



EAST WEST UNIVERSITY

Mesh Networking with NodeMCU ESP8266 and painlessMesh

Course Title: Internet of Things

Course Code: CSE 406

Section: 01

Submitted by

Name: Nushrat Jaben Aurnima

ID: 2022-2-60-146

Date: 10/08/2025

Submitted to

Dr. Raihan Ul Islam (DRUI)

Associate Professor, Department of Computer Science and Engineering

East West University

Other Group Members

<i>Name</i>	<i>ID</i>
<i>Zihad khan</i>	<i>2022-2-60-107</i>
<i>Shairin Akter Hashi</i>	<i>2022-2-60-102</i>
<i>Shahrukh Hossain Shihab</i>	<i>2022-1-60-372</i>

Table of Contents

Introduction	2
Required Materials	2
Hardware	2
Software	2
Tools.....	3
Tasks and Implementation	3
Task 1: Message Interpretation.....	3
Interpretation of Messages.....	3
Broadcast Communication Flow	4
Task 2: Message Interpretation.....	5
Implementation	5
Task 3: Multi-Hop Direct Messaging Based on Signal Strength	7
Network Configuration.....	7
Implementation and Observation.....	7
Advantages of Mesh Topology and Potential Applications	11
Advantages.....	11
Potential Applications	11
Conclusion.....	11

Introduction

In Modern Wireless Communication, mesh networking enables the networking device to communicate flexibly without relay on the Central Hub. This lab focuses on **PainlessMesh**, an **Arduino-compatible** library that simplifies the construction of dynamic network using **ESP32** or **ESP8266** boards. PainlessMesh allows Each Node to Automatically Connect, Makes A Self-Healing Mesh, and Shares Messages efficiently on many hops. In this experiment, we tried to find out:

- Different system messages like *New Connection*, *Connection Change*, and *Adjusted Time* so on.
- Direct node-to-node messaging,
- Multi-hop communication depending on signal strength.

Through practical analysis, the laboratory indicates how PainlessMesh constructs a routing path based on dynamic node availability and signal quality, making it an ideal solution for IoT and smart environment.

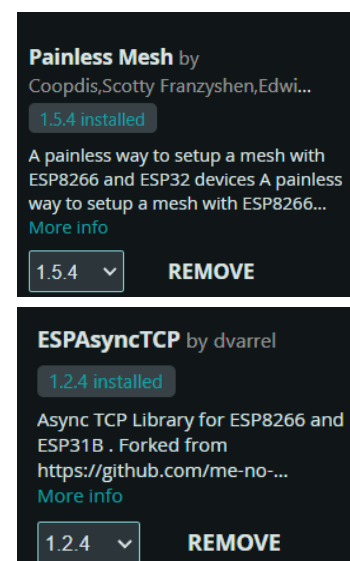
Required Materials

Hardware

- **3 ESP8266 Boards:** Used as individual mesh nodes for communication and routing.

Software

- **Arduino IDE:** For writing, compiling, and uploading code to the microcontrollers.
- **PainlessMesh Library:** A library used to enable mesh networking among the ESP boards. Installed via Arduino Library Manager.
- **ESPAsyncTCP Library:** Required by PainlessMesh for handling asynchronous TCP communication. Installed via the Library Manager.



Tools

- **USB Cables:** For connecting the ESP boards to the computer for programming and powering.
- **Serial Monitor:** Used for observing node behavior and messages exchanged between nodes during the experiment.


Tasks and Implementation

Task 1: Message Interpretation








The aim of this task was to understand the standard system messages of the PainlessMesh library and illustrate a broadcast system operation in a mesh network. These messages help to point out the internal characteristics of the Mesh network.

Interpretation of Messages

- **New connection:** This message is sent when a new node joins the network. It notices successful inclusion of the node in the system.





```
23:43:46.991 ->  Connected with Node ID: 110046777
```

- **Connection change:** This message shows when a connection of a node changes, for example, when disconnecting or reinforcing. It helps to monitor self-treatment behavior and network stability.

```
23:46:01.755 ->  Message sent to 1163270074   
23:46:06.452 ->  Message sent to 1163270074   
23:46:10.980 ->  Message sent to 1163270074   
23:46:12.979 ->  Connection list changed
```

- **Adjusted time:** This message works as a synchronization alert across Mesh Network has received. It ensures all nodes share a common time reference for timed tasks.

Example log:

```
23:44:47.260 ->  Time adjusted. Offset = -2920  
23:44:47.260 ->  Time adjusted. Offset = 8614  
23:44:47.446 ->  Time adjusted. Offset = -8761  
23:44:47.677 ->  Time adjusted. Offset = 37
```

In short, these messages help in debugging, observing network changes, and verifying that the mesh is functioning as intended.

Broadcast Communication Flow

In broadcast communication model, a sender transmits a message to **all nodes** in the mesh network.

- **Sender Node** (from Sender_Task1.ino):

```
void sendMessage() {  
  String msg = "Hello from Node " + String(mesh.getNodeId()) + " 🌐";  
  mesh.sendBroadcast(msg);  
  Serial.println(" 📡 Broadcast message sent to all nodes.");  
  taskSendMessage.setInterval(random(TASK_SECOND * 5, TASK_SECOND * 10));  
}
```

- **Receiver Nodes** (from Receiver1.ino and Receiver2.ino):

```
void receivedCallback(uint32_t from, String &msg) {  
  Serial.println("=====");  
  Serial.printf(" 📡 Message received from Node %u\n", from);  
  Serial.printf(" 📧 Message : %s\n", msg.c_str());  
  Serial.println("=====\\n");  
}
```

Process:

1. The sender broadcasts a message to all connected nodes.
2. Each receiver displays the message on the Serial Monitor.
3. Receivers send an acknowledgment (ACK) directly back to the sender.

Results:

- **Sender Output:** Broadcast sent to all nodes with ACKs received from each.

```
23:07:05.830 -> 🔗 New Node Connected: 110046777  
23:07:05.918 -> 📡 Broadcast message sent to all nodes.  
23:07:05.918 -> =====  
23:07:05.918 -> 📡 ACK received from Node 110046777 ✅  
23:07:05.918 -> 📧 Message : ✅ ACK from Node 110046777  
23:07:05.918 -> =====  
23:07:05.918 -> =====  
23:07:06.078 -> =====  
23:07:06.078 -> 📡 ACK received from Node 1163269996 ✅  
23:07:06.078 -> 📧 Message : ✅ ACK from Node 1163269996  
23:07:06.078 -> =====  
23:07:06.078 -> =====  
23:07:12.704 -> ⓘ [SENDER] My Node ID: 1163270074
```

- **Receiver 1 Output:** Successfully received the broadcast and sent back an ACK.

```
22:54:46.853 -> i [RECEIVER] My Node ID: 1163269996
22:54:49.283 -> =====
22:54:49.283 -> 📧 Message received from Node 1163270074
22:54:49.283 -> 📧 Message : Hello from Node 1163270074 🌐
22:54:49.283 -> =====
22:54:49.283 ->
22:54:49.283 -> 📩 Sent ACK to Node 1163270074
22:54:56.861 -> i [RECEIVER] My Node ID: 1163269996
22:54:57.710 -> =====
22:54:57.710 -> 📧 Message received from Node 1163270074
22:54:57.710 -> 📧 Message : Hello from Node 1163270074 🌐
22:54:57.749 -> =====
22:54:57.749 ->
22:54:57.749 -> 📩 Sent ACK to Node 1163270074
```

- **Receiver 2 Output:** Successfully received the broadcast and sent back an ACK.

```
22:55:50.079 -> i [RECEIVER] My Node ID: 110046777
22:55:54.548 -> =====
22:55:54.548 -> 📧 Message received from Node 1163270074
22:55:54.591 -> 📧 Message : Hello from Node 1163270074 🌐
22:55:54.591 -> =====
22:55:54.591 ->
22:55:54.591 -> 📩 Sent ACK to Node 1163270074
22:56:00.092 -> i [RECEIVER] My Node ID: 110046777
22:56:00.377 -> =====
22:56:00.377 -> 📧 Message received from Node 1163270074
22:56:00.377 -> 📧 Message : Hello from Node 1163270074 🌐
22:56:00.377 -> =====
22:56:00.377 ->
22:56:00.377 -> 📩 Sent ACK to Node 1163270074
```

Task 2: Message Interpretation

The aim of this task is to change the existing code so that a message is only sent to a specific node instead of broadcasting all nodes in the web. This demonstrates how to implement targeted communication using the **Mesh.SendSingle()** feature from the **PainlessMesh** Library.

Implementation

1. Modify the Sender Code

- The **sendMessage()** function was updated to use **mesh.sendSingle(targetNodeId, msg)** instead of broadcasting.

```
bool success = mesh.sendSingle(targetNodeId, msg);
if (success) {
    Serial.printf("📩 Message sent to %u ✅\n", targetNodeId);
} else {
    Serial.printf("❌ Failed to send to %u\n", targetNodeId);
}
```

- Here, **targetNodeId** is hardcoded as the ID of the intended receiver.

```
uint32_t targetNodeId = 1163270074;
```

- Therefore the output

```
23:43:49.289 -> 📬 Message sent to 1163270074 ✓
23:43:52.554 -> 📬 Message sent to 1163270074 ✓
23:43:54.749 -> 📬 Message sent to 1163270074 ✓
```

2. Receiver Code Behavior

- Target Receiver:** Displays the received message and sends back an acknowledgment.

```
00:07:15.362 -> 📬 Message received!
00:07:15.362 -> 🧑 Sender Node ID : 1163269996
00:07:15.362 -> 📬 Message : Hello from Node 1163269996 🚀 | Mesh network communication successful!
00:07:15.362 -> =====
```

- Non-Target Receiver:** Remains idle (no messages displayed). This proves that the direct message appeared only on the target node's serial monitor.

```
23:46:34.941 -> ver Ready | My Node ID: 110046777
23:46:34.941 -> 🔗 Connected with Node ID: 1163269996
23:47:09.680 -> 🔗 Connected with Node ID: 1163270074
```

3. Handling Disconnected Target Nodes

- To handle cases where the target node is not connected, the function **mesh.isConnected(nodeId)** was used before sending.
- If the node is disconnected, the system displays an error message.

```
23:43:25.331 -> ❌ Failed to send to 1163270074
23:43:25.331 -> ❌ Failed to send to 1163270074
23:43:25.331 -> ❌ Failed to send to 1163270074
```

Task 3: Multi-Hop Direct Messaging Based on Signal Strength

The purpose of this task is to demonstrate a **multi-hop** direct messaging using a **painlessMesh** library, where a message travels through an intermediate node when the sender and the target are out of the direct range.




Network Configuration

- **Node A (Sender) → ID: 1163269996**
- **Node B (Intermediate) → ID: 110046777**
- **Node C (Target Receiver) → ID: 1163270074**
- Node A and Node C were placed far enough apart so that they could not communicate directly. Node B was positioned between them to act as a relay.

Implementation and Observation



Step 1:

- **Sender (Node A)** sends a message to Node C, even though they were out of direct range. Therefore, message sent successfully.

```
00:34:23.078 ->  Message sent to 1163270074   
00:37:06.056 ->  Connection list changed
```

Step 2:

- **Intermediate Node B** log entries confirmed the message.

```
00:42:01.087 ->  Node B Connected with Node ID: 1163269996  
00:44:01.677 ->  Node B Connected with Node ID: 1163270074
```

- **mesh.setDebugMsgTypes(ERROR | STARTUP | CONNECTION)**
 - Enables **painlessMesh** to print **detailed connection activities** such as
 - Station mode connections (Station Mode Connected)
 - New node joining the mesh (New STA connection incoming)
 - Node ID registration (Node B Connected with Node ID:12)

- Network scans showing available nodes and their signal strengths
(scanComplete() with RSSI values).

```
00:45:11.069 -> CONNECTION: Event: Station Mode Connected
00:45:12.887 -> CONNECTION: Event: Station Mode Got IP (IP: 10.23.186.100 Mask: 255.255.255.0 Gateway: 10.23.186.1)
00:45:12.887 -> CONNECTION: New STA connection incoming
00:45:12.887 -> CONNECTION: painlessmesh::Connection: New connection established.
00:45:12.931 -> CONNECTION: newConnectionTask():
00:45:12.931 -> CONNECTION: newConnectionTask(): adding 1163270074 now= 9055756
```

- `mesh.init(MESH_PREFIX, MESH_PASSWORD, &userScheduler, MESH_PORT)`
 - Initializes the mesh network with the given SSID, password, and port.
 - This triggers **startup and initial connection logs** as the node begins participating in the mesh.
- `newConnectionCallback()`
 - Executes whenever a new node connects.
 - Prints the connected node's unique ID to the Serial Monitor.

```
00:45:12.931 -> ☞ Node B Connected with Node ID: 1163270074
00:45:23.410 -> CONNECTION: stationScan(): cse406
00:45:24.981 -> CONNECTION: scanComplete(): Scan finished
00:45:24.981 -> CONNECTION: scanComplete():-- > Cleared old APs.
00:45:24.981 -> CONNECTION: scanComplete(): num = 10
00:45:25.025 -> CONNECTION: found : cse406, -35dBm
00:45:25.025 -> CONNECTION: found : cse406, -55dBm
00:45:25.025 -> CONNECTION: Found 2 nodes
00:45:25.025 -> CONNECTION: connectToAP(): Unknown nodes found. Current stability: 37
00:46:55.016 -> CONNECTION: stationScan(): cse406
00:46:56.612 -> CONNECTION: scanComplete(): Scan finished
00:46:56.612 -> CONNECTION: scanComplete():-- > Cleared old APs.
00:46:56.612 -> CONNECTION: scanComplete(): num = 8
00:46:56.612 -> CONNECTION: found : cse406, -35dBm
00:46:56.612 -> CONNECTION: found : cse406, -57dBm
00:46:56.612 -> CONNECTION: Found 2 nodes
00:46:56.612 -> CONNECTION: connectToAP(): Already connected, and no unknown nodes found: scan rate set to slow
00:47:56.605 -> CONNECTION: stationScan(): cse406
00:47:58.207 -> CONNECTION: scanComplete(): Scan finished
00:47:58.207 -> CONNECTION: scanComplete():-- > Cleared old APs.
00:47:58.251 -> CONNECTION: scanComplete(): num = 11
00:47:58.251 -> CONNECTION: found : cse406, -33dBm
00:47:58.251 -> CONNECTION: found : cse406, -55dBm
00:47:58.251 -> CONNECTION: Found 2 nodes
00:47:58.251 -> CONNECTION: connectToAP(): Already connected, and no unknown nodes found: scan rate set to slow
```

- **Automatic Network Scanning**

- `painlessMesh` automatically performs periodic scans for nearby mesh nodes and logs:
 - The total number of nodes found
 - Their RSSI (signal strength in dBm)
 - Whether the connection is already established or if a new link should be made

Node B is shown **connected** to other nodes and actively scanning networks. Although Node B is connected to both sender and receiver, no message was printed in its serial monitor, proving it only relayed the message and did not receive it directly.

Step 3:

- `mesh.setDebugMsgTypes(CONNECTION | COMMUNICATION | ERROR | STARTUP)`
 - Instructs `painlessMesh` to print **both connection events** and **packet routing details**, including:
 - Node-to-node connection establishment (Connected with Node ID: ...)
 - Direct message sends (`sendSingle(): dest=...`)
 - Routing steps showing intermediate nodes (`routePackage(): Recvd from ...`)
 - Any startup or error information

```
00:38:29.676 -> 📡 Message sent to 1163270074 ✅
00:38:33.517 -> COMMUNICATION: sendSingle(): dest=1163270074 msg=Hello from Node 1163269996 📡 | Mesh network communication successful!
00:38:33.517 -> 📡 Message sent to 1163270074 ✅
00:38:34.205 -> COMMUNICATION: routePackage(): Recvd from 1163270074: {"nodeId":1163270074,"type":5,"dest":1163269996,"from":1163270074}
00:38:35.517 -> COMMUNICATION: routePackage(): Recvd from 110046777: {"nodeId":110046777,"type":5,"dest":1163269996,"from":110046777}
```

- `sendMessage()` with `sendSingle()`
 - Periodically sends a direct message from Node A to target Node C.
 - On success, COMMUNICATION logs confirm the send action and display the message content.

- **Automatic Routing Trace**

- painlessMesh automatically logs the full packet path.
- `sendSingle(): dest=1163270074` shows Node A's direct send request to Node C.
- `routePackage(): Recvd from 110046777` confirms Node B relayed the packet, proving **multi-hop routing**.

The CONNECTION and COMMUNICATION debug flags let painlessMesh display the entire journey of a message (sender to the target) without showing any intermediate hops. This demonstrates that the network successfully performed a multi-hop delivery via Node B.

Step 4:

- The nodes were placed at such distance that the target node initially appeared disconnected, triggering a “**Target Node ... is not connected!**” message. After re-scanning and re-establishing the mesh, the routing adapted automatically, using intermediate Node B to relay messages between Node A and Node C.

```
00:38:21.495 -> ⚠ Target Node 1163270074 is not connected!
00:38:23.797 -> CONNECTION: Event: Station Mode Got IP (IP: 10.23.186.100 Mask: 255.255.255.0 Gateway: 10.23.186.1)
00:38:23.864 -> CONNECTION: New STA connection incoming
00:38:23.864 -> CONNECTION: painlessmesh::Connection: New connection established.
00:38:23.938 -> COMMUNICATION: routePackage(): Recvd from 0: {"nodeId":1163270074,"type":6,"dest":1163269996,"from":1163270074}
00:38:23.972 -> 🔄 Connection list changed
00:38:23.972 -> CONNECTION: newConnectionTask():
00:38:23.972 -> CONNECTION: newConnectionTask(): adding 1163270074 now= 382046662
00:38:23.972 -> 📶 Connected with Node ID: 1163270074
00:38:24.046 -> COMMUNICATION: routePackage(): Recvd from 110046777: {"nodeId":110046777,"type":6,"dest":1163269996,"from":110046777}
00:38:24.046 -> COMMUNICATION: routePackage(): Recvd from 1163270074: {"type":4,"dest":1163269996,"from":1163270074,"msg":{"type":1,"t0":382136770}}
```

Step 5:

- In painlessMesh, hop selection is based on scanning nearby nodes and evaluating their signal strength (RSSI). The log shows **num = 7**, meaning seven net **Wi-Fi access points** were detected. Among these, two belong to **cse406** mesh, with signal strengths of **-38 dBm** and **-63 dBm**. The mesh chooses the stronger link (-38 dBm) as the best access point, ensuring data the most reliable path for data transfer

```
01:32:27.715 -> CONNECTION: stationScan(): cse406
01:32:29.866 -> CONNECTION: scanComplete(): Scan finished
01:32:29.866 -> CONNECTION: scanComplete():-- > Cleared old APs.
01:32:29.866 -> CONNECTION: scanComplete(): num = 7
01:32:29.866 -> CONNECTION: found : cse406, -38dBm
01:32:29.866 -> CONNECTION: found : cse406, -63dBm
01:32:29.900 -> CONNECTION: Found 2 nodes
01:32:29.900 -> CONNECTION: connectToAP(): Best AP is 1163269996<---
```

Advantages of Mesh Topology and Potential Applications

Advantages

- **Robustness:** Avoids single path failure by rerouting through other available paths.
- **Coverage:** Allows nodes far apart to connect via intermediate nodes, covering larger areas than a single hub.
- **Reliability** Ensures consistent communication by redundant paths.
- **Scalability:** Adds new nodes without reconfiguring the entire network and integrate automatically.
- **Load Distribution:** Spreads traffic across multiple routes, reducing congestion on any single path.

Potential Applications

- **Smart Homes:** Lighting, security, and appliance control without reliance on a central router. Thus, avoiding center failure problems like star topology.
- **Sensor Networks:** Environmental or agricultural monitoring where sensors are spread over large areas ensuring extended communication.
- **Industrial IoT System:** Reliable machine-to-machine communication in factories or plants. Since every node is directly connected with each other.

Conclusion

This lab has proven flexibility and robustness by using creating a dynamic mesh community and self-employing using the ESP8266. Mesh manages connections, directs messages, and adapts to adjustments in network conditions which are observed by involving operations such as transmission messages, direct node verbal exchange, and multi-hop routing based totally on sign strength. Practical experiments supported that mesh networks can preserve dependable communication, even when nodes are out of direct attain, proving their suitability to actual-world IoT applications, where resilience, coverage, and adaptability are critical.