



Technion- Israel Institute of Technology
Electrical and Computer Engineering
NSSL – Networked Software Systems Laboratory

Project Report

IoT Communication Network
A computer network of tiny computers

Ali Sweed and Suray Sweed

With supervisor Roy Mitrany

Winter 2022

Abstract

Many forests have been destroyed and harshly damaged because of the fire outbreak. As a result, a monitoring system to predict fire outbreaks and send a warning or image is needed. So, this project aims to establish and manage a network among tiny computers (esp32 cam boards) scattered in the forest. When an esp32 board detects a potential fire outbreak, an image is taken and derived through the esp's network till arrived to the user.

The network established by the tiny computers is an ESP-MESH network, which contains mesh clients and one mesh root, the clients and the root are ESP32-CAM boards. The root got the warning image from one of the clients and send it to a server listening on the user's computer.

The MESH network is established using the painlessMesh library, and the messages are sent in Jason format.

Table of contents

Abstract	2
Table of figures	5
Table of abbreviations	7
Introduction	8
Chapter 1: Environments	9
1.1 Arduino environment	9
1.2 GitHub	13
Chapter 2: building the hardware circuit	14
2.1 introduction – the HW circuit	14
2.2 ESP32-CAM processor	14
2.2.1 ESP32 processor	15
2.2.2 ESP32-CAM processor	15
Chapter 3: Wireless mesh network	17
3.1 Abstract	17
3.2 Introduction	17
3.3 IEEE 802.11s wireless mesh network standard	21
3.4 PainlessMesh library – ESP mesh Wi-Fi	22
3.4.1 introduction	22
3.4.2 Data Transmission	23
3.4.2.1 ESP Wi-Fi Mesh Packet	23
3.4.2.2 Bi-Directional Data Stream	23
3.4.3 ESP Wi-Fi Mesh Programming	30
3.4.3.1 Software Stack	24
3.4.3.2 LwIP & ESP WiFi Mesh	23
3.5 Json Format	24
3.6 TaskScheduler Library	25
Chapter 4: Warning Test Implementation	26
Chapter 5: Running and Results.....	31
4.1 Assessing Image Transmission Quality in ESP Mesh Network.....	31
4.2 Packet Size Variation Test.....	33
4.3 Alerts Results	34

Chapter 6: Conclusion 38

Chapter 7: Suggestions to improvements 39

Chapter 8: References40

Table of figures

Figure 1: choosing a board in Arduino IDE	8
Figure 2: new sketch in Arduino IDE	8
Figure 3: choosing a port for uploading in Arduino IDE	9
Figure 4: the serial monitor in Arduino IDE	10
Figure 5: library manager for downloading libraries in Arduino IDE	10
Figure 6: project's git	11
Figure 7: hardware circuit	12
Figure 8: ESP32-CAM processor	13
Figure 9: ESP32-CAM pins layout	13
Figure 10: Mesh Network Architecture	15
Figure 12: A typical architecture of WMN	16
Figure 13- Mesh Network Node Types	24
Figure 14: ESP-WIFI-MESH Routing Tables Example	25
Figure 15: ESP-WIFI-MESH Network Building Process	26
Figure 16: ESP-WIFI-MESH Root Node Designation	27
Figure 17: ESP-WIFI-MESH packet and its relation with a Wi-Fi data frame	28
Figure 18: ESP-WIFI-MESH Bi-Directional Data Stream	29
Figure 19: ESP-WIFI-MESH Software Stack	30
Figure 20: ESP-WIFI-MESH System Event Delivery	30
Figure 21: ESP-WIFI-MESH Application code	31
Figure 22: ESP-WIFI-MESH Initialize Mesh	31
Figure 23: ESP-WIFI-MESH Network Configuration	32
Figure 24: ESP-WIFI-MESH start	32
Figure 25: client	35
Figure 26: master	35
Figure 27: server & web	35
Figure 28: client 1	37
Figure 29: master	37
Figure 30: Server	37
Figure 31: client #1	38

Figure 32: client #2	38
Figure 33: master	38
Figure 34: Server	38
Figure 35: client #3	39
Figure 36: master	39
Figure 37: Server	39
Figure 38: setup infrastructure	40
Figure 39: client & master screen	40
Figure 40: server's screen	41
Figure 41: Software Flow Chart	42

Table of abbreviations

IDE - Integrated Development Environment.

IoT – internet of things

Node - Any device that can be part of a Mesh Network.

Root/Bridge Node - The top node in the network.

MANET – mobile ad hoc network

WMN – wireless mesh network

MR – mesh router

IGW – internet gateway

IEEE – institute of electronics and electrical engineering

WLAN – wireless local area network

ISP – internet service provider

TG – task group

MSTAs – mesh stations

MBSS – mesh basic service set

AP – access point

BSSID - basic service set identifier

ESS – extended service set

HWMP – hybrid wireless mesh protocol

PREQ – path request

PREP – path reply

PERR - path error

RANN – root announcement

QoS – quality of service

RM-AODN – radio metric ad-hoc on demand vector

LwIP – lightweight IP

Introduction

The new era of the Internet of Things (IoT), has already arrived in corporate and home networks, as well as in video streaming, data transmission etc. IoT devices can easily be remotely controlled and monitored using wireless communication interfaces.

With the increase in the number of devices, it becomes difficult to maintain the availability of wireless networks due to increased mutual interference and noise, so the actual problem is to create mesh networks, where the node does not need to be connected to the switch.

Chapter1: Environments

Arduino environment:

The Arduino IDE environment is an open-source software platform used for programming and developing electronic projects. It provides an environment that users can use to write, compile, and upload code for Arduino boards.

Arduino environment is user-friendly and easy to use, so it's good for beginners and provides advanced features that fit more experienced users.

The environment includes an integrated development environment (IDE) that provides a text editor, code highlighting, and debugging tools. It also includes a set of libraries that provide pre-written code for commonly used functions, such as controlling LEDs or reading sensors

Why Arduino?

- The Arduino environment makes it easy for students and hobbyists to get started with electronics and programming.
- Using c language which is easy to use.
- Arduino environment is open source with a large community of users and developers contributing to its development. Also, it's free to download.

Guide for using Arduino IDE:

A Typical Workflow

To upload code to an Arduino board using the IDE, one typically does the following:

1. Install your board - this means installing the right "package" for your board. Without the package, you can simply not use your board. Installing is done directly in the IDE, and is a quick and easy operation.

Tools -> boards -> chose a board.

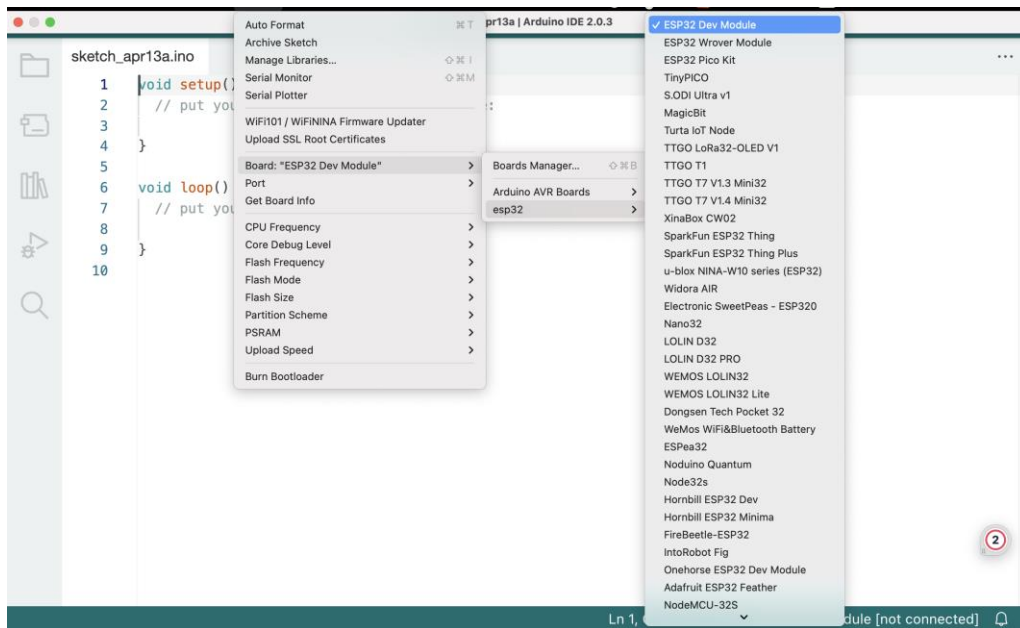


Figure 1: choosing a board in Arduino IDE

2. Create a new sketch - a sketch is your main program file. Here we write a set of instructions we want to execute on the microcontroller.

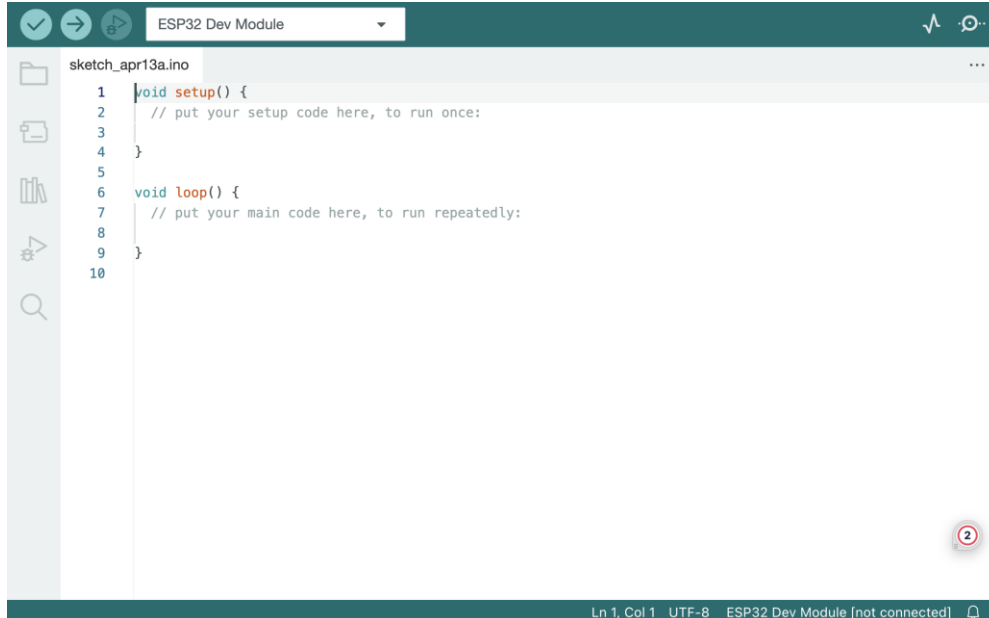



Figure 2: new sketch in Arduino IDE including the minimum two required functions

Void setup() - this function executes only once, Here we define things such as the mode of a pin (input or output), the baud rate of serial communication, or the initialization of a library.

Void loop() - this is where we write the code that we want to execute over and over again.

3. Compile your sketch - the code we write is not exactly how it looks like when uploaded to our Arduino: compiling code means that we check it for errors, and convert it into a binary file (1s and 0s). If something fails, you will get this in the error console.

Compiling the sketch is done using the  button.

4. Upload your sketch - once the compilation is successful, the code can be uploaded to your board. In this step, we connect the board to the computer physically and select the right serial port.

Tools -> port -> chose a port.

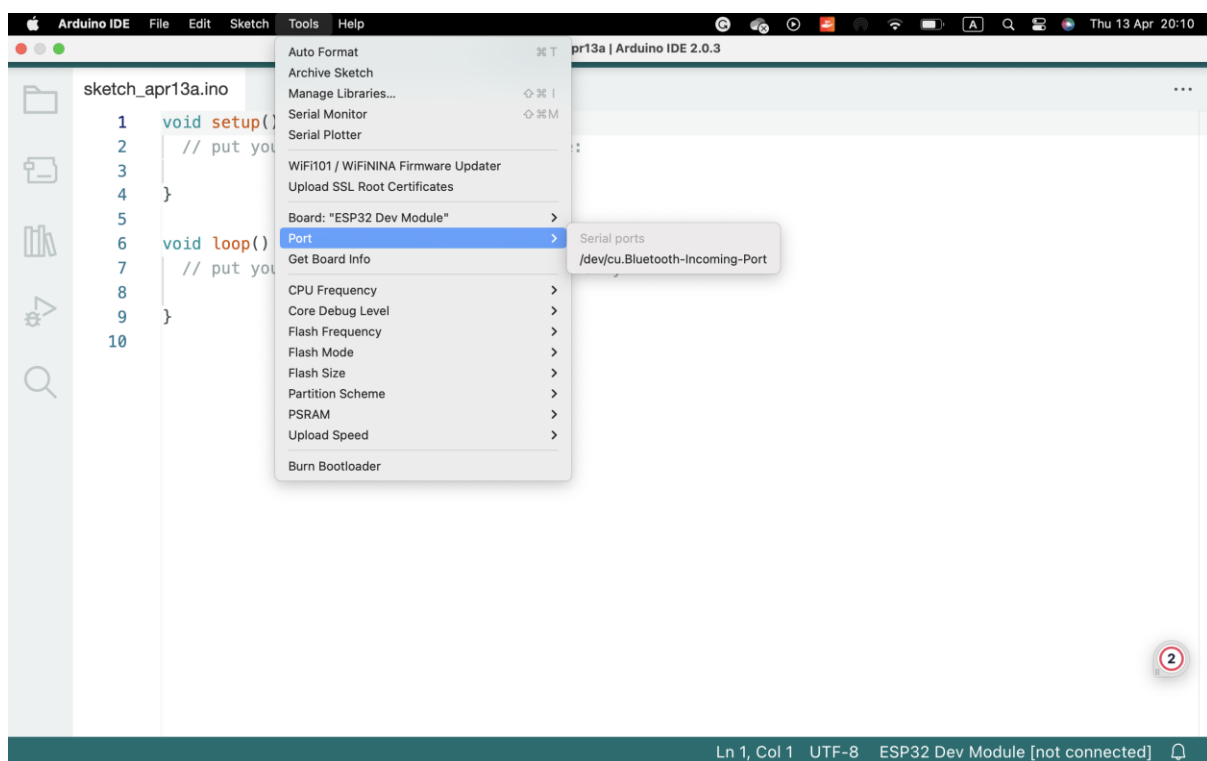



Figure 3: choosing a port for uploading in Arduino IDE

Uploading the code is done using the  button.

5. Serial Monitor (optional) - for most Arduino projects, it is important to know what's going on on your board. The Serial Monitor tool available in all IDEs allows for data to be sent from your board to your computer.

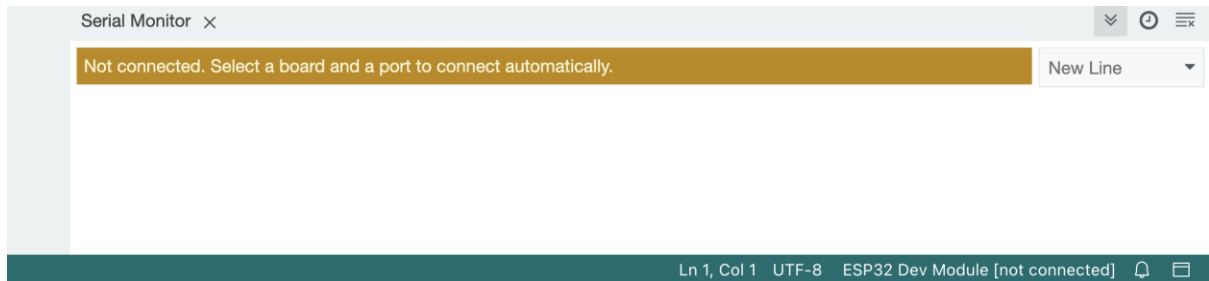


Figure 4: the serial monitor in Arduino IDE

Every version of the IDE has a library manager for installing Arduino software libraries. Thousands of libraries, both official and contributed libraries, are available for direct download.



Figure 5: library manager for downloading libraries in Arduino IDE

The libraries we have used in the project:

- ArduinoJson.
- AsyncTCP.
- ESPAsyncTCP.
- Painless_Mesh.
- TaskScheduler.

Reference: <https://docs.arduino.cc/learn/starting-guide/getting-started-arduino>

Getting started with arduino

GitHub

GitHub Inc. is a web-based hosting service for version control using Git. It was founded in 2008 and is now owned by Microsoft. GitHub provides a platform for developers to collaborate on software projects by storing and managing code repositories, tracking changes to the code over time, and facilitating collaboration among team members.

In addition to hosting code repositories, GitHub provides a range of tools and features that support software development, including bug tracking, project management, code review, and more. GitHub has become a popular platform for open-source software development, with millions of users around the world contributing to a vast array of projects.

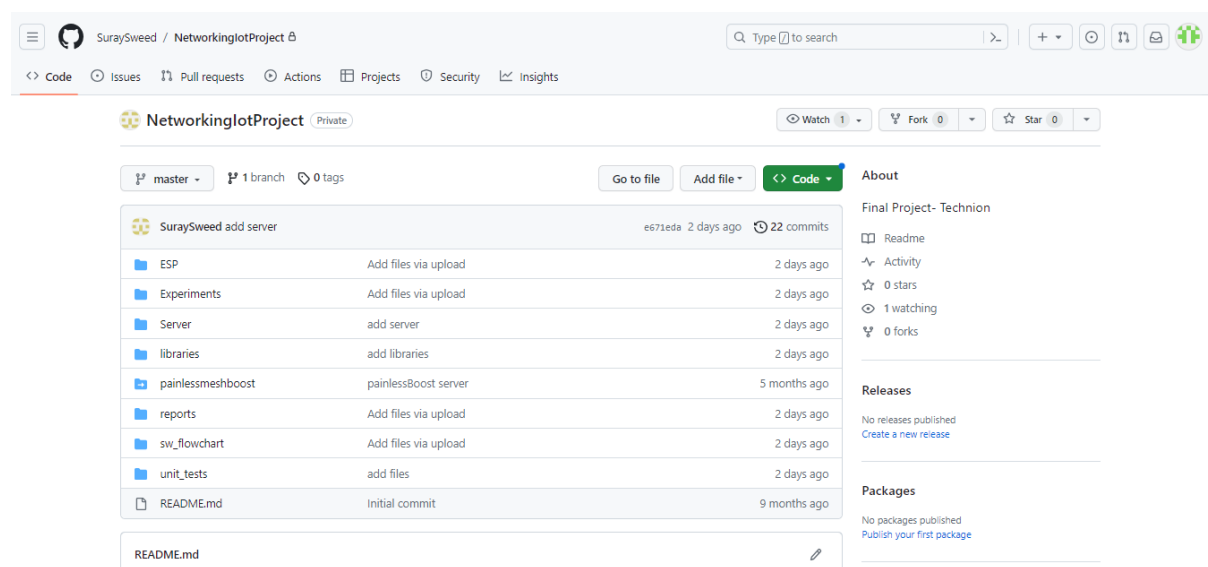


Figure 6: project's git

Chapter 2: Building the hardware circuit

2.1 Introduction- The hardware circuit

The software is tested on a simple circuit using the following components:

- Number of ESP32-CAM, one of them operates as a master and the rest as clients.
- Server running on a laptop.

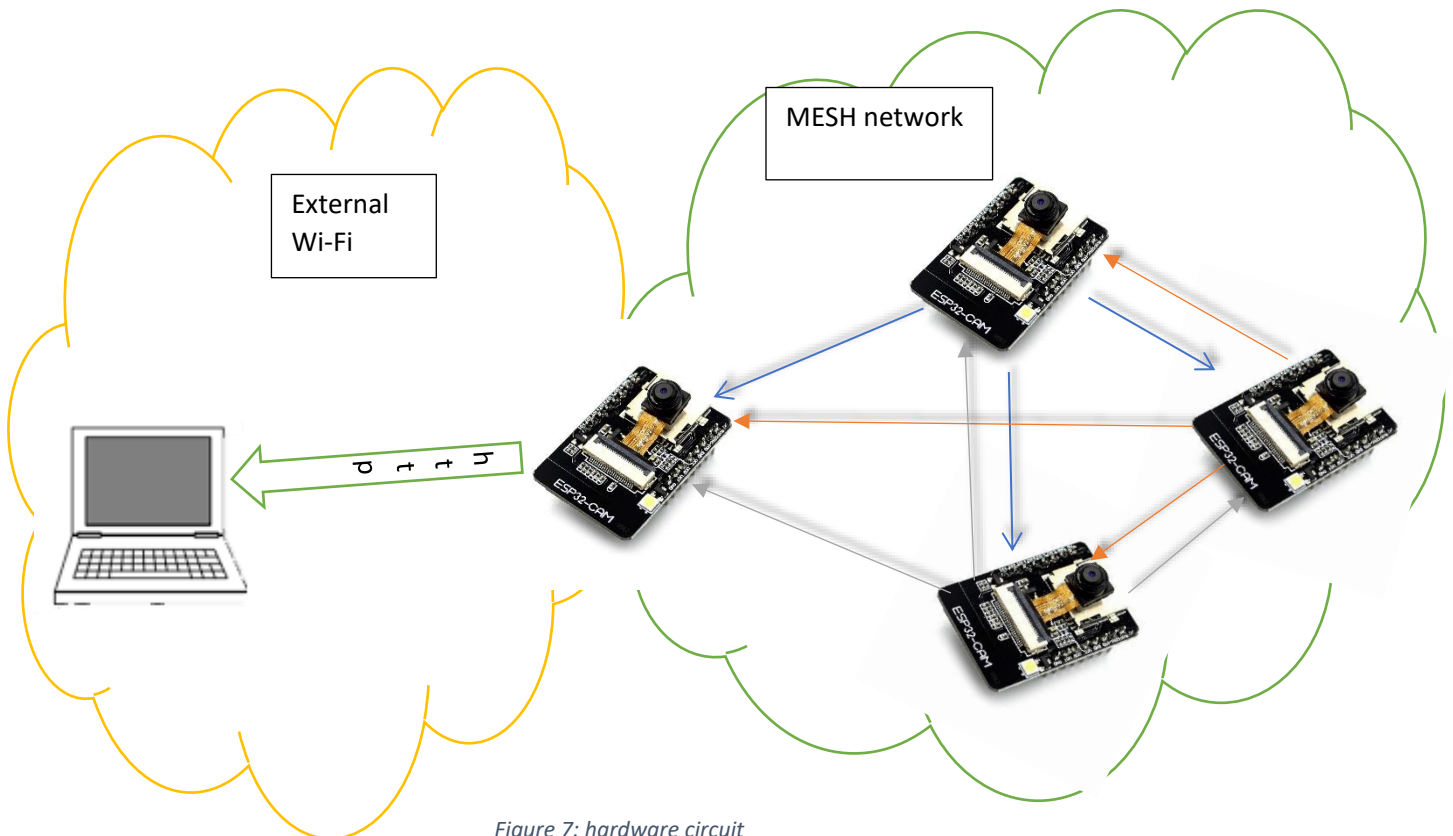


Figure 7: hardware circuit

2.2 The processor- ESP32-CAM

2.1.1 The processor ESP32

ESP32 is a powerful SoC microcontroller with integrated Wi-Fi, dual mode Bluetooth version 4.2 and a variety of peripherals. It is equipped with 4MB of flash memory. The ESP32 chip has been developed by Espressif Systems, which currently offers several ESP32 versions of the SoC in the form of ESP32 Developer Kit, the ESP32 Wrover Kit which also includes an SD card and 3.2 " LCD display, and finally the ESP32 Azure IoT kit with USB Bridge and other built-in sensors. The ESP32 device can be programmed using any Windows, Linux, or MacOS operating system. The easiest way to start writing code for the ESP32 platform is to use the Arduino platform. The main advantages of using ESP32 chip include wide deployment capabilities, support for Wi-Fi standards and protocols, Bluetooth communication, and low-cost solutions.

2.1.2 The processor ESP32-CAM

The ESP32-CAM is a popular development board that combines ESP32 microcontroller with a camera module, making it capable of capturing images and streaming video over a network. It offers a cost-effective and versatile solution for projects requiring camera functionality and wireless communication.

In addition, ESP32-CAM comes with an OV2640 camera module that features a 2MP (megapixel) image sensor. It supports various resolutions and formats, including JPEG, BMP, YUV, and RGB. The camera module connects to the ESP32 via the Serial Camera Control Bus (SCCB) interface.

ESP32-CAM is suitable for a wide range of applications, such as surveillance systems, video streaming, home automation robotics and IoT projects. Its compact size and low power consumption make it ideal for embedded and portable devices.



Figure 8: ESP32-CAM processor

Pin layout:

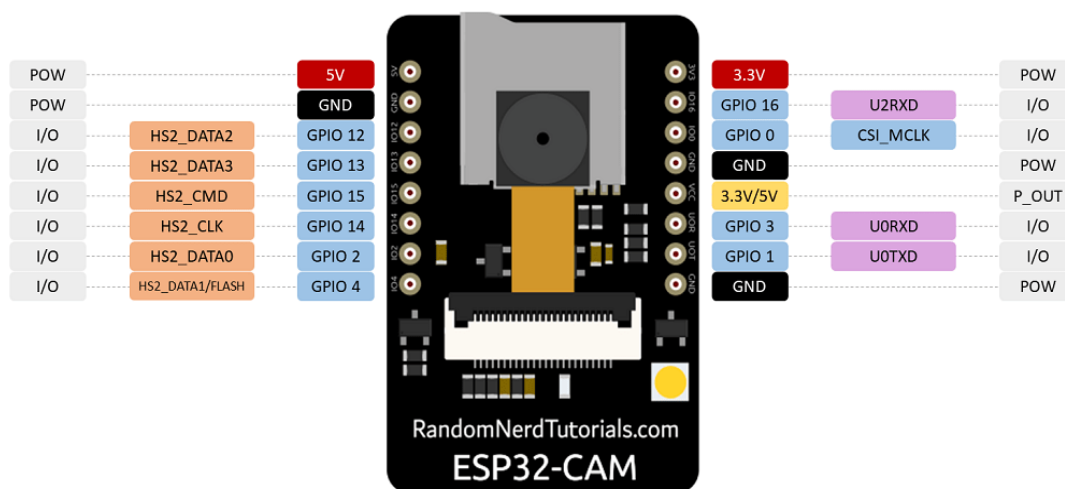


Figure 9: ESP32-CAM pins layout

There are three GND pins and two pins for power: either 3.3V or 5V.

GPIO 1 and GPIO 3 are the serial pins. You need these pins to upload code to your board. Additionally, GPIO 0 also plays an important role, since it determines whether the ESP32 is in flashing mode or not. When GPIO 0 is connected to GND, the ESP32 is in flashing mode.

The following pins are internally connected to the microSD card reader:

- GPIO 14: CLK
- GPIO 15: CMD
- GPIO 2: Data 0
- GPIO 4: Data 1 (also connected to the on-board LED)
- GPIO 12: Data 2
- GPIO 13: Data 3

Chapter 3: Wireless Mesh Network

3.1 Abstract:-

A wireless mesh network (WMN) is a type of mobile ad hoc network (MANET), which provides access to many users covering a wide area. MANET is formed dynamically by mobile devices with no need for infrastructure or prior network configuration. Like MANET, WMN has the abilities of self-organizing, self-discovering, self-healing, and self-configuration. However, WMN contains mesh routers (MRs), some of them have wired connection and acts like gateways to provide an internet connection to the mesh network.

3.2 Introduction-

Mesh networking is the local network topology in which nodes are directly and dynamically connected spread over a large physical area. The nodes communicate with one another to efficiently send the data from/to clients. Because of that, the whole network does not rely on one node only. As a result, the large number of nodes can be connected to the Internet without adding routers to the network. Mesh networking contain an auto-networking functionality which means that when the user sets up a mesh network, any node can scan the access point and connect easily.

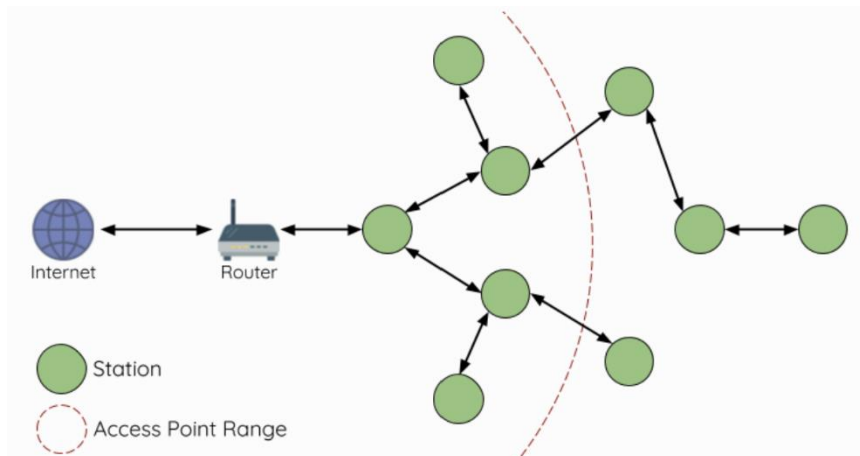


Figure 10- Mesh Network Architecture

The MANET was initialized by the defense advanced research projects agency around 30 years ago for military uses in most cases. Recently, WMN emerged as a multi-hop ad hoc network to provide Internet access for communities, enterprises, and cities.

In comparison to pure MANET, WMN has a hierarchical structure that includes two parts as shown in Figure 2:

- 1- Mesh backbone which contains the MRs. These MRs interconnect through single/multi-hop wireless links forming the backbone. Some of these MRs are wirely connected to the internet.
- 2- Mesh clients, which are mobile wireless devices such as cellphones and laptops. The mesh clients connect to any MR to receive internet services.

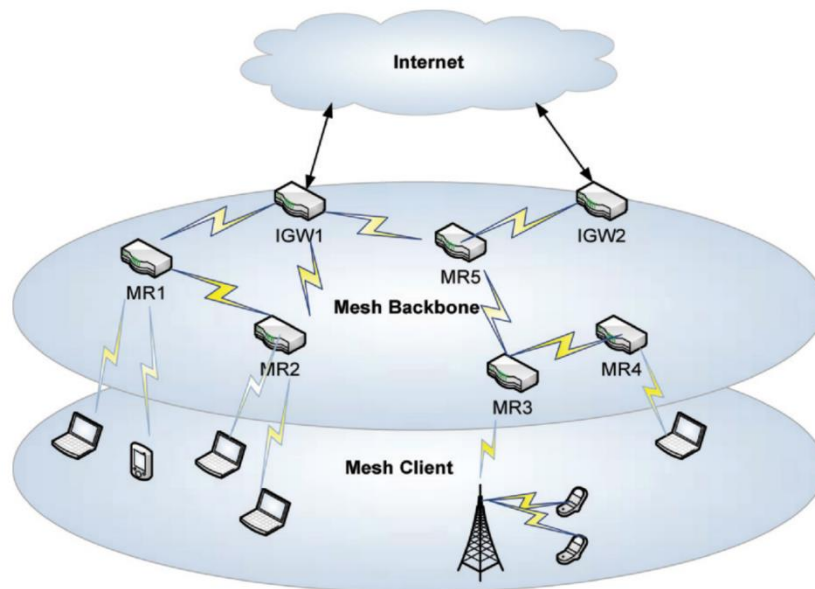


Figure 11: A typical architecture of WMN

Mesh backbones topology is relatively stable and supplies a cost-efficient way to access the internet compared to traditional wired and wireless networks such as WI-FI. This is because of the reduced dependence on infrastructure and multi-hop routing.

MANET's main characteristic (which is also available in WMN):

- **Self-forming, self-configuring, and self-healing:** the network is established spontaneously, and the multi-hops communications are managed without any centralized network authority.
- **Dynamic topology:** the topology of MANET changes permanently because of the device's mobility, new node's enrollment, and some nodes leave without prior notification.
- **Constrained resources:** the nodes are usually powered by a battery to receive and send packets to other nodes, so they work for a limited period.

From MANET to WMN:

While MANET faced problems to supply internet service to its nodes, WMN supplies internet services by MRs forming the mesh backbone, so mesh clients (mobile devices) can access the internet via these MRs. A MR not only handles the packets that arrived from its clients but also relays the traffic from other MRs, so a client can access the internet through a multi-hop manner. As a result, the installation cost of the whole system will be reduced.

Main improvements of WMN over MANET:

- Infrastructure support of MANET: as said before, MANET clients haven't internet access. On the contrary, MRs that are connected to the internet forming internet gateways (IGs) provide internet accessibility for mesh clients.
- Minimal mobility of MRs in a WMN: each MR is situated in a fixed position, like the roof of a building, and has minimum mobility, therefore the topology of the backbone is fixed unless a new enrolling of MR happened or any MR leaves. This is important for reducing the probability of link breakage and for the improvement of the network throughput.
- Rich energy of MR: in WMN each MR is connected to a rich power supply unlikely MANET which suffers from energy constraints.
- Multi-radio and multichannel: each MR in WMN can use multiple orthogonal channels so the network bandwidth is improved. This is abled because MRs can simultaneously transmit/receive packets in multiple radios without interference with each other.

Some of WMN's benefits (why WMN):

- **High-speed wireless system:** the wireless backbone of the WMN provides internet accessibility for mesh clients. IEEE 802.11g and 802.11n wireless protocols provide high-speed connectivity in the 2.4 GHz band, with data rates of up to 54 Mbps and between 100-200 Mbps respectively. These high rates are enabled by advanced physical layer technology such as modulation coding schemes and multiple input/output. The wireless backbone can support real-time commercial applications and bandwidth-consuming communication needs, and its capacity can be improved by deploying multiple orthogonal channels and infrastructure consisting of IGWs nodes. As the MRs number that performs as IGWs rise the network capacity is improved.
- **Promising coverage and connectivity:** in pure MANET, internet accessibility isn't available due to its infrastructure-less nature. In a wireless local area network (WLAN) each device is connected directly to the access point using centralized medium access. Therefore, WLAN coverage is a limited area, and the connectivity is only available in a single hop range. In wireless personal area network (WPAN), a high-speed data transmission is provided but it's for a short range because of limited transmission power and its operating frequency band. In WMN, each MR serves as both a host for its associated mobile clients and a router to forward packets on behalf of other MRs. As a result, WMN covers a wide area and supplies connectivity for many clients using multi-hop fashion. Additionally, mobile devices can quickly transition in the WMN from one accessing MR to another using an appropriate migrating scheme.
- **Flexible and cost-efficient:** WMN can be constructed and managed by one single internet service provider (ISP), or the core MRs can be controlled by one or several ISPs while other MRs can be added by users, which is called some managed WMN. Moreover, a WMN can be operated in an unmanaged way where each MR is managed independently. Regardless of the commercial operating model, an MR can be added to WMN or uninstalled from it using the self-configuration and self-forming capabilities. Internet accessibility is achieved using IGWs, which reduced the need for expensive links and hardware. For those reasons, WMN is a cost-efficient solution for mobile users.

Reference: book: "guide to wireless mesh network", chapter 1, pages 17-23

3.3 IEEE 802.11s Wireless Mesh Network Standard -

IEEE 802.11s standard defines a hybrid wireless mesh protocol (HWMP) that combines the flexibility of reactive on-demand route discovery and the efficiency of proactive routing. The reactive on-demand mode in HWMP is based on the Radiometric ad-hoc on-demand distance vector (RM-AODV) protocol, while the proactive mode is implemented by tree-based routing. This combination can achieve optimal and efficient routing selection.

Moreover, the standard determines the HWMP information elements which are path request (PREQ), path reply (PREP), path error (PERR), and root announcement (RANN). This information is used to propagate metric information among the mesh STAs and determines the cost of links in the network.

The metric cost of links is used in HWMP to determine which path to build. HWMP can support various radio metrics in path selection, such as throughput, quality of service (QoS), load balancing, power awareness, and more. The default metric is the airtime cost metric, which considers the PHY and MAC protocol overhead, frame payload, and the packet error rate to reflect the radio link condition.

The airtime cost of each link C_a is given by:

$$C_a = \left[O + \frac{B_t}{r} \right] \frac{1}{1 - e_f}$$

When:

- O and B_t are constants for each 802.11 modulation type.
- r is the data rate in Mb/s.
- e_f is the frame error rate in Mb/s for the test frame with size B_t .

Reference: mobile lightweight wireless system pages: 276-278 + 280

3.4 PainlessMesh Library- ESP Mesh Wi-Fi

3.4.1 Introduction-

PainlessMesh is a Wi-Fi mesh network library specifically written for the ESP32-cam platforms and is actively maintained by the community. The library is written in c++ language. It is called PainlessMesh for it is intended to be auto-configure and easy to setup. It is an ad-hoc network which requires neither routing plan nor central controller and all nodes are equal.

The PainlessMesh library is unique for it enables the resource limited ESP32-cam device to form a mesh network. Two or more modules, also called nodes, with the same SSID (service set identifier) will be automatically connected to form a mesh network. In the PainlessMesh network, each node can serve as both an access point for other nodes to connect to, and also a station connecting to another node. Every node is aware of the whole network topology which is updated periodically every 3 seconds. Every node tells its neighbors about other nodes it is directly or indirectly connected to. A node's station which is not connected to any access point will actively look for an access point that has the strongest signal but is not listed yet in its network topology. This mechanism prevents the formation of cyclic paths. So there will be only a single path between a pair of nodes.

The PainlessMesh works on layer 3 of the OSI Model- Network layer but not based on IP addressing. Instead, each node is distinguished by a unique 32-bit number taken from chip MAC address. Communication in the mesh is done via JSON formatted messaging. Although it is less efficient than low level binary messaging, it is easier for users to understand and to incorporate in JavaScript or web applications. There are two types of user message, i.e., single and broadcast message. Single message is sent from a node to another specific node, whereas broadcast message is addressed to all other nodes connected in the network. All nodes are time synchronized with a precision of less than 10 Ms. This is useful for the nodes to run synchronous tasks. Mesh network offers its usefulness by allowing a collection of ESP32-cam nodes to cover larger area through multi-hop messaging. Any nodes within reachable range will be automatically forming a mesh network. Therefore, the mesh network also leverages the network reliability by automatically joining to a neighboring network in the event of disconnection.

3.4.2 Data Transmission

3.4.7.1 Esp Wi-Fi Mesh Packet

Mesh network data transmissions use ESP Wi-Fi Mesh packets. These packets are entirely contained within the frame body of a Wi-Fi data frame. During a multi-hop data transmission within the mesh network, each wireless hop involves the packet being carried by a different Wi-Fi data frame.

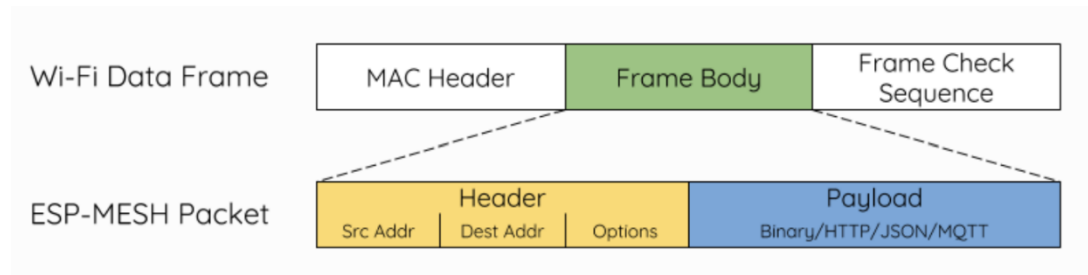


Figure 16- ESP-WIFI-MESH packet and its relation with a Wi-Fi data frame

- The header of this packet contains the MAC addresses of the source and destination nodes. The options field contains information pertaining to the special types of these packets such as a group transmission or a packet originating from the external IP network.
- The payload of this packet contains the actual application data. This data can be raw binary data or encoded under an application layer protocol such as HTTP, MQTT and JSON.

3.4.7.4 Bi-Directional Data Stream

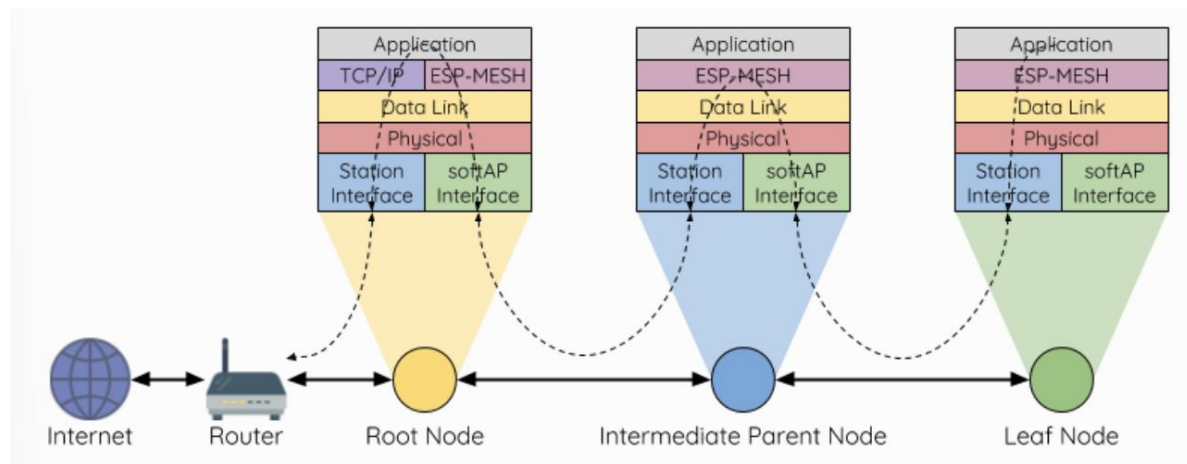


Figure 17: ESP-WIFI-MESH Bi-Directional Data Stream.

A TCP/IP layer is only required on the root node when it transmits a packet to an external IP network.

3.4.8 Esp Wi-Fi Mesh programming

3.4.8.1 Software Stack

The ESP-WIFI-MESH software stack is built atop the Wi-Fi Driver and may use the LwIP Stack in some instance (i.e., the root node). The following diagram illustrates the ESP-WIFI-MESH software stack.

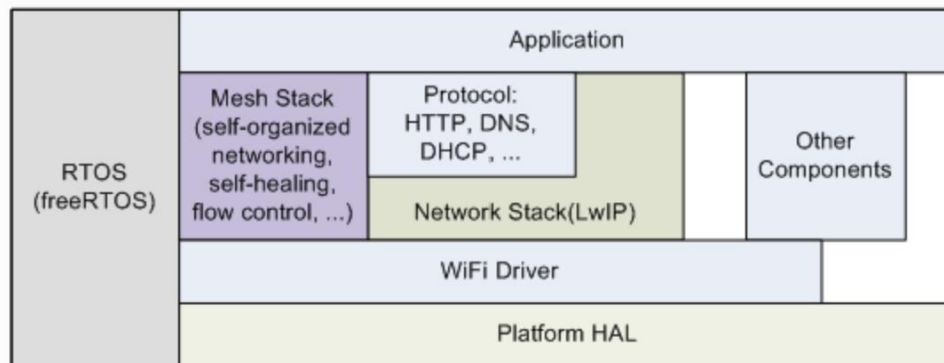


Figure 18: ESP-WIFI-MESH Software Stack

3.4.8.3 LwIP & ESP-WIFI-MESH

LwIP (Lightweight IP) is an open-source TCP/IP networking stack that is widely used in embedded systems and microcontrollers. It provides a lightweight implementation of the IP protocols, including IP, TCP, UDP, and ICMP. LwIP is designed to be memory-efficient and requires relatively low processing power, making it suitable for resource-constrained devices.

The LwIP stack enables network communication in ESP32 devices by handling IP-based protocols and allowing applications to establish connections, send and receive data, and handle network-related tasks. It provides APIs for socket programming and various networking functions, making it easier for developers to create networked applications.

ESP-WIFI-MESH requires a root node to be connected with a router. Therefore, in the event that a node becomes the root, the corresponding handler must start the DHCP client service and immediately obtain an IP address. Doing so will allow other nodes to begin transmitting/receiving packets to/from the external IP network. However, this step is unnecessary if static IP settings are used.

3.5 Json Format

JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is widely used for data serialization and communication between web services, applications and IoT devices.

JSON data is represented as key-value pairs and organized into a structured format. There are some basic elements in JSON such as, Objects, Arrays, Strings, Numbers, Booleans, Null.

3.6 TaskScheduler Library

The TaskScheduler library for Arduino allows to schedule and manage tasks to be executed at specific intervals or under certain conditions. It provides a simple interface for creating tasks and defining their execution rules.

To use the TaskScheduler library:

- Include the library.
- Create tasks and define their execution rules: Use TaskScheduler library's functions and macros to create tasks and specify when and how they should be executed.

Example that demonstrates the usage of the TaskScheduler library:

```
#include <TaskScheduler.h>

TaskScheduler scheduler;

// Task 1: Blink LED1 every 1 second
void task1() {
    digitalWrite(LED1_PIN, !digitalRead(LED1_PIN));
}

// Task 2: Blink LED2 every 500 milliseconds
void task2() {
    digitalWrite(LED2_PIN, !digitalRead(LED2_PIN));
}

void setup() {
    pinMode(LED1_PIN, OUTPUT);
    pinMode(LED2_PIN, OUTPUT);

    scheduler.init();

    // Add Task 1 to the scheduler with an interval of 1000 milliseconds
    scheduler.addTask(task1, 1000);

    // Add Task 2 to the scheduler with an interval of 500 milliseconds
    scheduler.addTask(task2, 500);

    // Enable all tasks in the scheduler
    scheduler.enableAll();
}

void loop() {
    // Run the scheduler to execute enabled tasks
    scheduler.execute();
}
```

In this example, we include the TaskScheduler library and create an instance of the TaskScheduler object named scheduler. We define two tasks- task1 and task2 which toggle the states of led1_pin and led2_pin respectively. Then we initialize the scheduler and add the tasks with their respective intervals. Finally, in the loop() function we run the scheduler to execute the enabled tasks.

Chapter 4: Warning Test Implementation

This chapter provides a guide to establishing a warning test infrastructure consists ESP32-CAM processors. The configuration involves three ESP32-CAM processors operating as clients, one ESP32-CAM processor functioning as the master, and a server running on a laptop. The server will be configured to listen on an IP address and port, displaying the alert messages arrived from the client through the master.

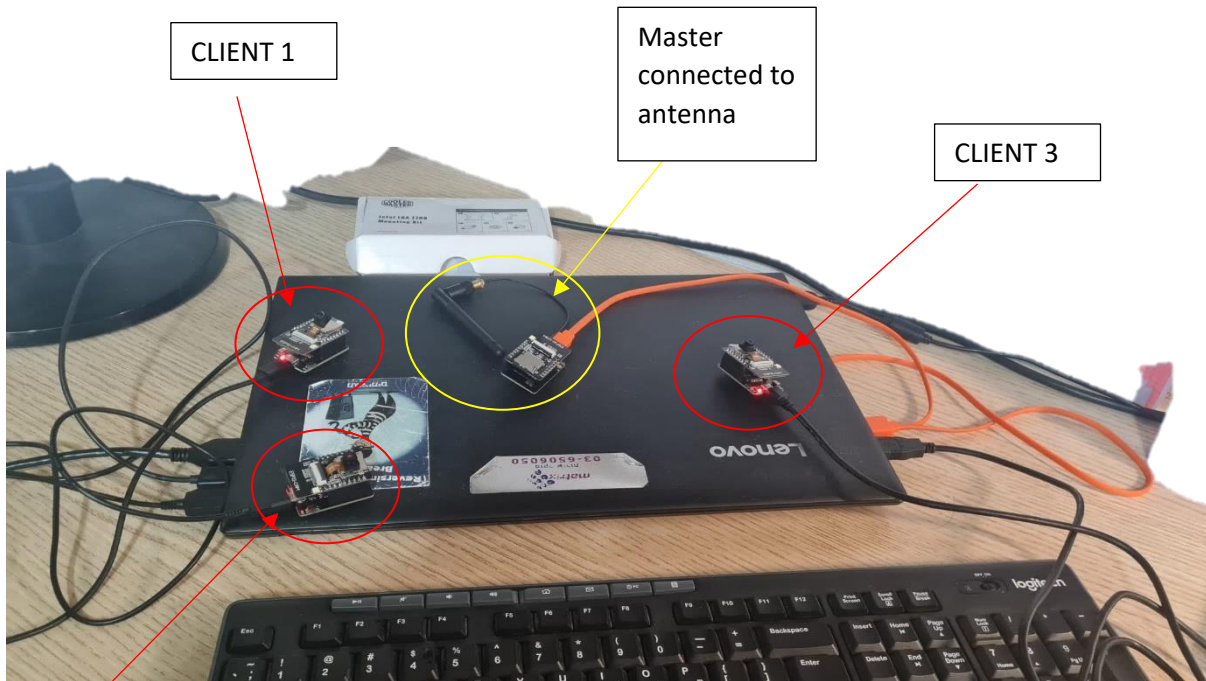
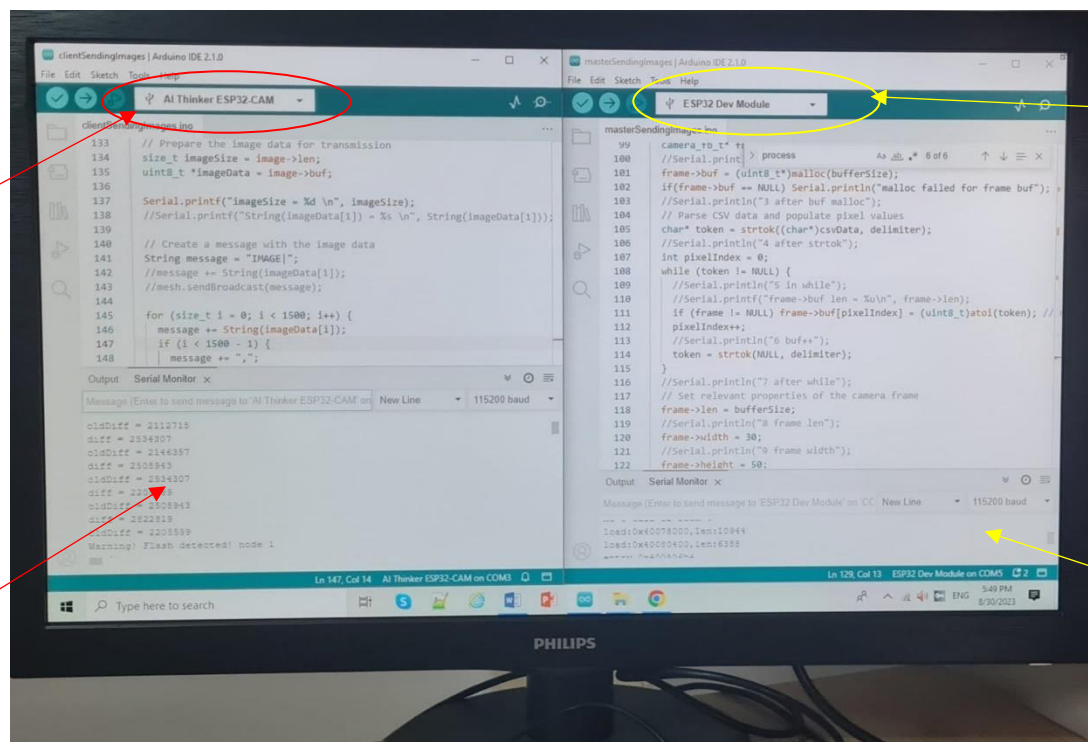


Figure 37: setup infrastructure

CLIENT 2

- We used micro-USB cables to connect ESP processors to the laptop for uploading relevant codes on these processors.



Board for master

Master's output

Figure 38: client & master screen

Server listening on an IP
address and port

The
comma
nd to
run the
server
on CMD
window

```
PS C:\Users\MyAsus\OneDrive\Desktop\NetworkingIotProject\Server> node server.js
Server listening on 192.168.93.210:3000
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3412893357
Message Data: Warning! Flash detected! node 2
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631658285
Message Data: Warning! Flash detected! node 3
Received data from node: 3631658285
Message Data: Warning! Flash detected! node 3
Received data from node: 3631658285
Message Data: Warning! Flash detected! node 3
Received data from node: 3412893357
Message Data: Warning! Flash detected! node 2
Received data from node: 3412893357
Message Data: Warning! Flash detected! node 2
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
```

Figure 39: server's screen

To run the server:

1. Set Up Node.js Environment: Ensure you have Node.js and npm (Node Package Manager) installed on your computer. If you don't have them installed, download and install Node.js from the official website: <https://nodejs.org/>
2. Create a New Folder: Create a new folder on your computer where you want to store the server code.
3. Create a Package.json File: In the new folder, create a file named **package.json** and add the following content:


```
- {
-   "name": "mesh_server",
-   "version": "1.0.0",
-   "main": "server.js",
-   "dependencies": {
-     "express": "^4.17.1",
-     "body-parser": "^1.19.0"
-   }
- }
```
4. Install Dependencies: Open a terminal or command prompt, navigate to the folder where you created **package.json**, and run the following command to install the required dependencies (Express and body-parser): `npm install`
5. Create the Server File: Create a new file named **server.js** in the same folder and paste the provided server code into it.
6. Run the Server: In the terminal or command prompt, while still in the same folder, run the following command to start the server: `node server.js`

Flow chart:

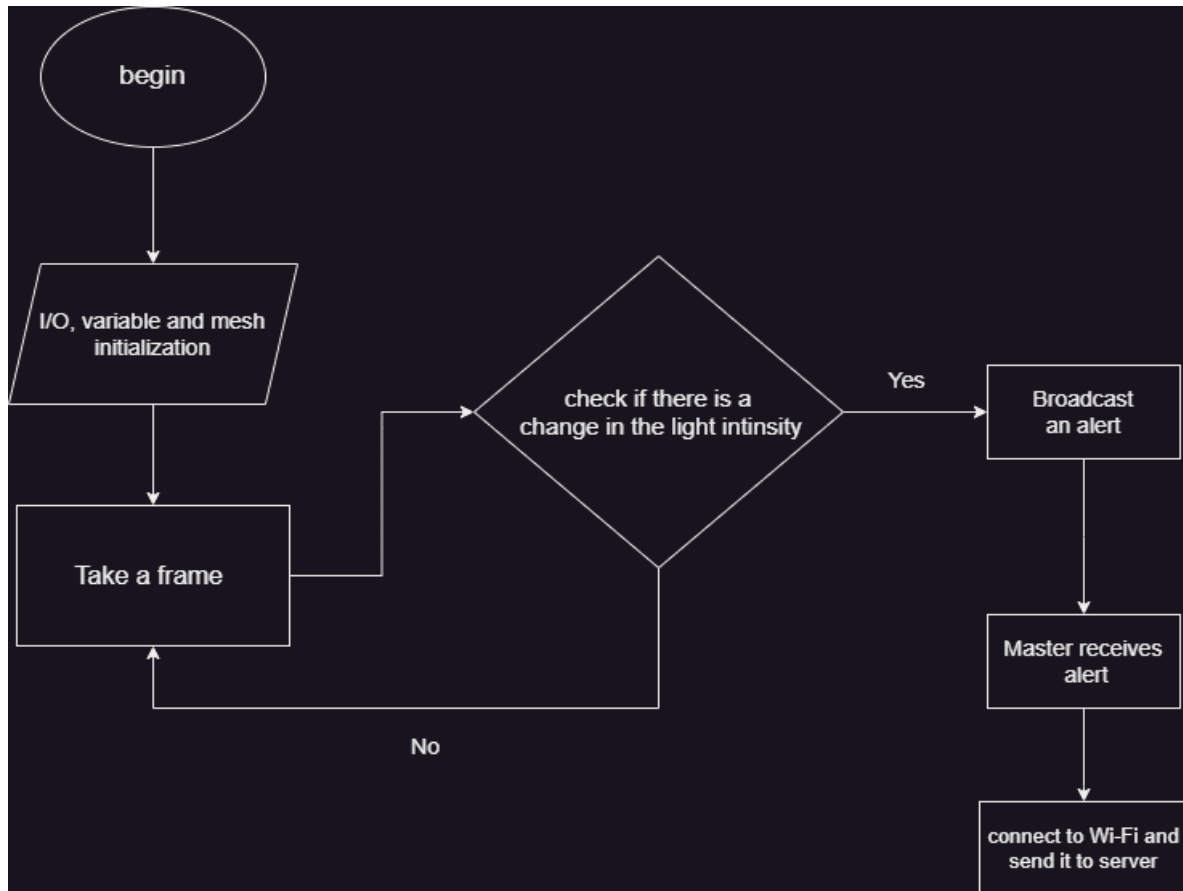


Figure 40: Software Flow Chart

To check if there is a change in light intensity we used Light intensity changes detection algorithm (Client Side):

```

int diff = 0;

for (int i = 0; i < fb->len; i += 2) {
    diff += abs((int)fb->buf[i] - (int)fb->buf[i + 2]);
}

Serial.printf("diff = %d\n", diff);
Serial.printf("oldDiff = %d\n", oldDiff);
// Add your logic here to trigger a warning message based on the flash
detection
bool triggerWarning = (diff - oldDiff > FLASH_DIFF); // Adjust
FLASH_THRESHOLD as needed

if (triggerWarning) {
    String message = "Warning! Flash detected! node 1 ";
    sendMessage(message); // Send the warning message to all nodes in the
network
    Serial.println(message);
}
  
```

```

    oldDiff = diff;
    esp_camera_fb_return(fb); // Release the camera frame buffer
  }

void sendMessage(const String& message) {
  Serial.println("sending warning to master");
  mesh.sendBroadcast(message);
}

```

1. `int diff = 0;` Initializes a variable `diff` to keep track of the cumulative difference between pixel values in the frame buffer.
2. The for loop iterates through the pixel values in the frame buffer, incrementing `i` by 2 in each iteration. This is because each pixel value consists of two consecutive bytes (e.g., Red and Green components). `fb->len - 2` is used as the loop condition to prevent accessing memory beyond the buffer's length.
3. Inside the loop, `abs((int)fb->buf[i] - (int)fb->buf[i + 2])` calculates the absolute difference between the pixel value at index `i` and the pixel value at index `i + 2`. This difference represents the change in brightness between adjacent pixels.
4. `diff += ...;` The calculated absolute difference is added to the `diff` variable, accumulating the overall brightness change across the frame.
5. After the loop, the code prints the `diff` and `oldDiff` values to the serial console for debugging purposes.
6. The next line `bool triggerWarning = (diff - oldDiff > FLASH_DIFF);` calculates the difference between the current `diff` value and the previous `oldDiff` value. If this difference is greater than a predefined `FLASH_DIFF` threshold, it's interpreted as a significant change in brightness, potentially indicating a flash of light.

Note: The accuracy of flash detection depends on the choice of `FLASH_DIFF` and the nature of the input image frames. Adjustments may be necessary for optimal results.

The code of sending alerts to server (master Side) using http protocol:

```

void sendMsgToServer(uint32_t nodeId, const String& msg) {

  WiFi.begin(ssid, password);
  do {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  } while (WiFi.status() != WL_CONNECTED) ;

  if (WiFi.status() == WL_CONNECTED){
    Serial.println("Connected to WiFi");
    HTTPClient http;
    http.begin(serverUrl);

    // Create a JSON payload with node ID and message data
    DynamicJsonDocument data(128);
    data["node_id"] = String(nodeId);
  }
}

```

```

    data["msg_data"] = msg;

    String dataStr;
    serializeJson(data, dataStr);

    http.addHeader("Content-Type", "application/json");
    int httpResponseCode = http.POST(dataStr);

    if (httpResponseCode > 0) {
      Serial.print("HTTP response code: ");
      Serial.println(httpResponseCode);
      hello_sended = true;
    } else {
      Serial.println("Error sending message to server");
    }

    http.end();
  }
}

```

Server code using java script:

```

const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());

const ipAddress = '192.168.93.210'; // Replace with the desired IP address
const port = 3000; // Specify the port number

// Route to handle the POST request from the master node
app.post('/receive-data', (req, res) => {
  const receivedData = req.body;
  console.log('Received data from node:', receivedData.node_id);
  console.log('Message Data:', receivedData.msg_data);

  // You can perform any further processing or store the data in a database
  here

  res.sendStatus(200); // Send a success response
});

// Start the server
app.listen(port, ipAddress, () => {
  console.log(`Server listening on ${ipAddress}:${port}`);
});

```


Chapter 5: Running and Results

4.1 Assessing Image Transmission Quality in ESP Mesh Networks

Purpose of the test: The primary objective of this test was to assess the efficacy of image transmission within a mesh network employing ESP cards. Our focus was on evaluating the network's proficiency in transmitting images while also uncovering the constraints and exploring potential remedies for transmitting larger images effectively. This endeavor aimed to provide insights into the network's image-handling capabilities and pave the way for enhancing its performance in image transmission scenarios.

Test Environment: This experimentation took place within the well-established Arduino development environment, which was executed on a Windows operating system. The selection of this environment was deliberate, as it allowed us to utilize Arduino-compatible ESP cards effectively to construct and operate our mesh network. The importance of this choice lies in its suitability for programming and rigorous testing of these ESP-based devices, ensuring accurate and reliable results throughout the evaluation process.

Methodology: In the course of our investigation, we employed a well-established networking methodology: breaking the image into smaller, more manageable chunks. These image fragments were transmitted individually across the mesh network. This approach aligns with best practices in networking, where data is often divided into discrete packets to facilitate efficient transmission. The methodology was chosen to enable a thorough assessment of the network's capacity to transmit and accurately reassemble data.

```
diff = 2822819
oldDiff = 2205599
Warning! Flash detected! node 1
sending image data
imageSize = 71365
image is IMAGE|255,216,255,224,0,16,74,70,73,70,0,1,1,1,0,0,0,0,0,255,219,0,1
diff = 2347001
oldDiff = 2822819
```

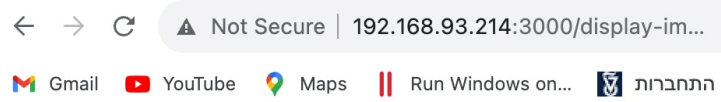
The client detects a change in light intensity, then sends an image to the master. The image is sent as pixels' values (only first 1500 pixel is sent due to resource limitations of ESP processor).

Figure 24: client

```
New connection, nodeId = 3631657293
Received message from 3631657293: IMAGE|255,216,255,224,0,16,74,70,73,70,0,1,1,
process image at master
1 at createCameraFrameFromCSV
send image to server
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connected to WiFi
image len = 1500
message to server is 255,216,255,224,0,16,74,70,73,70,0,1,1,0,0,0,0,0,255,2
Image sent to server. HTTP response code: 200
```

The master receives an image of 1500 pixels from the client, process it, connects to Wi-Fi and send it to the server.

Figure 25: master



The server displays the received images on web. Here we see the web before receiving any image.

No image available

The server received an image from the master.

```
aliswade@Alis-MacBook-Pro serverNode % node imageServerNode.js
Server listening on 192.168.93.214:3000
Image received from master.
```

Here we see the received image uploaded to the web by the server.

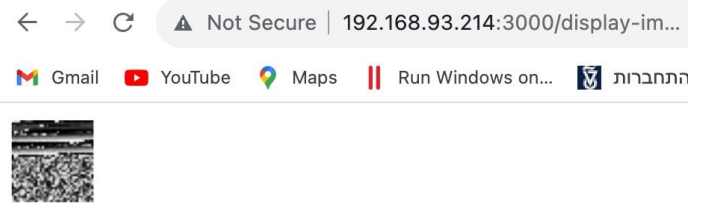


Figure 26: server & web

Result: It's obvious that the shown image on web is uninformative, and this is because the client can send only 1500 pixels. So we concluded that the client should split the image to chunks of 1500 pixels, and then send them, before doing that we test if the master is capable to receive all chunks...

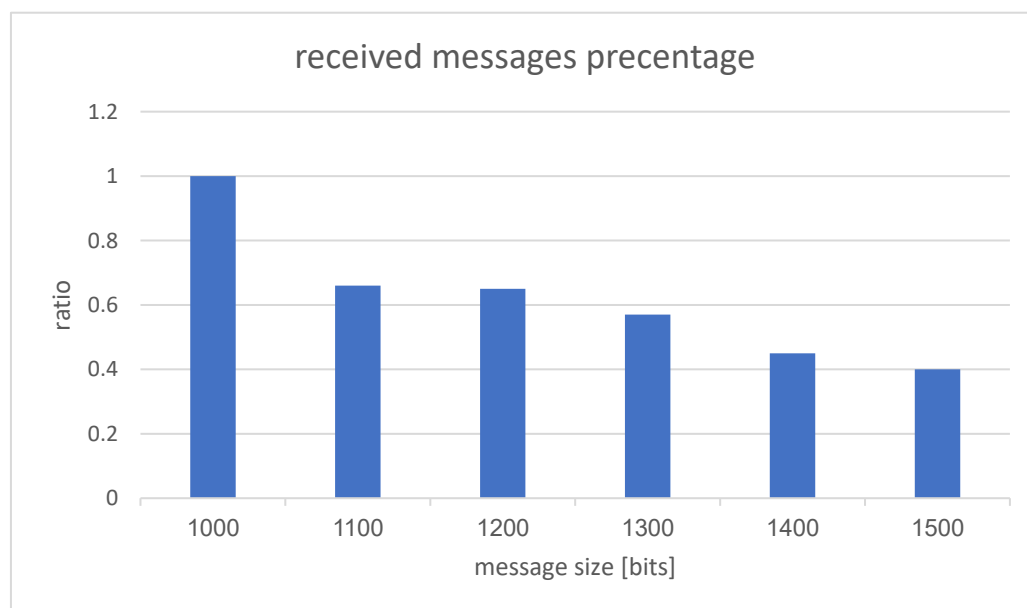
4.2 Packet Size Variation Test

Purpose of the Test: The primary objective of this test phase was to investigate the influence of packet size variations on the server's ability to receive data packets reliably. We systematically adjusted the packet size, ranging from 1000 bytes to 1500 bytes, with 100-byte increments in each iteration. The overarching goal was to gauge how altering packet sizes impacted the percentage of successfully received packets on the server. This analysis was pivotal in understanding the network's behavior under different packet size scenarios.

Test Environment: The experiments were conducted within the established Arduino development environment, operating on a Windows system. This choice of environment was strategic, as it provided an ideal platform for programming and rigorously testing the ESP-based mesh network components. The environment ensured that the results obtained were consistent and reliable throughout the evaluation process.

Methodology: To achieve our objective, we initiated a systematic process where we progressively adjusted the size of data packets during transmission. We began with packet sizes of 1000 bytes and subsequently increased them in increments of 100 bytes, culminating at 1500 bytes. This alteration was executed for 100 consecutive transmission attempts per packet size. The core methodology aimed to assess how variations in packet size affected the server's ability to receive and process these packets accurately. This approach aligned with best practices in network testing, facilitating a comprehensive understanding of the network's behavior in response to variable packet sizes.

The result of this test is as shown below:



Result: Unfortunately, we see that the master did not get all chunks sent, so we decided to give up on sending images📷. Instead, we decided to send alert messages from the client to the server.

4.3 Alerts Test

Purpose of the Test: The goal was to detect a change in the light intensity of the camera and send an alert to the server through the mesh network with expanding the network.

Test Environment: The experiments were conducted within the established Arduino development environment, operating on a Windows system. This choice of environment was strategic, as it provided an ideal platform for programming and rigorously testing the ESP-based mesh network components. The environment ensured that the results obtained were consistent and reliable throughout the evaluation process.

Methodology: In the initial test, denoted as Test #1, we established a rudimentary network structure comprising a single master node and a lone client node, both interconnected with a central server node. This foundational setup allowed us to gauge the library's performance in a simplified environment, laying the essential groundwork for our subsequent evaluations. Here, we aimed to understand how the library operated under basic conditions and its ability to facilitate communication between minimal network components.

4.3.1: Fundamental Structure, 1 master, 1 client with server

```
Warning! Flash detected! node 1
diff = 10068993
oldDiff = 1438043
sending warning to master
```

Figure 27: client#1

The client detects a change in light intensity, then sends an alert to the master

```
MESH_STATUS: Changed connections in neighbour 3631657293
MESH_STATUS: New connection 3631657293
New connection, nodeId = 3631657293
Received message from 3631657293: Warning! Flash detected! node 1
sending warning to server
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connected to WiFi
HTTP response code: 200
```

Figure 28: master

The master receives a message from client#1, connects to Wi-Fi and send this message to

```
aliswade@Alis-MacBook-Pro serverNode % node server.js
Server listening on 192.168.93.214:3000
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
```

Server Output

Figure 29: Server

4.3.2 Expanding Network, 1 master, 2 clients with server

Building upon the insights gained from Test #1, we transitioned to Test #2, where we expanded our network by introducing an additional client node while retaining the master and server nodes. This expansion provided a critical opportunity to examine the library's adaptability as the interaction levels within the network increased.

```
sending warning to master
Warning! Flash detected! node 1
diff = 10342083
oldDiff = 10112515
Received message from 3412893357: Warning! Flash detected! node 2
```

Figure 30: client #1

```
Received message from 3631657293: Warning! Flash detected! node 1
diff = 6688173
oldDiff = 6802221
diff = 2313911
oldDiff = 6688173
diff = 3348439
oldDiff = 2313911
sending warning to master
Warning! Flash detected! node 2
```

Figure 31: client #2

client#1 receives an alert from client#2 (since client#2 sends alerts by broadcast), client#1 and client#2 send to the masters an alerts

```
Received message from 3631657293: Warning! Flash detected! node 1
sending warning to server
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connected to WiFi
HTTP response code: 200
Received message from 3412893357: Warning! Flash detected! node 2
sending warning to server
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connected to WiFi
HTTP response code: 200
```

Figure 32: master

The master receives alerts from client#1 and client#2, connects to Wi-Fi and send the alerts to the Server.

```
aliswade@Alis-MacBook-Pro serverNode % node server.js
Server listening on 192.168.93.214:3000
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3412893357
Message Data: Warning! Flash detected! node 2
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3412893357
Message Data: Warning! Flash detected! node 2
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3412893357
Message Data: Warning! Flash detected! node 2
Received data from node: 3631657293
```

Figure 33: Server

Server
Output

4.3.3 Intricate Network Configuration, 1 master, 3 clients with server

The zenith of our exploration, Test #3, entailed orchestrating an intricate symphony of nodes within the network. This advanced configuration featured one master node, three client nodes, and a central server node. By configuring this intricate network structure, we aimed to assess the library's resilience in navigating the complexities inherent in a multi-node environment.

```

Received message from 3412893357: Warning! Flash detected! node 2
diff = 2783893
oldDiff = 3520889
diff = 3873477
oldDiff = 2783893
sending warning to master
Warning! Flash detected! node 3
diff = 3707225
oldDiff = 3873477

```

Figure 34: client #3

```

Received message from 3631657293: Warning! Flash detected! node 1
sending warning to server
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connected to WiFi
HTTP response code: 200
Received message from 3631658285: Warning! Flash detected! node 3
sending warning to server
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connected to WiFi
HTTP response code: 200
Received message from 3412893357: Warning! Flash detected! node 2
sending warning to server
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connecting to WiFi...
Connected to WiFi
HTTP response code: 200

```

Figure 35: master

```

PS C:\Users\MyAsus\OneDrive\Desktop\NetworkingIoTProject\Server> node server.js
Server listening on 192.168.93.210:3000
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3412893357
Message Data: Warning! Flash detected! node 2
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631658285
Message Data: Warning! Flash detected! node 3
Received data from node: 3631658285
Message Data: Warning! Flash detected! node 3
Received data from node: 3631658285
Message Data: Warning! Flash detected! node 3
Received data from node: 3412893357
Message Data: Warning! Flash detected! node 2
Received data from node: 3412893357
Message Data: Warning! Flash detected! node 2
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1
Received data from node: 3631657293
Message Data: Warning! Flash detected! node 1

```

Figure 36: Server

Result: Our system successfully achieved the goal of detecting changes in light intensity using camera sensors and transmitting timely alerts through the mesh network to the central server. This accomplishment offered numerous advantages, including:

1. **Real-time Alerting:** Changes in light intensity were promptly detected and reported, allowing for immediate responses.
2. **Redundancy:** The mesh network architecture ensured that alert messages could reach the server through multiple paths, reducing the risk of message loss.
3. **Scalability:** The system's modular design allowed for easy expansion by adding more client nodes or enhancing the server's capabilities as needed.
4. **Enhanced Situational Awareness:** The system provided valuable insights into light-related events, aiding decision-making and security in various applications.

Chapter 6: Conclusion

In the culmination of our project, it becomes evident that utilizing ESP components for real-time fire detection and image transmission presents substantial challenges. The practicality of sending images in real time is hindered by the inherent limitations of the ESP hardware. With images composed of 120,000 pixels, the need to segment them into smaller 2,500-pixel packets results in a sequence of 48 packets per image. However, our project's exploration, as discussed in Chapter 4.2, reveals that the ESP components' capabilities pose a barrier to executing this packet sequence smoothly.

In light of these findings, a clear recommendation emerges: considering the current constraints, deploying ESP components for real-time fire detection and image transmission is not a viable solution. The project underscores the importance of aligning application demands with hardware capabilities and the need for pragmatic decisions when integrating advanced technologies into practical applications. This project contributes not only insights into ESP components but also a broader understanding of the intricate interplay between technology aspirations and hardware realities.

Chapter 7: Suggestions to improvement

- **Enhancing Fire Detection Algorithm:** The algorithm we implemented only detects a change in the light intensity of the camera so that it may warn of irrelevant things, it is possible to improve the algorithm that will analyze an image and detect fires.
- Send images in real time with high resolution and informative.

References

- [1] Yoppy. (2019). Performance Evaluation of ESP8266 Mesh Networks.
<https://iopscience.iop.org/article/10.1088/1742-6596/1230/1/012023/meta>
- [2] Luís Santos. (2022). Performance Assessment of ESP8266 Wireless Mesh Networks.
<https://www.mdpi.com/2078-2489/13/5/210>
- [3] Andrew Froehlich. (2022). wireless ad hoc network (WANET).
<https://www.techtarget.com/searchmobilecomputing/definition/ad-hoc-network>
- [4] What Is an Ad Hoc Network?
<https://www.comptia.org/content/guides/what-is-an-ad-hoc-network>
- [5] ESP32-CAM: Set Access Point (AP) for Web Server (Arduino IDE)
<https://randomnerdtutorials.com/esp32-cam-access-point-ap-web-server/>
- [6] How to Configure an ESP Mesh Network using Arduino.
<https://circuitdigest.com/microcontroller-projects/how-to-configure-an-esp-mesh-network-using-arduino-ide-to-communicate-between-esp32-esp8266-and-nodemcu>
- [7] ESP-MESH Getting Started using painlessMesh Library and ESP32/ESP8266.
<https://microcontrollerslab.com/esp-mesh-esp32-esp8266-painlessmesh-tutorial/>
- [8] ESP-WIFI-MESH.
<https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/esp-wifi-mesh.html>
- [9] ESP-WIFI-MESH Programming Guide.
<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp-wifi-mesh.html>
- [10] ESP32-CAM, Camera Module Based On ESP32, OV2640 Camera and ESP32-CAM-MB.
<https://www.waveshare.com/esp32-cam.htm>
- [11] Sudip Misra. Subhas Chandra misra- Isaac Woungang. Guide to Wireless Mesh Networks.
- [12] ESP32-MeshKit Guide.
https://github.com/espressif/esp-mdf/blob/master/examples/development_kit/README.md
- [13] Steven Conner. Intel Crop. (2006). IEEE 802.11s Tutorial Overview of the Amendment for Wireless Local Area MeshNetworking.