technology-Overview.pdf.

Appendix A

Contents of the DVD

The enclosed DVD medium contains the following files:

- doc/ - source files of this text for Diploma thesis.
- micropython_616.zip/ - copy of GitHub repository from glenn-g20/ branch of MicroPython with working ESP-NOW support on ESP32-Buddy boards. This version is probably re-based and unavailable.
- codes/ - source codes of implementation in MicroPython together with Readme file with a guide.
- thesis.pdf - PDF of this thesis.