Module 8 : Mesh

Module 8 : Mesh

Mesh Concept

A mesh network is a type of network topology where devices, called nodes, are interconnected, allowing data to be transmitted between them even if some nodes are out of direct range of each other. This creates a robust and self-healing network where data can take multiple paths to reach its destination, enhancing reliability and coverage.

In a mesh network, each node (such as an ESP32) can both send and receive data, effectively acting as a repeater. This decentralized approach contrasts with traditional star networks, where communication is routed through a central hub.

The use of ESP32 microcontrollers in mesh networks is particularly effective due to their low power consumption, Wi-Fi capabilities, and flexibility in handling communication protocols. By leveraging ESP32 devices, you can create a scalable and resilient mesh network suitable for applications like home automation, IoT, and remote sensing.

Types of Mesh Networks

BLE Mesh

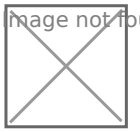
Bluetooth Low Energy (BLE) Mesh is a wireless communication standard designed for low-power devices to create a mesh network. BLE Mesh allows devices to communicate with each other over relatively short distances, typically up to 100 meters, by relaying messages through intermediate nodes. It's optimized for scenarios where low power consumption is crucial, such as smart lighting, industrial IoT, and home automation. BLE Mesh networks are highly scalable, supporting thousands of nodes, and are commonly used in environments where devices need to communicate with each other without requiring high data rates.

Wi-Fi Mesh

Wi-Fi Mesh is a network topology where multiple Wi-Fi access points (nodes) work together to provide seamless and extended wireless coverage over a larger area. In a Wi-Fi Mesh network, each node communicates with its neighboring nodes, allowing devices to connect to the strongest signal available. This ensures consistent and reliable connectivity across different parts of a home, office, or larger space. Wi-Fi Mesh networks are typically used to eliminate dead zones and improve network coverage and speed. They are ideal for applications requiring high data throughput, such as streaming, online gaming, and handling large amounts of data.

	BLE Mesh	Wi-Fi Mesh
Range and Coverage	Typically covers shorter distances (up to 100 meters per hop), suitable for low-power, short-range applications.	Offers broader coverage over larger areas, with nodes typically placed farther apart.
Power Consumption	Optimized for low power consumption, making it ideal for battery-operated devices and scenarios where energy efficiency is critical.	Generally consumes more power, suitable for applications where devices are typically connected to a power source.
Data Rate	Supports lower data rates, sufficient for simple control commands, sensor data, and small amounts of information.	Supports higher data rates, enabling faster data transfer and handling more bandwidth-intensive tasks.
Applications	Commonly used in smart homes, wearables, industrial IoT, and other environments where low power and low data rate communication are sufficient.	Used in residential, commercial, and enterprise environments where consistent, high-speed internet connectivity is needed.
Scalability	Can support thousands of nodes in a network due to its efficient communication protocol.	Typically involves fewer nodes, as each node is more powerful and covers a larger area.

Traditional Wi-Fi Network Architecture



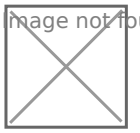
A traditional infrastructure Wi-Fi network operates as a point-to-multipoint system where a central node, called the access point (AP), directly connects to all other nodes, known as stations. The AP manages and forwards transmissions between these stations, and in some cases, it also handles communication with an external IP network through a router. However, this type of Wi-Fi network has the drawback of limited coverage, as each station must be within range of the AP to connect. Additionally, traditional Wi-Fi networks can become overloaded since the number of stations that can connect is restricted by the AP's capacity.

ESP Wi-Fi Mesh

ESP-WIFI-MESH differs from traditional infrastructure Wi-Fi networks by eliminating the need for nodes to connect to a central node. Instead, nodes can connect with neighboring nodes, with each node sharing the responsibility of relaying transmissions. This design enables an ESP-WIFI-MESH

network to cover a much larger area since nodes can remain interconnected even when they are out of range of a central node. Additionally, ESP-WIFI-MESH is less prone to overloading because the number of nodes on the network is not constrained by the capacity of a single central node.

Node Types



Root Node

The root node is the highest node in a Wi-Fi mesh network and acts as the sole interface between the mesh network and an external IP network. It connects to a conventional Wi-Fi router and is responsible for relaying packets between the external IP network and the nodes within the mesh network. There can only be one root node in a Wi-Fi mesh network, and its only upstream connection is to the router. In the diagram above, node A is the root node of the network.

Leaf Nodes

A leaf node is a node that cannot have any child nodes (no downstream connections). It can only send or receive its own packets and cannot forward packets from other nodes. A node is designated as a leaf node if it is located at the network’s maximum allowed layer, preventing it from creating downstream connections and thus avoiding the addition of an extra layer to the network. Additionally, nodes without a softAP interface (station-only nodes) are assigned as leaf nodes, as a softAP interface is required for downstream connections. In the diagram above, nodes L, M, and N are positioned at the network's maximum layer and are therefore designated as leaf nodes.

Intermediate Parent Nodes

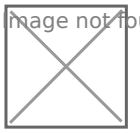
Nodes that are neither the root node nor leaf nodes are considered intermediate parent nodes. An intermediate parent node must have one upstream connection (one parent node) and can have zero to multiple downstream connections (zero to multiple child nodes). These nodes can transmit and receive packets as well as forward packets from their upstream and downstream connections. In the diagram above, nodes B to J are intermediate parent nodes. Intermediate parent nodes without downstream connections, such as nodes E, F, G, I, and J, differ from leaf nodes because they can still form downstream connections in the future.

Idle Nodes

Nodes that have not yet joined the network are classified as idle nodes. These nodes will try to establish an upstream connection with an intermediate parent node or attempt to become the root node under certain conditions (see Automatic Root Node Selection). In the diagram above, nodes K

and O are idle nodes.

Building a Mesh Network



The process of building a Wi-Fi mesh network involves first selecting a root node, followed by establishing downstream connections layer by layer until all nodes have joined the network. The specific structure of the network may vary based on factors such as root node selection, parent node selection, and asynchronous power-on reset. However, the Wi-Fi mesh network building process can generally be summarized in the following steps:

1. Root Node Selection

The root node can either be designated during the configuration process (see the section on User Designated Root Node) or dynamically elected based on the signal strength between each node and the router (see Automatic Root Node Selection). Once the root node is selected, it connects to the router and begins facilitating downstream connections. In the figure above, node A is chosen as the root node and forms an upstream connection with the router.

2. Second Layer Formation

After the root node connects to the router, idle nodes within range of the root node begin connecting to it, creating the second layer of the network. These second-layer nodes become intermediate parent nodes (assuming the maximum permitted layers exceed two), enabling the formation of the next layer. In the figure above, nodes B to D are within range of the root node, so they connect to it and become intermediate parent nodes.

3. Formation of Remaining Layers

The remaining idle nodes connect with nearby intermediate parent nodes, forming new layers within the network. Upon connection, these idle nodes become either intermediate parent nodes or leaf nodes, depending on the network’s maximum permitted layers. This process repeats until all idle nodes have joined the network or the maximum layer depth has been reached. In the figure above, nodes E, F, and G connect to nodes B, C, and D respectively, becoming intermediate parent nodes themselves.

4. Limiting Tree Depth

To ensure the network does not exceed the maximum allowed number of layers, nodes at the maximum layer automatically become leaf nodes once they connect. This prevents any further idle nodes from connecting to these leaf nodes, thereby preventing the formation of an additional layer. If an idle node cannot find a suitable parent node, it will remain idle indefinitely. In the figure

above, the network’s maximum permitted layers are set to four, so when node H connects, it becomes a leaf node to prevent any further downstream connections.

PainlessMesh

Intro to painlessMesh

painlessMesh is a library that takes care of the particulars of creating a simple mesh network using esp8266 and esp32 hardware. The goal is to allow the programmer to work with a mesh network without having to worry about how the network is structured or managed.

True ad-hoc networking

painlessMesh is a true ad-hoc network, meaning that no-planning, central controller, or router is required. Any system of 1 or more nodes will self-organize into fully functional mesh. The maximum size of the mesh is limited (we think) by the amount of memory in the heap that can be allocated to the sub-connections buffer and so should be really quite high.

JSON based

painlessMesh uses JSON objects for all its messaging. There are a couple of reasons for this. First, it makes the code and the messages human readable and painless to understand and second, it makes it painless to integrate painlessMesh with javascript front-ends, web applications, and other apps. Some performance is lost, but I haven’t been running into performance issues yet. Converting to binary messaging would be fairly straight forward if someone wants to contribute.

Wifi & Networking

painlessMesh is designed to be used with Arduino, but it does not use the Arduino WiFi libraries, as we were running into performance issues (primarily latency) with them. Rather the networking is all done using the native esp32 and esp8266 SDK libraries, which are available through the Arduino IDE. Hopefully though, which networking libraries are used won’t matter to most users much as you can just include painlessMesh.h, run the init() and then work the library through the API.

painlessMesh is not IP networking

painlessMesh does not create a TCP/IP network of nodes. Rather each of the nodes is uniquely identified by its 32bit chipId which is retrieved from the esp8266/esp32 using the

`system_get_chip_id()` call in the SDK. Every node will have a unique number. Messages can either be broadcast to all of the nodes on the mesh, or sent specifically to an individual node which is identified by its `nodeId.

Limitations and caveats

- Try to avoid using `delay()` in your code. To maintain the mesh we need to perform some tasks in the background. Using `delay()` will stop these tasks from happening and can cause the mesh to lose stability/fall apart. Instead we recommend using TaskScheduler which is used in `painlessMesh` itself. Documentation can be found here. For other examples on how to use the scheduler see the example folder.
- `painlessMesh` subscribes to WiFi events. Please be aware that as a result `painlessMesh` can be incompatible with user programs/other libraries that try to bind to the same events.
- Try to be conservative in the number of messages (and especially broadcast messages) you sent per minute. This is to prevent the hardware from overloading. Both esp8266 and esp32 are limited in processing power/memory, making it easy to overload the mesh and destabilize it. And while `painlessMesh` tries to prevent this from happening, it is not always possible to do so.
- Messages can go missing or be dropped due to high traffic and you can not rely on all messages to be delivered. One suggestion to work around is to resend messages every so often. Even if some go missing, most should go through. Another option is to have your nodes send replies when they receive a message. The sending nodes can the resend the message if they haven't gotten a reply in a certain amount of time.

Installation

`painlessMesh` is included in both the Arduino Library Manager and the platformio library registry and can easily be installed via either of those methods.

Dependencies

painlessMesh makes use of the following libraries, which can be installed through the Arduino Library Manager

- ArduinoJson
- TaskScheduler
- ESPAsyncTCP (ESP8266)
- AsyncTCP (ESP32)

If platformio is used to install the library, then the dependencies will be installed automatically.

Examples

StartHere is a basic how to use example. It blinks built-in LED (in ESP-12) as many times as nodes are connected to the mesh. Further examples are under the examples directory and shown on the platformio page.

Development on your own machine

After cloning the repository, you will need to initialize and update the submodules.

```
git submodule init
git submodule update
```

After that you can compile the library using the following commands

```
cmake -G Ninja
ninja
```

This will compile a number of test files under `./bin/catch_` that can be run. For example using:

```
run-parts --regex catch_ bin/
```

Getting help

There is help available from a variety of sources:

- The included examples
- The API documentation
- The wiki
- On our new forum/mailinglist
- On the gitter channel

Contributing

We try to follow the git flow development model. Which means that we have a `develop` branch and `master` branch. All development is done under feature branches, which are (when finished) merged into the development branch. When a new version is released we merge the `develop` branch into the `master` branch. For more details see the CONTRIBUTING file.

painlessMesh API

Using painlessMesh is painless!

First include the library and create an painlessMesh object like this.

```
#include <painlessMesh.h>
painlessMesh  mesh;
```

The main member functions are included below. **Full documentation can be found here**

Member Functions

void painlessMesh::init(String ssid, String password, uint16_t port = 5555, WiFiMode_t connectMode = WIFI_AP_STA, _auth_mode authmode = AUTH_WPA2_PSK, uint8_t channel = 1, phy_mode_t phymode = PHY_MODE_11G, uint8_t maxtpw = 82, uint8_t hidden = 0, uint8_t maxconn = 4)

Add this to your setup() function. Initialize the mesh network. This routine does the following things.

- Starts a wifi network
- Begins searching for other wifi networks that are part of the mesh
- Logs on to the best mesh network node it finds... if it doesn't find anything, it starts a new search in 5 seconds.

`ssid` = the name of your mesh. All nodes share same AP ssid. They are distinguished by BSSID. `password` = wifi password to your mesh. `port` = the TCP port that you want the mesh server to run on. Defaults to 5555 if not specified. `connectMode` = switch between WIFI_AP, WIFI_STA and WIFI_AP_STA (default) mode

void painlessMesh::stop()

Stop the node. This will cause the node to disconnect from all other nodes and stop/sending messages.

void painlessMesh::update(void)

Add this to your loop() function This routine runs various maintenance tasks... Not super interesting, but things don't work without it.

void painlessMesh::onReceive(&receivedCallback)

Set a callback routine for any messages that are addressed to this node. Callback routine has the following structure.

`void receivedCallback( uint32_t from, String &amp;msg )`

Every time this node receives a message, this callback routine will the called. “from” is the id of the original sender of the message, and “msg” is a string that contains the message. The message can be anything. A JSON, some other text string, or binary data.

void painlessMesh::onNewConnection(&newConnectionCallback)

This fires every time the local node makes a new connection. The callback has the following structure.

`void newConnectionCallback( uint32_t nodeId )`

`nodeId` is new connected node ID in the mesh.

void painlessMesh::onChangedConnections(&changedConnectionsCallback)

This fires every time there is a change in mesh topology. Callback has the following structure.

`void onChangedConnections()`

There are no parameters passed. This is a signal only.

bool painlessMesh::isConnected(nodeId)

Returns if a given node is currently connected to the mesh.

`nodeId` is node ID that the request refers to.

void painlessMesh::onNodeTimeAdjusted(&nodeTimeAdjustedCallback)

This fires every time local time is adjusted to synchronize it with mesh time. Callback has the following structure.

`void onNodeTimeAdjusted(int32_t offset)`

`offset` is the adjustment delta that has been calculated and applied to local clock.

void onNodeDelayReceived(nodeDelayCallback_t onDelayReceived)

This fires when a time delay measurement response is received, after a request was sent. Callback has the following structure.

`void onNodeDelayReceived(uint32_t nodeId, int32_t delay)`

`nodeId` The node that originated response.

`delay` One way network trip delay in microseconds.

bool painlessMesh::sendBroadcast(String &msg, bool includeSelf = false)

Sends msg to every node on the entire mesh network. By default the current node is excluded from receiving the message (`includeSelf = false`). `includeSelf = true` overrides this behavior, causing the `receivedCallback` to be called when sending a broadcast message.

returns true if everything works, false if not. Prints an error message to Serial.print, if there is a failure.

bool painlessMesh::sendSingle(uint32_t dest, String &msg)

Sends msg to the node with Id == dest.

returns true if everything works, false if not. Prints an error message to Serial.print, if there is a failure.

String painlessMesh::subConnectionJson()

Returns mesh topology in JSON format.

std::list<uint32_t> painlessMesh::getNodeList()

Get a list of all known nodes. This includes nodes that are both directly and indirectly connected to the current node.

uint32_t painlessMesh::getNodeId(void)

Return the chipId of the node that we are running on.

uint32_t painlessMesh::getNodeTime(void)

Returns the mesh timebase microsecond counter. Rolls over 71 minutes from startup of the first node.

Nodes try to keep a common time base synchronizing to each other using an SNTP based protocol

bool painlessMesh::startDelayMeas(uint32_t nodeId)

Sends a node a packet to measure network trip delay to that node. Returns true if nodeId is connected to the mesh, false otherwise. After calling this function, user program have to wait to the response in the form of a callback specified by `void painlessMesh::onNodeDelayReceived(nodeDelayCallback_t onDelayReceived)`.

nodeDelayCallback_t is a function in the form of `void (uint32_t nodeId, int32_t delay)`.

void painlessMesh::stationManual(String ssid, String password, uint16_t port, uint8_t *remote_ip)

Connects the node to an AP outside the mesh. When specifying a `remote_ip` and `port`, the node opens a TCP connection after establishing the WiFi connection.

Note: The mesh must be on the same WiFi channel as the AP.

void painlessMesh::setDebugMsgTypes(uint16_t types)

Change the internal log level. List of types defined in Logger.hpp: ERROR | MESH_STATUS | CONNECTION | SYNC | COMMUNICATION | GENERAL | MSG_TYPES | REMOTE

Example Codes

Sender Node

Below is an example code of a node which broadcasts a message to every other node in the mesh network every 10 seconds. A sender node usually behaves as child nodes.

```
#include <painlessMesh.h>

// Mesh network parameters
#define MESH_PREFIX     "yourMeshNetwork"
#define MESH_PASSWORD   "yourMeshPassword"
#define MESH_PORT       5555

painlessMesh mesh;

// FreeRTOS Task Handle for mesh updates
TaskHandle_t meshUpdateTaskHandle;

// Function to handle received messages
void receivedCallback(uint32_t from, String &msg) {
    Serial.printf("Received message from node %u: %s\n", from, msg.c_str());
}

// Task to continuously update the mesh network
void meshUpdateTask(void *pvParameters) {
    while (true) {
        mesh.update();
        vTaskDelay(10 / portTICK_PERIOD_MS);  // Short delay to yield to other tasks
    }
}

void setup() {
    Serial.begin(115200);

    // Initialize mesh network
```

```
    mesh.setDebugMsgTypes(ERROR | STARTUP | CONNECTION);  // Debug message types
    mesh.init(MESH_PREFIX, MESH_PASSWORD, MESH_PORT);
    mesh.onReceive(&receivedCallback);  // Register the message receive callback

    // Create FreeRTOS task for updating mesh
    xTaskCreate(
        meshUpdateTask,             // Function to implement the task
        "MeshUpdateTask",           // Task name
        8192,                       // Stack size
        NULL,                       // Task input parameter
        1,                          // Priority
        &meshUpdateTaskHandle       // Task handle
    );
}

void loop() {

}
```

Receiver Node

Below is an example code of a node which receives messages from every other node in the mesh network that are within range. A receiver node usually behaves as parent nodes.

```
#include <painlessMesh.h>

// Mesh network parameters
#define MESH_PREFIX     "yourMeshNetwork"
#define MESH_PASSWORD   "yourMeshPassword"
#define MESH_PORT       5555

painlessMesh mesh;

// FreeRTOS Task Handle for mesh updates
TaskHandle_t meshUpdateTaskHandle;

// Function to handle received messages
void receivedCallback(uint32_t from, String &msg) {
    Serial.printf("Received message from node %u: %s\n", from, msg.c_str());
```

```
}

// Task to continuously update the mesh network
void meshUpdateTask(void *pvParameters) {
    while (true) {
        mesh.update();
        vTaskDelay(10 / portTICK_PERIOD_MS);  // Short delay to yield to other tasks
    }
}

void setup() {
    Serial.begin(115200);

    // Initialize mesh network
    mesh.setDebugMsgTypes(ERROR | STARTUP | CONNECTION);  // Debug message types
    mesh.init(MESH_PREFIX, MESH_PASSWORD, MESH_PORT);
    mesh.onReceive(&receivedCallback);  // Register the message receive callback

    // Create FreeRTOS task for updating mesh
    xTaskCreate(
        meshUpdateTask,             // Function to implement the task
        "MeshUpdateTask",           // Task name
        8192,                       // Stack size
        NULL,                       // Task input parameter
        1,                         // Priority
        &meshUpdateTaskHandle       // Task handle
    );
}

void loop() {

}
```

Root Node with MQTT Bridge

```
#include <Arduino.h>
#include <painlessMesh.h>
#include <PubSubClient.h>
#include <WiFiClient.h>
```

```
#define   MESH_PREFIX     "whateverYouLike"
#define   MESH_PASSWORD   "somethingSneaky"
#define   MESH_PORT       5555

#define   STATION_SSID     "YourAP_SSID"
#define   STATION_PASSWORD "YourAP_PWD"

#define HOSTNAME "MQTT_Bridge"

// Prototypes
void receivedCallback( const uint32_t &from, const String &msg );
void mqttCallback(char* topic, byte* payload, unsigned int length);

IPAddress getlocalIP();

IPAddress myIP(0,0,0,0);
IPAddress mqttBroker(192, 168, 1, 1);

painlessMesh  mesh;
WiFiClient wifiClient;
PubSubClient mqttClient(mqttBroker, 1883, mqttCallback, wifiClient);

void setup() {
  Serial.begin(115200);

  mesh.setDebugMsgTypes( ERROR | STARTUP | CONNECTION );  // set before init() so that you can see startup
messages

  // Channel set to 6. Make sure to use the same channel for your mesh and for you other
  // network (STATION_SSID)
  mesh.init( MESH_PREFIX, MESH_PASSWORD, MESH_PORT, WIFI_AP_STA, 6 );
  mesh.onReceive(&receivedCallback);

  mesh.stationManual(STATION_SSID, STATION_PASSWORD);
  mesh.setHostname(HOSTNAME);

  // Bridge node, should (in most cases) be a root node. See [the
wiki](https://gitlab.com/painlessMesh/painlessMesh/wikis/Possible-challenges-in-mesh-formation) for some
background
  mesh.setRoot(true);
  // This node and all other nodes should ideally know the mesh contains a root, so call this on all nodes
```

```
  mesh.setContainsRoot(true);
}

void loop() {
  mesh.update();
  mqttClient.loop();

  if(myIP != getlocalIP()){
    myIP = getlocalIP();
    Serial.println("My IP is " + myIP.toString());

    if (mqttClient.connect("painlessMeshClient")) {
      mqttClient.publish("painlessMesh/from/gateway","Ready!");
      mqttClient.subscribe("painlessMesh/to/#");
    }
  }
}

void receivedCallback( const uint32_t &from, const String &msg ) {
  Serial.printf("bridge: Received from %u msg=%s\n", from, msg.c_str());
  String topic = "painlessMesh/from/" + String(from);
  mqttClient.publish(topic.c_str(), msg.c_str());
}

void mqttCallback(char* topic, uint8_t* payload, unsigned int length) {
  char* cleanPayload = (char*)malloc(length+1);
  memcpy(cleanPayload, payload, length);
  cleanPayload[length] = '\0';
  String msg = String(cleanPayload);
  free(cleanPayload);

  String targetStr = String(topic).substring(16);

  if(targetStr == "gateway")
  {
    if(msg == "getNodes")
    {
      auto nodes = mesh.getNodeList(true);
      String str;
      for (auto &&id : nodes)
        str += String(id) + String(" ");
```

```
      mqttClient.publish("painlessMesh/from/gateway", str.c_str());
    }
  }
  else if(targetStr == "broadcast")
  {
    mesh.sendBroadcast(msg);
  }
  else
  {
    uint32_t target = strtoul(targetStr.c_str(), NULL, 10);
    if(mesh.isConnected(target))
    {
      mesh.sendSingle(target, msg);
    }
    else
    {
      mqttClient.publish("painlessMesh/from/gateway", "Client not connected!");
    }
  }
}

IPAddress getlocalIP() {
  return IPAddress(mesh.getStationIP());
}
```