Iraqi Journal of Industrial Research, Vol. 12, No. 2 (2025)

Open Access

Iraqi Journal of Industrial Research (IJOIR)

Journal homepage: http://ijoir.gov.iq

Deployment and Evaluation of Mesh Routing Protocols on Embedded Systems with Industrial Case Studies

[1]Ahmed A. Al-Healy*, [2]Qutaiba I. Ali

[1]Department of Vocational Education, Ministry of Education, Iraq

[2]Department of Computer Engineering, College of Engineering, University of Mosul, Iraq

Article information

Article history:

Received: August, 23, 2025

Accepted: October, 13, 2025

Available online: December, 15, 2025

Keywords:
Ad Hoc network,
OLSR,
OpenWRT,
Raspberry Pi,
Routing protocols

Corresponding Author:
Ahmed A. Al-Healy
ahmedalhealy91@gmail.com

DOI:
https://doi.org/10.53523/ijoirVol12I2ID599

Abstract

Wireless mesh networks (WMNs) are crucial for enabling communication without fixed infrastructure in various scenarios such as disaster response, rural connectivity, and educational experimentation. Although the Optimized Link State Routing (OLSR) protocol is widely studied in the research community, practical deployment reports remain limited and fragmented. This paper presents a clear and reproducible methodology for deploying OLSR (using OLSRd, a routing daemon that installs and updates routes in the Linux kernel) on embedded Linux systems, namely OpenWRT running on Raspberry Pi 3B+ devices. In addition, a Multi-Criteria Decision Analysis (MCDA) framework is applied to evaluate four distinct routing protocols (OLSRd, BATMAN, Babel, and HWMP) based on usability, configuration complexity, GUI support, documentation quality, and hardware compatibility. Experimental tests are conducted to measure network performance in terms of latency, throughput, and convergence time. Four case studies are also presented to show protocol suitability in different contexts, including community networks, IoT deployments, disaster simulations, and industrial environments. The results conclusively show that OLSRd is the most deployment-friendly protocol, combining procedural simplicity with reliable performance. This study provides practical guidance and valuable technical references for researchers, educators, and practitioners working on wireless mesh testbeds with embedded platforms, ultimately aiming to bridge the gap between academic theory and real-world application.

1. Introduction

Wireless multi-hop networks [1], particularly those based on ad hoc or mesh topologies, have gained substantial attention due to their decentralized nature, scalability, and suitability for infrastructure-less environments [2, 3]. These networks rely on routing protocols to maintain robust and adaptive connectivity in the face of dynamic topologies [4]. One prominent protocol in this context is the Optimized Link State Routing (OLSR) protocol [5], which exists in two versions: OLSRv1 and OLSRv2. While designed originally for static networks, it has also been considered for mobile scenarios, though its suitability in highly mobile environments requires careful validation [6, 7].

The OLSR protocol adopts a proactive routing strategy, maintaining and updating routing tables continuously to ensure low-latency route availability. OLSRv1, standardized in RFC 3626 [5], leverages Multipoint Relays (MPRs) to reduce control overhead typical of link-state approaches. OLSRv2, defined in RFC 7181 [8], enhances the protocol with modularity, improved IPv6 support, and more flexible routing metrics. Despite its aging development status and the discontinuation of official resources such as olsr.org, OLSR remains actively used in community mesh networks and academic experiments [9, 10, 11]. However, implementations often depend on manual builds, custom firmware, or outdated packages, typically requiring expert-level knowledge of embedded Linux systems.

From a platform perspective, OLSR is primarily supported on Unix-like operating systems, particularly Linux-based distributions including Ubuntu, Raspberry Pi OS, and OpenWRT. Among these, OpenWRT [12] is particularly well-suited for embedded applications, thanks to its lightweight architecture and support for networking extensions [13]. While official OpenWRT repositories no longer actively maintain OLSR packages, community-driven firmware such as Freifunk [14] continues to offer built-in OLSR support, reducing the complexity for non-expert users. Graphical interfaces are typically unavailable on conventional desktop environments, but OpenWRT provides a web-based GUI (Luci) that facilitates OLSR configuration—an advantage especially relevant for headless embedded devices like Raspberry Pi or Onion Omega2+.

The motivation for this work stems from the noticeable absence of practical documentation for deploying OLSR on embedded platforms. Most existing studies either remain theoretical or omit essential details about hardware setup and software configuration. Even works that report physical implementations tend to provide limited reproducibility due to missing procedural information. This gap is particularly problematic for researchers aiming to experiment with real-world wireless sensor networks or prototype mesh-based systems.

Accordingly, this paper presents a step-by-step practical guide for deploying OLSRv1 on embedded Linux-based platforms, with a particular focus on OpenWRT and Raspberry Pi 3B+. The purpose is not to introduce novel routing algorithms or comparative performance analysis, but to document the entire deployment process, from hardware and software prerequisites to installation steps, configuration options, and encountered challenges. In doing so, this work aims to serve as a foundational technical reference for researchers and practitioners seeking to implement functional wireless mesh testbeds using embedded devices.

2. Related Work

Much of the academic literature on OLSR has focused on protocol-level analysis, simulations, and performance evaluations under various network conditions. For example, studies have compared OLSR to AODV, DSR, and other routing protocols in terms of packet delivery ratio, delay, and throughput [15, 16] . However, these works often rely on simulators like NS-2, NS-3, or OMNeT++ and do not document physical implementation or testbed setup. In what follows, we highlight selected studies that have gone beyond simulation and have implemented OLSR in real-world testbeds. Table (1) summarizes the key characteristics of these empirical implementations, including hardware platforms, operating systems, and the routing protocols evaluated.

Sumarudin and Adiono (2015) [17] present the design and implementation of a high-throughput Wi-Fi-based wireless sensor network (WSN) using the OLSR protocol in a mesh topology. The hardware includes Raspberry Pi boards and I2C-based sensors. The software stack comprises the Linux-based Raspbian OS and Python for sensor interfacing and data transmission via Secure Shell (SSH). The system achieved robust performance with low jitter (3.87 ms), high throughput (8.5 Mbps), minimal packet loss (0.019%), and a range of up to 175 meters, demonstrating suitability for real-time environmental monitoring applications.

Lumbantoruan and Sagala (2015) [18] evaluated the performance of the OLSR routing protocol in ad hoc networks using a real-world testbed built with Raspberry Pi Model B devices and TP-Link WN722N USB Wi-Fi adapters. The study assessed network availability, multi-hop communication, and self-healing capabilities under varying distances and scenarios. The system used the Raspbian OS with OLSRd 0.6.6.1 installed. Results demonstrated that the OLSR protocol supports stable communication up to ~180 meters in line-of-sight and successfully reroutes traffic during link failures, confirming its suitability for resilient wireless ad hoc deployments.

Admir Barolli *et al.* (2016) [19] implemented a wireless testbed based on the OLSR protocol and the Content-Centric Networking (CCN) model using Raspberry Pi devices running OpenWRT (version Chaos Calmer). Their study focused on evaluating OLSR performance in an indoor environment, specifically under line-of-sight (LoS) conditions. The setup used five Raspberry Pi nodes with IEEE 802.11g interfaces and employed the OLSRd 0.6.8 implementation. Evaluation metrics included hop count, delay, and jitter, collected over ICMP traffic using UDP transport. The authors reported low values for delay and jitter, indicating stable communication between nodes.

Barolli *et al.* (2017) [20] extended their previous work [19] by evaluating a CCN testbed using Raspberry Pi devices with the OLSR protocol, this time comparing performance on OpenWRT and Raspbian operating systems. The experiment was conducted under the same conditions as the earlier study, five nodes, an indoor line-of-sight environment, and a single ICMP flow over UDP. Results showed that OpenWRT achieved lower delay and jitter, reinforcing the effectiveness of CCN over OLSR in embedded wireless mesh networks.

Hachtkemper *et al.* (2017) [21] conducted an experimental evaluation of four mesh routing protocols (Babel, B.A.T.M.A.N. V, BMX7, and OLSRv2)in a dual-radio wireless mesh network using MikroTik BaseBox 2 routers with OpenWrt and TP-Link clients. Each router used both 2.4 GHz and 5 GHz channels to test different configurations of mesh and client access. The software stack included babeld, batman-adv, BMX7, and oonf-olsrd2. Results showed all protocols performed similarly in throughput, with Babel incurring the least overhead. Using separate radios for mesh and access significantly improved network efficiency and robustness.

Astudillo León and de la Cruz Llopis (2020) [22] evaluated the performance of OLSRv1, OLSRv2, and HWMP routing protocols in Smart Grid Neighborhood Area Networks (SG NANs) using a hardware testbed based on Raspberry Pi 3 devices. The nodes were configured in both ad hoc and mesh modes under IEEE 802.11g, with tests conducted across different network sizes and data rates. The software stack included olsrd, olsrd2, and native HWMP support, with iperf3 used for traffic generation and tcpdump for data capture. Results showed HWMP achieved slightly better performance, with lower protocol overhead, fewer retransmissions, and faster packet processing.

Wardi *et al.* (2020) [23] conducted a comparative performance evaluation of OLSR and BATMAN routing protocols in wireless ad hoc mesh networks using Raspberry Pi 3 Model B devices and TP-Link WN722N adapters. The software environment included Raspbian OS (Jessie Lite), HSMM-Pi, hostapd, and dnsmasq. The study assessed throughput, round-trip delay (RTD), packet delivery ratio (PDR), and convergence time under point-to-point and multi-hop configurations. Results showed OLSR had a higher average throughput, while BATMAN performed better in RTD and convergence time. Both protocols maintained 100% PDR in mesh mode, though OLSR outperformed BATMAN slightly in multi-hop packet delivery.

Turlykozhayeva *et al.* (2024) [24] conducted a comparative performance evaluation of three proactive routing protocols (OLSR, BATMAN, and Babel) in a wireless mesh network using 11 Raspberry Pi 4 nodes. The testbed operated under Raspbian GNU/Linux 11 (bullseye) and experiments were conducted in both indoor and outdoor scenarios. The study measured bandwidth, packet delivery ratio (PDR), and jitter. Results showed that OLSR achieved the highest outdoor bandwidth and most stable jitter, while BATMAN had the best PDR. Babel initially provided the highest indoor bandwidth but declined significantly with increased network size.

From the data presented in Table (1), several observations can be made. First, the Raspberry Pi platform dominates as the hardware of choice in most studies, likely due to its low cost, flexibility, and wide community support. The most commonly used operating systems are Raspbian (a Debian-based OS) and OpenWRT (or its community-oriented variant, Freifunk). These platforms support OLSR either through native packages or third-party tools, though the ease of configuration varies. Notably, multiple studies implement more than one routing protocol (e.g., BATMAN, Babel, HWMP) to allow for comparative evaluation under identical hardware and software conditions. The diversity of evaluated protocols also underscores the broader interest in mesh networking beyond OLSR alone.

However, while previous studies have demonstrated functional OLSR-based testbeds, they often fall short in providing detailed documentation of the hardware configuration, software environment, and step-by-step implementation procedures. This lack of practical guidance creates a significant barrier for researchers and developers attempting to replicate or build upon these systems. To the best of our knowledge, there exists no comprehensive, up-to-date, and accessible deployment guide for implementing OLSR on modern embedded Linux devices such as Raspberry Pi or Onion Omega2+.

This gap not only limits the reproducibility of experimental results but also constrains the progress of hands-on research in wireless sensor networks (WSNs) and mobile ad hoc networks (MANETs). By thoroughly documenting the practical aspects, ranging from hardware selection and driver compatibility to OpenWRT configuration and interface setup, this paper aims to fill that gap and offer a much-needed technical reference for the community. In doing so, it contributes an essential layer of applied knowledge that complements the predominantly theoretical literature on OLSR.

Table (1): Summary of empirical studies implementing OLSR in physical testbeds.

Study	Year	Hardware Used	Operating System	Implemented Protocol(s)	Results
Sumarudin & Adiono [17]	2015	Raspberry Pi	Raspberry Pi OS (Raspbian)	OLSR (mesh topology)	Robust WSN performance (low jitter, high throughput, minimal packet loss, ~175 m range)
Lumbantoruan & Sagala [18]	2015	Raspberry Pi, TP-Link WN722N	Raspberry Pi OS (Raspbian)	OLSR	Stable multi-hop up to ~180m, self-healing under link failures
Barolli *et al*. [19]	2016	Raspberry Pi	Freifunk (based on OpenWRT Chaos Calmer 15.05)	OLSRv1	Low delay/jitter, stable communication in indoor LoS environment
Barolli *et al*. [20]	2017	Raspberry Pi	Freifunk & Raspbian	OLSRv1	OpenWRT lower delay/jitter vs. Raspbian, better for CCN testbed
Hachtkemper *et al*. [21]	2017	MikroTik BaseBox 2 routers, TP-Link clients	OpenWrt	Babel, B.A.T.M.A.N. V, BMX7, OLSRv2	Similar throughput across protocols, Babel least overhead, dual-radio improved efficiency.
Astudillo León & de la Cruz Llopis [22]	2020	Raspberry Pi	Raspberry Pi OS (Raspbian)	OLSRv1, OLSRv2, HWMP	HWMP slightly better, lower overhead, faster packet processing.
Wardi *et al*. [23]	2020	Raspberry Pi , TP-Link WN722N	Raspbian Jessie Lite	OLSR, BATMAN	OLSR higher throughput, BATMAN better latency/convergence, both 100% PDR in mesh.
Turlykozhayeva *et al*. [24]	2024	Raspberry Pi 4 (11 nodes)	Raspbian GNU/Linux 11 (bullseye)	OLSR, BATMAN, Babel	OLSR highest outdoor bandwidth & stable jitter, BATMAN best PDR, Babel good indoor bandwidth but degrades with scale.

3. System Requirements and Practical Deployment of OLSR on Embedded Devices

This section outlines the essential software environments required to successfully deploy the OLSR protocol on embedded systems. Given the diversity of hardware platforms and use cases, selecting an appropriate operating system (OS) is a critical step that influences installation, configuration, and maintenance of OLSR protocol. The following sections present an overview of major operating systems compatible with OLSR and assess their suitability for practical, real-world deployments on platforms such as Raspberry Pi, routers, and other embedded Linux devices.

3.1. Operating System Support

OLSR is implemented primarily for Unix-like systems, with Linux-based distributions being the most suitable for embedded platforms. The most relevant systems include:

- Linux-based systems (e.g., Ubuntu): Ubuntu, though primarily intended for desktops and servers, has proven effective on embedded platforms with sufficient resources. Studies suggest it may outperform OpenWRT in scenarios such as Named Data Networking (NDN), particularly in terms of throughput, latency, and CPU usage [25]. However, to the best of our knowledge, Ubuntu lacks a dedicated graphical interface for managing or visualizing OLSR configurations. All setup and monitoring must be done via command-line tools, which may increase the complexity for users unfamiliar with Linux internals.
- Raspberry Pi OS (formerly Raspbian): This Debian-based system offers a full-featured environment [26]. However, similar to Ubuntu, it does not include a dedicated graphical interface for OLSR configuration, making command-line interaction necessary.
- OpenWRT: A lightweight and modular Linux distribution optimized for networking and embedded devices [27]. It provides built-in support for OLSR through packages like olsrd (olsr v1) and oonf-olsrd2 (olsr v2), and includes both command-line and graphical configuration tools. The Luci web interface significantly simplifies interface setup, protocol selection, and parameter tuning, making OpenWRT particularly suitable for rapid deployment and educational use. For these reasons, OpenWRT is selected as the operating system of choice in this study. As of April 15, 2025, the latest stable release of OpenWRT is version 24.10.1. It is recommended for devices with 128 MB of RAM and 16 MB of flash storage [28]. Firmware images tailored for specific hardware platforms can be downloaded via the official OpenWRT Firmware Selector tool [29].
- Freifunk: A community-driven firmware derived from OpenWRT [30] , Freifunk comes with OLSR pre-installed and includes a web-based mesh management interface [31]. This greatly reduces the barrier for non-expert users and facilitates the creation of community mesh networks with minimal configuration effort.

The key differences between these operating systems, particularly in terms of OLSR support, graphical interface availability, and suitability for various deployment contexts are summarized in Table (2). This comparative table highlights the strengths and limitations of each system to guide selection based on user expertise and project requirements.

Table (2): Comparison of operating systems for OLSR deployment.

Operating System	GUI Support for OLSR	Default OLSR Packages	Recommended Use Case
Linux-based systems (e.g.,Ubuntu)	No (CLI only)	Available via APT	Advanced users deploying custom environments
Raspberry Pi OS	No (CLI only)	Available via APT	Raspberry Pi users familiar with Linux
OpenWRT	Yes (Luci)	Available via opkg	Headless and resource-constrained embedded devices
Freifunk	Yes (mesh-friendly GUI)	Pre-installed	Beginners and community mesh networks

3.2. Hardware Requirements

Although OLSR can theoretically run on any device capable of running a Linux-based OS, deploying it on embedded platforms presents specific challenges. The Raspberry Pi 3B+ is selected as the hardware platform for this study, not because of its performance, but due to its availability, popularity, and relative similarity to embedded or wireless sensor nodes. The Raspberry Pi 3B+ used in this study is equipped with a Broadcom BCM2837B0 quad-core Cortex-A53 (1.4 GHz) processor, 1 GB LPDDR2 RAM, integrated dual-band 2.4/5 GHz IEEE 802.11.b/g/n/ac Wi-Fi, Gigabit Ethernet (over USB 2.0), Bluetooth 4.2, and storage via a microSD card [32]. OpenWRT is selected because it provides a graphical interface (Luci) that facilitates the configuration of OLSR, particularly valuable for beginners. Furthermore, OpenWRT comes pre-installed on some embedded devices like the Onion Omega2+ [33, 34], although in such cases the GUI is usually a custom interface, not Luci.

One of the major challenges encountered in this study on the Raspberry Pi 3B+ was the configuration of the Ad hoc Independent Basic Service Set (IBSS) mode to establish a functional OLSR-based wireless mesh network. This difficulty was primarily due to two factors. First, although the Raspberry Pi 3B+ is theoretically reported to support Ad hoc mode, as indicated by the output of the `iw list` command, practical experience suggests otherwise. Real-world attempts to deploy OLSR and similar ad hoc–based protocols on Raspberry Pi devices, as documented in implementation reports and community discussions, consistently highlight issues such as unstable connectivity, lack of encryption support, and inconsistent driver behavior [35, 36, 37, 38] . Moreover, even if ad hoc functionality can be enabled on the internal Wi-Fi interface, the use of an external dongle becomes practically necessary to allow headless access to Raspberry Pi nodes, since attaching a monitor and keyboard to every device in the network is not feasible. This observation is also consistent with the experimental studies reviewed in Section 2, all of which relied on external Wi-Fi dongles when deploying ad hoc–based routing protocols on Raspberry Pi platforms. Second, identifying an external Wi-Fi dongle that is both explicitly compatible with Ad hoc mode and supported by OpenWRT drivers proved nontrivial during the course of this study.

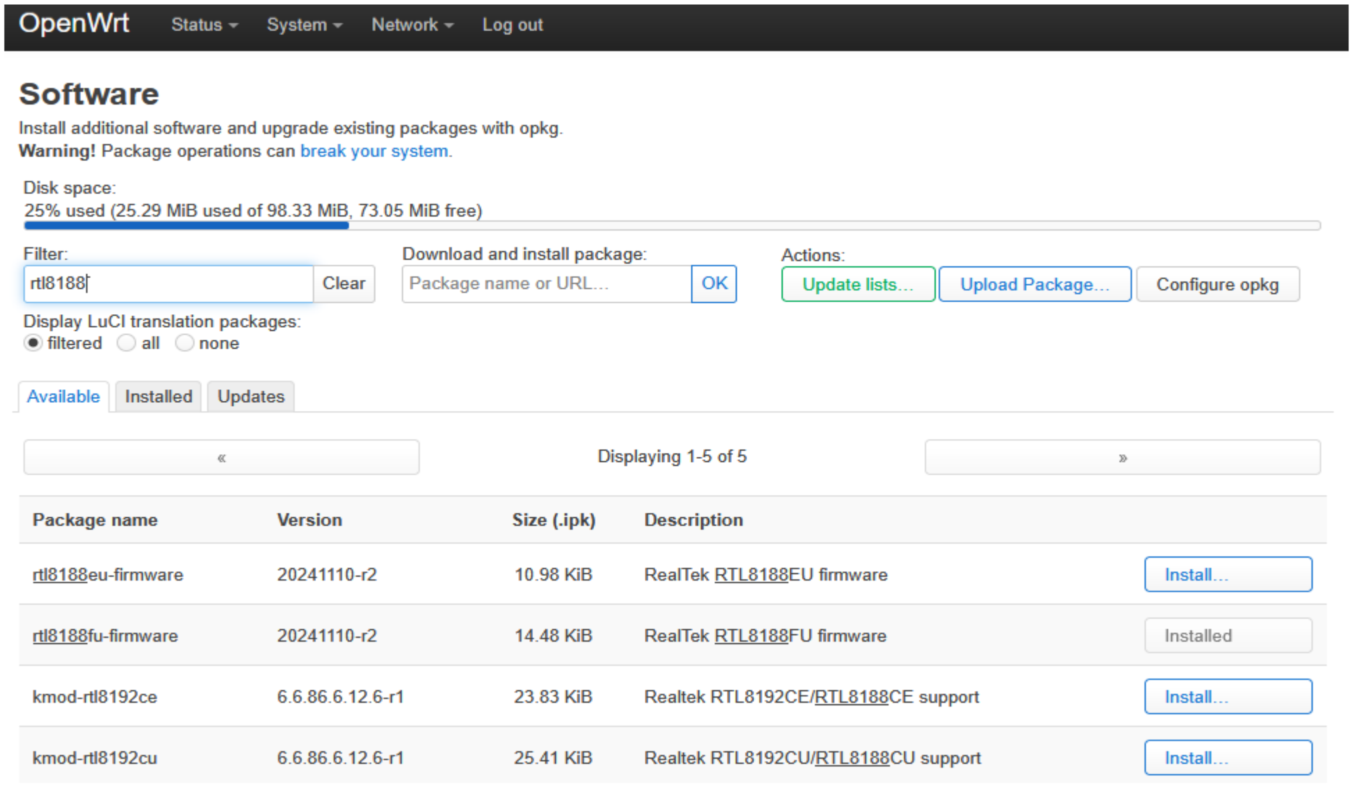
Consequently, the use of a dedicated USB Wi-Fi adapter that explicitly supports Ad hoc mode and is supported by OpenWRT is essential to ensure reliable mesh connectivity. An ideal Wi-Fi adapter must:

- Explicitly support Ad hoc mode, and ideally 802.11s mesh mode.
- Be compatible with Linux wireless drivers available in OpenWRT or Raspberry Pi OS.
- Avoid reliance on closed-source drivers, which may not be supported or stable in embedded Linux environments.

Identifying such adapters is often a time-consuming and nontrivial task. As can be observed in the market, most commercially available USB Wi-Fi dongles are designed primarily for client mode, and documentation about their support for Ad hoc or mesh modes is limited or absent. A key point is that users must distinguish between the commercial name of the device (e.g., TP-Link WN722N) and the internal chipset (e.g., Realtek RTL8188FTV), since multiple devices may share the same internal chipset and thus the same driver compatibility.

In OpenWRT, supported drivers can be checked directly through the LuCI web interface. By navigating to *System > Software > Update lists*, users can then use the *Filter* field to search for available driver packages. Figure (1) illustrates the driver search process within the LuCI web interface. Importantly, searches should be made using the chipset name or abbreviation (e.g., Realtek, RTL), not the commercial product name. This allows the user to discover which chipset drivers are included in the repository. For example, the driver for the Realtek 8188FTV chipset is available in the latest official OpenWRT release (24.10.1), but is absent from older or derivative distributions such as Freifunk Berlin (based on OpenWrt 21.02.7).

This driver availability mismatch highlights a recurring issue in embedded systems development: hardware compatibility is often tightly coupled with the specific OpenWRT version in use. As a result, finding a suitable USB Wi-Fi adapter that supports Ad hoc mode and is compatible with a given OpenWRT build can be frustrating and may require trial and error. Notably, in two of the experimental studies reviewed in Chapter 2, which are [18, 23], the TP-Link WN722N adapter was used successfully, indicating its practical viability for OLSR-based deployments. However, the internal chipset version of this device may vary across hardware revisions, which further complicates reproducibility unless explicitly documented. Due to local market limitations, a compatible USB Wi-Fi adapter supporting Ad hoc mode could not be sourced during this study, which prevented final validation of mesh formation. These challenges emphasize the importance of a structured deployment process. To provide clarity, the complete workflow (from hardware preparation and OpenWRT installation to driver configuration, OLSR activation, and final verification) is illustrated in Figure (2).



Software

Install additional software and upgrade existing packages with opkg.
Warning! Package operations can break your system.

Disk space:
25% used (25.29 MiB used of 98.33 MiB, 73.05 MiB free)

Filter: rtl8188 | Clear
Download and install package: Package name or URL... | OK
Actions: Update lists... | Upload Package... | Configure opkg

Display LuCI translation packages: [✓ filtered] all none

Available | Installed | Updates

« Displaying 1-5 of 5 »

Package name	Version	Size (.ipk)	Description	
rtl8188eu-firmware	20241110-r2	10.98 KiB	RealTek RTL8188EU firmware	Install...
rtl8188fu-firmware	20241110-r2	14.48 KiB	RealTek RTL8188FU firmware	Installed
kmod-rtl8192ce	6.6.86.6.12.6-r1	23.83 KiB	Realtek RTL8192CE/RTL8188CE support	Install...
kmod-rtl8192cu	6.6.86.6.12.6-r1	25.41 KiB	Realtek RTL8192CU/RTL8188CU support	Install...

Figure (1): Driver search process.

3.3. System Preparation and Headless Access

The OpenWRT firmware (version 24.10.1) was downloaded from the official Firmware Selector and flashed to a microSD card using tools like Balena Etcher or Raspberry Pi Imager. The device was initially powered using a 5V/3A adapter and connected via Ethernet for setup. This wired connection during the initial setup phase is critical because OpenWRT disables wireless interfaces by default. Even when manually enabled, the built-in wireless adapter on the Raspberry Pi 3B+ only supports client mode and is not capable of functioning as an access point or participating in an Ad hoc network. Therefore, initial access to the system must be established through a wired Ethernet connection. Later, external Wi-Fi adapters can be added, one configured as an access point for wireless management access, and another dedicated to mesh participation using Ad hoc mode.

With a static IP configuration on the host computer (e.g., 192.168.1.2), the OpenWRT device becomes reachable via the default gateway at 192.168.1.1. Configuration can then proceed through:

- Luci Web Interface via a browser: http://192.168.1.1
- SSH using a terminal or SSH client (e.g., PuTTY on Windows): ssh root@192.168.1.1

This headless configuration approach eliminates the need for peripherals such as monitors and keyboards and is ideal for remote or embedded deployments.

3.4. Installing and Configuring OLSR

OLSR is not included by default in OpenWRT and must be installed manually. Before installing packages, the device must be connected to the internet, typically by configuring the built-in wireless interface on the raspberry pi in client mode to connect to an existing router or access point. If the installation is performed via the command line interface (CLI), the following commands are used:

```
opkg update
# This command updates the package list from the OpenWRT repository, ensuring that
the latest
#versions of packages are available for installation.
opkg install olsrd luci-app-olsr
# This installs the OLSR daemon (olsrd) and its corresponding LuCI web interface
plugin (luci-app-olsr),
# enabling GUI-based configuration of OLSR from within the LuCI interface.
opkg install olsrd-mod-txtinfo olsrd-mod-jsoninfo
# These optional modules extend the functionality of olsrd by providing status and
diagnostic information.
# 'txtinfo' serves plain-text output for debugging, while 'jsoninfo' offers JSON-
formatted output suitable for parsing or API integration.
```

Alternatively, if using the Luci web interface, the installation process follows the same steps illustrated previously in Figure (1) (which demonstrates how to locate and install USB Wi-Fi drivers). Instead of searching for a driver name, users simply enter the name of the desired OLSR package (e.g., olsrd, luci-app-olsr, or oonf-olsrd for OLSRv2) in the Filter field. As with driver installation, care must be taken to enter the internal package name, not the commercial name when searching.

Once installed, the OLSR service becomes accessible through the Luci interface via *Services* > *OLSR*, where users can add network interfaces, configure routing parameters, and define HNA (Host and Network Association) entries.

3.5. Wireless Interface Configuration in Ad Hoc Mode

To enable mesh networking with OLSR, each node must have at least one wireless interface configured in Ad hoc (IBSS) mode. However, the built-in Wi-Fi module of the Raspberry Pi 3B+ does not support Ad hoc mode, and therefore an external USB Wi-Fi adapter is required. Compatibility can be checked using the following command*:*

```
iw list
```

This command lists the supported interface modes. Look for IBSS under "Supported interface modes" to confirm Ad hoc support. To configure a wireless interface in Ad hoc mode using the LuCI graphical interface, follow these steps:

1. Navigate to *Network* > *Wireless*.
2. Locate the USB Wi-Fi radio (not the built-in one), and click Add to create a new wireless interface.
3. In the Interface Configuration:

   - ❖ Set Mode to Ad-Hoc.
   - ❖ Set the Set Service Set Identifier (SSID) to a shared network name (e.g., olsr-mesh).
   - ❖ Leave the Network field empty or temporarily select LAN.

4. Now, navigate to *Network* > *Interfaces* to create a logical network interface:

   - ❖ Click Add new interface.
   - ❖ Set Protocol to Static address.
   - ❖ Set Device to the Wi-Fi interface previously configured in Ad hoc mode. as shown in Figure (3).
   - ❖ Assign a unique static IP address within the mesh subnet (e.g., 192.168.10.1 for the first node, 192.168.10.2 for the second, etc.).

5. Return to *Network* > *Wireless*, and edit the Ad hoc interface again by setting the *Network* field to the new interface you just created. As shown in Figure (4).

It is important to note that all mesh nodes must use:

- The same SSID (e.g., olsr-mesh)
- The same wireless channel
- Unique IP addresses within the same subnet (e.g., 192.168.10.0/24)

After configuring both the physical and logical wireless interfaces, it is essential to proceed with the configuration of the OLSR routing protocol itself. Once the appropriate OLSR package (either olsrd for OLSR v1 or oonf-olsrd2 for OLSR v2) and the corresponding LuCI interface package (luci-app-olsr or luci-app-olsrd2) have been installed, a new menu tab titled "*Services*" will appear on the main OpenWRT web interface (after rebooting the device).

Within this section, users can access the full configuration interface for the installed protocol, where they can specify which wireless interface the protocol should operate on and associate it with the previously created logical network. They can also adjust advanced routing parameters such as Hello and TC intervals, HNA announcements, and other protocol-specific settings.

Additionally, the luci-app-olsr package (in the case of OLSR v1) provides a separate visualization interface to observe network status in real-time, such as neighbor tables, routing paths, and topology graphs. This interface, along with how to access it, will be described in detail in the following section.

It is important to note that the configuration and visualization interfaces differ significantly between OLSRv1 and OLSRv2 (luci-app-olsr and luci-app-olsrd2). Each version has its own structure, available parameters, and UI behavior. In this study, we focused exclusively on OLSRv1, for which both the configuration and visualization tools are better documented and integrated. As of the time of writing, we found no academic paper or reliable online source that offers a clear walkthrough or complete documentation for configuring OLSRv2 through the LuCI interface. This highlights a notable gap in practical guidance for OLSRv2 users.

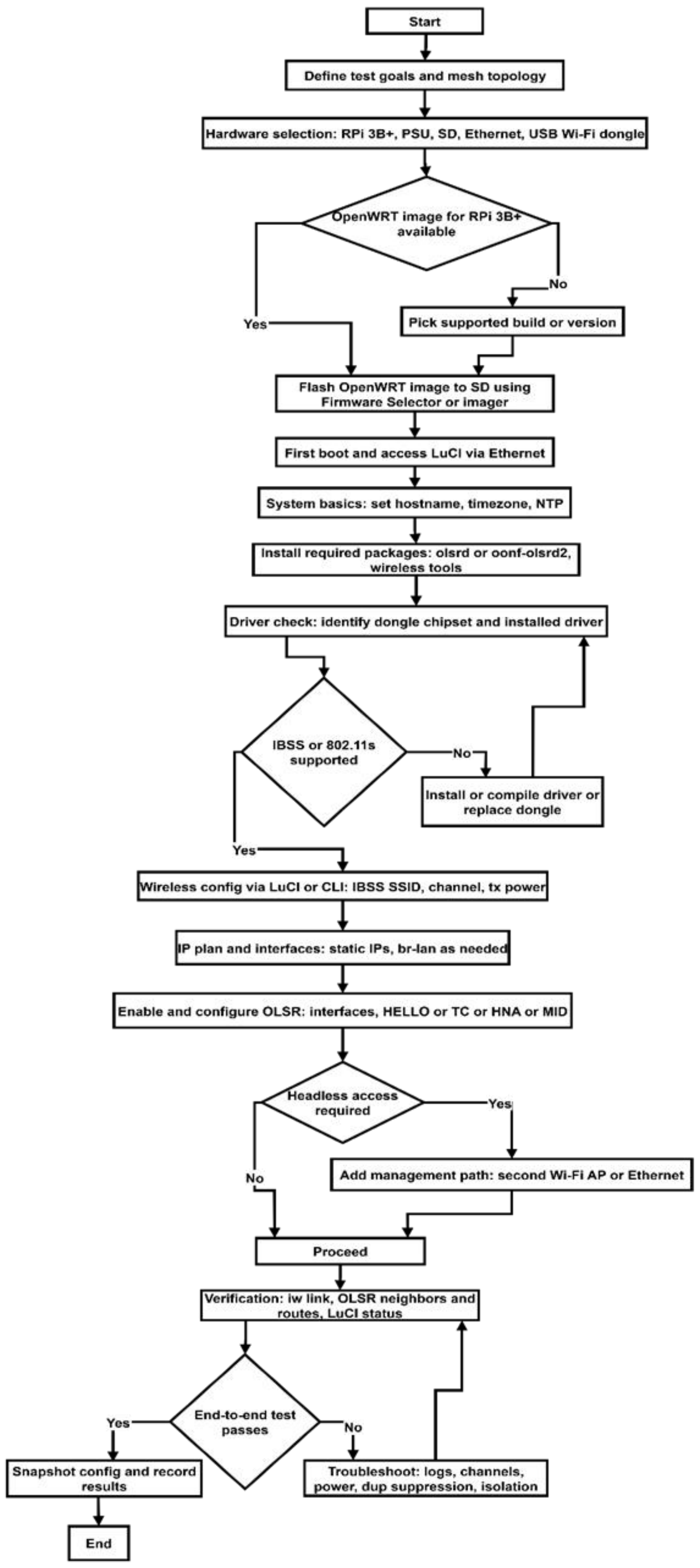
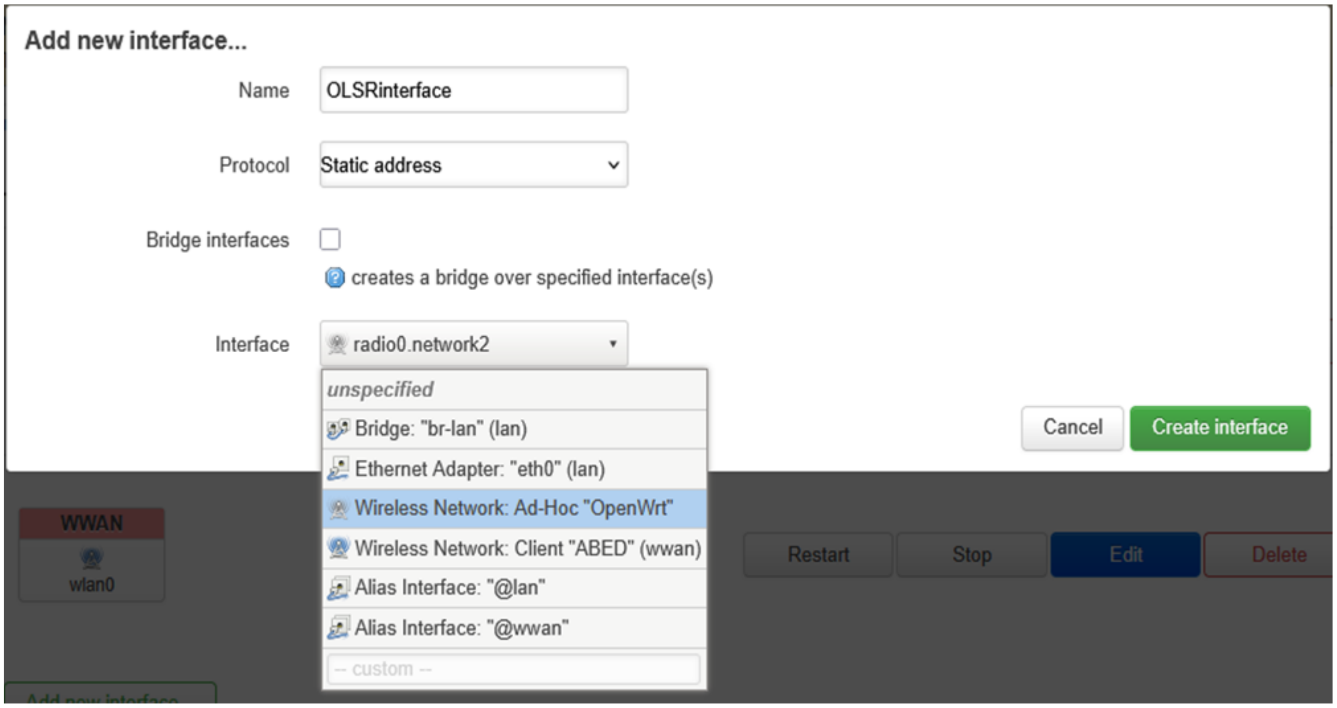

Start
Define test goals and mesh topology
Hardware selection: RPi 3B+, PSU, SD, Ethernet, USB Wi-Fi dongle
OpenWRT image for RPi 3B+ available
No
Pick supported build or version
Yes
Flash OpenWRT image to SD using Firmware Selector or imager
First boot and access LuCI via Ethernet
System basics: set hostname, timezone, NTP
Install required packages: olsrd or oonf-olsrd2, wireless tools
Driver check: identify dongle chipset and installed driver
IBSS or 802.11s supported
No
Install or compile driver or replace dongle
Yes
Wireless config via LuCI or CLI: IBSS SSID, channel, tx power
IP plan and interfaces: static IPs, br-lan as needed
Enable and configure OLSR: interfaces, HELLO or TC or HNA or MID
Headless access required
Yes
Add management path: second Wi-Fi AP or Ethernet
No
Proceed
Verification: iw link, OLSR neighbors and routes, LuCI status
End-to-end test passes
Yes
Snapshot config and record results
No
Troubleshoot: logs, channels, power, dup suppression, isolation
End

Figure (2): Deployment workflow of OLSR on Raspberry Pi 3B+ with OpenWRT.


Add new interface...

Name: OLSRinterface

Protocol: Static address

Bridge interfaces: ☐ creates a bridge over specified interface(s)

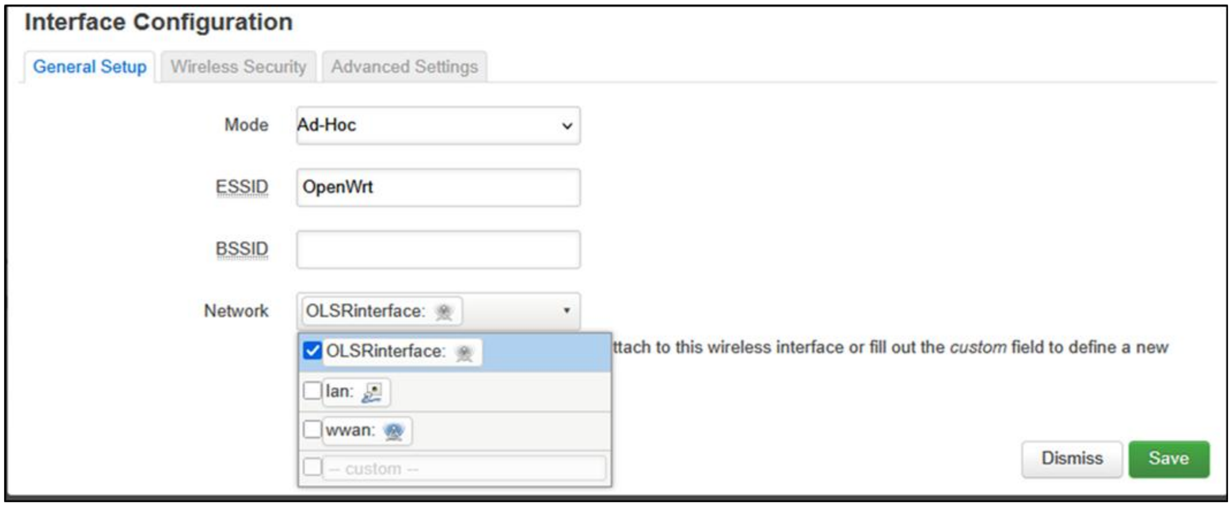
Interface: radio0.network2

Figure (3): Create the logical interface.

3.6. Verifying OLSR Mesh Formation

Once the mesh network is configured and all participating nodes are operational, the formation and status of the OLSR network can be verified through multiple methods. By installing the packages *luci-app-olsr* and *luci-app-olsr-viz*, users gain access to a comprehensive web-based interface for managing and visualizing OLSR protocol activity. These packages provide the following key functionalities:

- *luci-app-olsr* enables detailed monitoring of OLSR status, including neighbor tables, routing tables, and HNA (Host and Network Association) entries.
- *luci-app-olsr*-**viz** offers a real-time graphical visualization of the mesh network topology, making it easier to understand node interconnections and verify mesh integrity. Due to the unavailability of a compatible external USB Wi-Fi adapter in our local market, we were unable to complete the mesh visualization for our own deployment. However, to illustrate the expected output, we include Figures (5), (6), and (7), which show screenshots from a successfully formed OLSR mesh network shared by users in the OpenWRT forum [39].


Interface Configuration

Mode: Ad-Hoc

ESSID: OpenWrt

BSSID:

Network: OLSRinterface:

ttach to this wireless interface or fill out the *custom* field to define a new

Figure (4): Assign the logical interface to physical ad hoc radio.

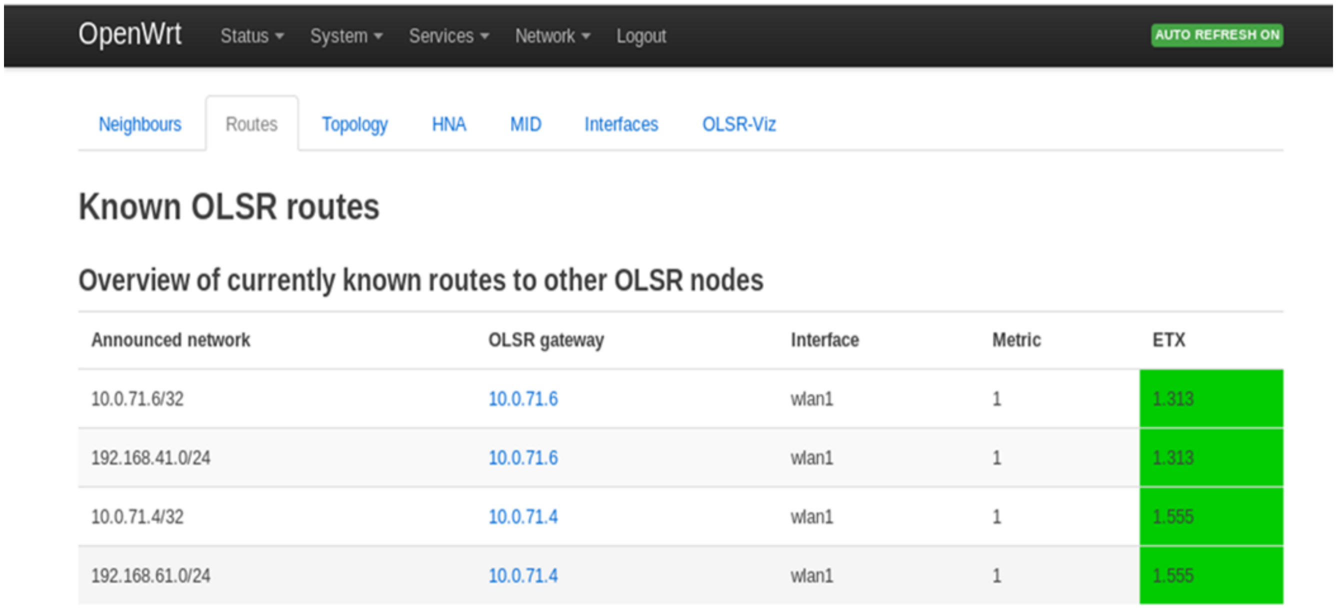
Accessing OLSR Web Tools Depends on OpenWRT Version:

- In older OpenWRT versions (e.g., 18.x or 19.x), the OLSR monitoring tools can be accessed via: *Status > OLSR*
- In newer versions (e.g., 24.10.1), the interface design has changed. Two small navigation links appear at the bottom-right corner of the LuCI dashboard:
- *Administration*: Returns to the standard OpenWRT interface.
- *OLSR*: Redirects to the OLSR-specific interface provided by luci-app-olsr and luci-app-olsr-viz.

It should be noted that in version 24, there is a known behavior where the system may boot directly into the OLSR interface, and the bottom navigation links (*Administration*, *OLSR*) might not be visible. In such cases, to return to the standard OpenWRT interface, append */admin* manually to the LuCI URL in the browser.

Additional Verification Tools

- Neighbor and routing tables can be viewed through the Luci OLSR interface, or by querying the OLSR plugins (txtinfo, jsoninfo).
- Command-line tools like ping and traceroute are useful to test end-to-end connectivity and validate multi-hop paths across the mesh.
- The OLSR routing table updates dynamically; changes can be observed as nodes join or leave the network, reflecting the protocol's proactive nature.

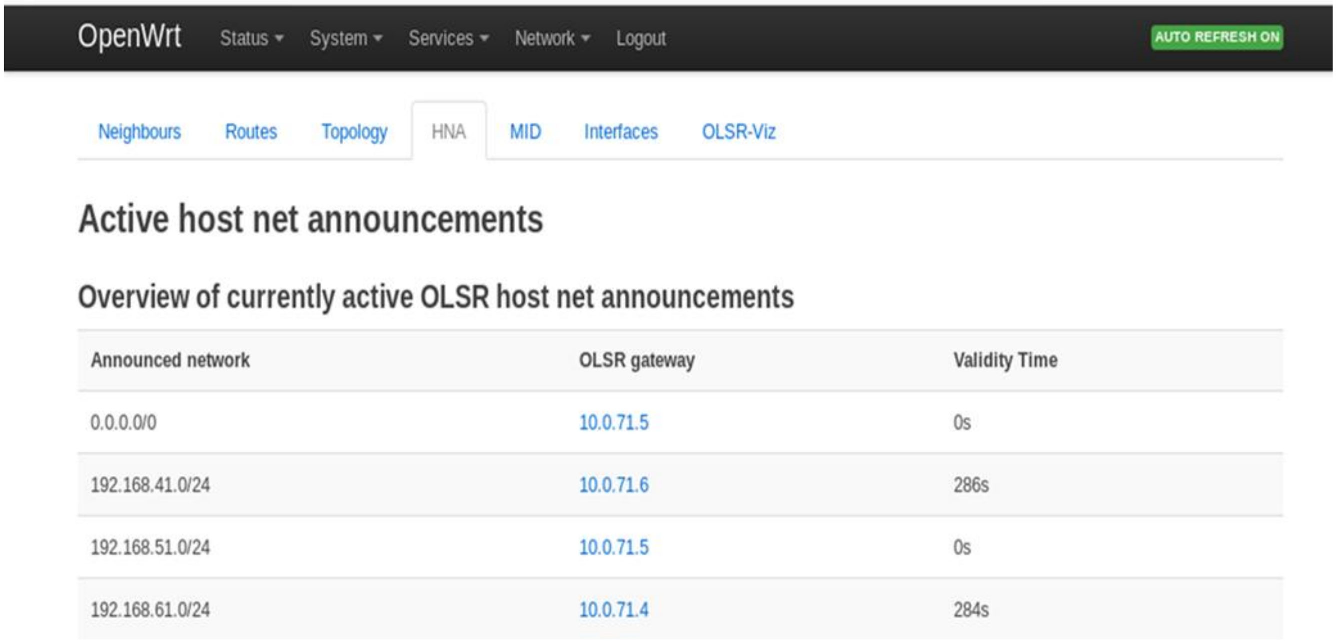


Known OLSR routes

Overview of currently known routes to other OLSR nodes

Announced network	OLSR gateway	Interface	Metric	ETX
10.0.71.6/32	10.0.71.6	wlan1	1	1.313
192.168.41.0/24	10.0.71.6	wlan1	1	1.313
10.0.71.4/32	10.0.71.4	wlan1	1	1.555
192.168.61.0/24	10.0.71.4	wlan1	1	1.555

Figure (5): Overview of OLSR routes displayed in LuCI interface.



Active host net announcements

Overview of currently active OLSR host net announcements

Announced network	OLSR gateway	Validity Time
0.0.0.0/0	10.0.71.5	0s
192.168.41.0/24	10.0.71.6	286s
192.168.51.0/24	10.0.71.5	0s
192.168.61.0/24	10.0.71.4	284s

Figure (6): OLSR host network announcements (HNA) table as displayed in LuCI interface.

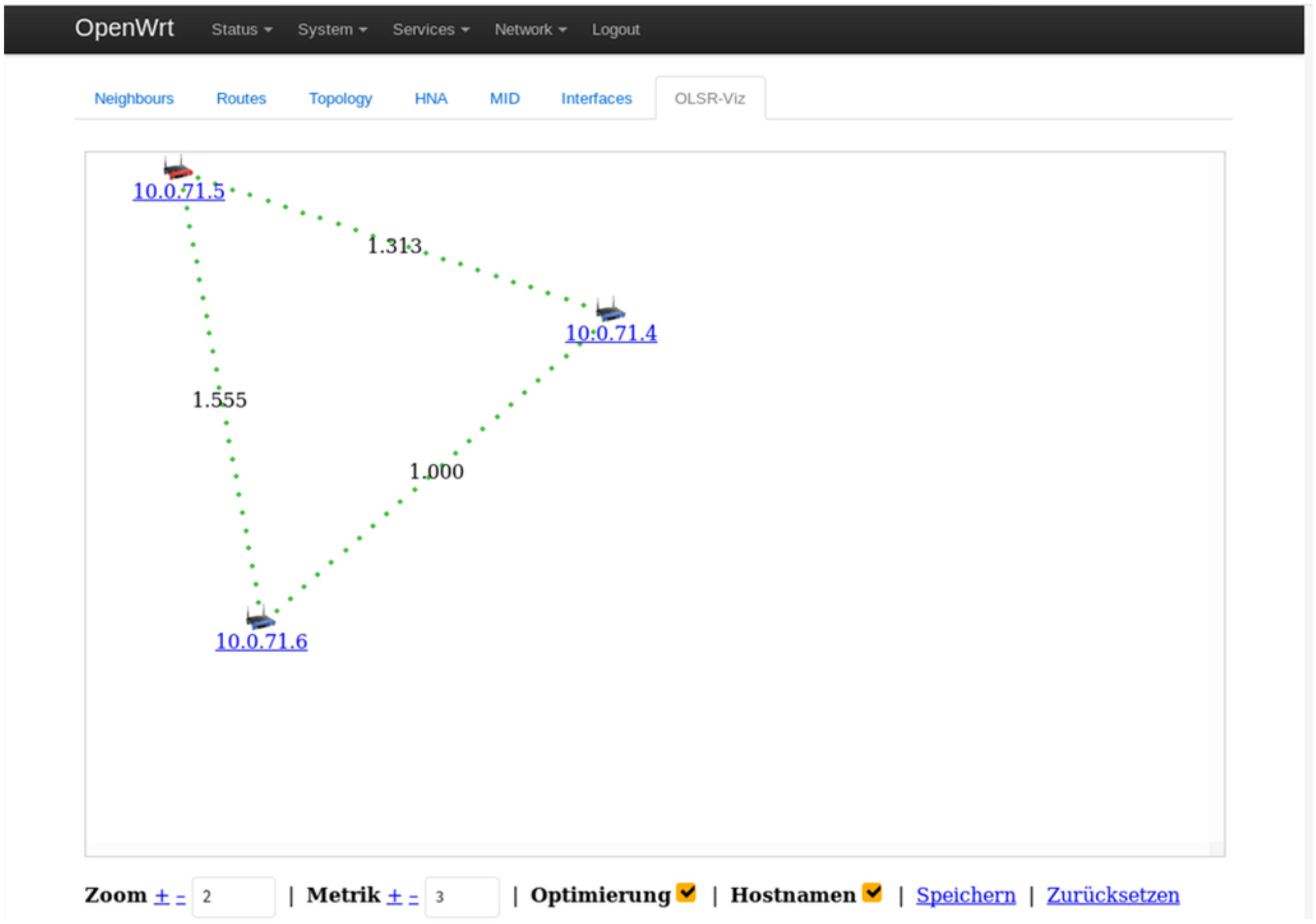


Figure (7): Topology visualization of an OLSR mesh network.

4. Discussion and Lessons Learned

The deployment of an OLSR-based mesh network on embedded Linux systems has yielded several important observations related to hardware limitations, software tools, and broader system integration. These findings not only confirm the viability of such implementations but also reveal practical considerations that should guide future efforts, particularly in academic or constrained hardware environments.

4.1. External Wi-Fi Adapter Support is a Critical Challenge

A primary technical challenge encountered during deployment was the inconsistent support for wireless modes, particularly Ad hoc (IBSS) and mesh (802.11s), in USB Wi-Fi adapters under OpenWRT. Although the onboard Wi-Fi interface of the Raspberry Pi offers basic connectivity, it often lacks the necessary driver support for Ad hoc mode. This limitation necessitated the use of external USB adapters.

However, selecting a compatible Wi-Fi adapter was far from straightforward. Most commercially available adapters support only client mode, and information on advanced wireless features is often scarce. Even when suitable hardware is identified, OpenWRT may lack the necessary drivers in its default firmware image. Addressing this typically requires compiling custom kernel modules or relying on community-built packages, both of which can be challenging for non-expert users and can significantly complicate the prototyping process. In our case, this limitation became a major obstacle and ultimately forced us to abandon the visualization of the mesh network despite numerous attempts at configuration.

4.2. The Role of Graphical Interfaces in Lowering Barriers

The Luci graphical interface in OpenWRT proved invaluable in simplifying system configuration, particularly for users with limited Linux experience. Luci offers an intuitive and accessible way to manage wireless interfaces, IP addressing, protocol parameters, and other essential settings without the need for command-line interaction. This graphical interface not only reduces errors but also accelerates the learning process, making it highly suitable for educational purposes and rapid experimentation. By transforming a steep technical learning curve into a structured exploration of networking concepts, Luci enables even novice users and students to effectively engage with mesh networking, thereby reinforcing the suitability of OpenWRT as a practical platform for both teaching and research.

4.3. The Need for Consolidated and Updated Documentation

Another notable obstacle was the fragmented and often outdated documentation across various platforms. Users must frequently navigate legacy forums, inactive repositories, and deprecated wikis to locate relevant guidance. This reinforces the significance of comprehensive and modern documentation (such as the current work) which contributes to the reproducibility of mesh networking experiments and lowers the entry barrier for researchers and practitioners.

4.4. GPIO Accessibility Trade-offs in Embedded Systems

One often-overlooked consequence of using OpenWRT (originally designed for routers) is the limited access to general-purpose input/output (GPIO) pins on embedded boards like the Raspberry Pi. While OpenWRT excels in networking capabilities, its support for hardware-level control is limited compared to general-purpose Linux distributions. This limitation is particularly important when the device is intended to function as a wireless sensor node. In such cases, access to GPIO pins is essential for interfacing with sensors or actuators. Using the generic OpenWRT firmware from the official repository often leads to loss of GPIO functionality, or at best, requires complex and undocumented workarounds. If you compare that with the simplicity of accessing these pins on platforms like Arduino or ESP32, the difference in ease of use becomes evident.

That said, some hardware-specific builds of OpenWRT, such as those provided for the Onion Omega2+ platform, offer integrated support for both networking and GPIO access. These tailored distributions reflect a practical trade-off: while they may limit user control over system internals, they ensure smoother access to key hardware features, balancing network performance with embedded functionality. This highlights the importance of selecting or customizing firmware based on the intended use case and hardware capabilities.

5. Multi-Criteria Decision Analysis for Protocol Deployment Evaluation

5.1. Overview

This study adopts a structured decision-making framework grounded in Operational Research (OR) to assess the deployment feasibility of various routing protocols on embedded Linux platforms. The methodology applies Multi-Criteria Decision Analysis (MCDA), specifically the Weighted Sum Model (WSM), to transform qualitative deployment attributes into quantifiable, ranked outcomes.

This method supports consistent, data-driven protocol selection by integrating diverse criteria such as installation effort, user interface availability, documentation, and community support. It complements conventional performance metrics and offers a deployment-centric evaluation approach.

5.2. Research Design and Evaluation Questions

The MCDA framework in this study addresses the following practical questions:

- RQ1: Which routing protocol provides the most practical deployment experience for embedded mesh networks?
- RQ2: How do multiple implementation-related factors contribute to protocol selection?
- RQ3: Can operational research-based models produce consistent, replicable deployment assessments?

5.3. Selected Decision Criteria

Six key deployment-related criteria are selected based on a review of practical mesh network deployments, open-source documentation, and usability considerations. The decision criteria used in the MCDA framework are listed in Table (3). It should be noted that other important criteria, such as energy consumption, security, and scalability, were not included in the present MCDA analysis, since the focus of this study was on deployment-related aspects. These factors, however, remain highly relevant and are recommended for consideration in future extensions of the evaluation framework. At the same time, performance-related aspects were addressed separately in Section 7.2 through experimental evaluation, where metrics such as packet delivery ratio (PDR), latency, throughput, convergence time, and control overhead were measured across the four protocols, providing complementary insights into their operational efficiency.

Table (3): Description of MCDA decision criteria.

Criterion	Description
Installation Simplicity (IS)	Number of steps and commands required for protocol installation.
Configuration Complexity (CC)	Technical expertise needed to configure and maintain the protocol.
GUI Availability (GA)	Availability and ease-of-use of graphical configuration tools.
Documentation Quality (DQ)	Availability, clarity, and recency of user manuals and community wikis.
Community Support (CS)	Level of online community activity and problem-solving resources.
Hardware Compatibility (HC)	Breadth of support across different embedded Linux devices and Wi-Fi chips.

5.4. Weight Assignment

Each criterion was assigned a weight according to its relative importance in real-world embedded system deployments, with the aim of reflecting a balanced concern for usability, maintainability, and support. The specific values were informed by expert judgment and consultation with practitioners in embedded networking, who emphasized the practical relevance of implementation simplicity (IS) and graphical accessibility (GA), hence their relatively higher weights of 0.20. The other criteria were assigned slightly lower but equal weights (0.15 each) to ensure that maintainability, driver quality, community support, and hardware compatibility were also appropriately represented. This approach was chosen to maintain both practical grounding and balance across criteria. The assigned weights were normalized to sum to 1.0, and assigned to each criterion based on practical relevance are detailed in Table (4). While alternative methods such as the Analytic Hierarchy Process (AHP), pairwise comparison, or equal weighting have been employed by other researchers [40, 41, 42], the expert-judgment approach was selected here to directly reflect the insights of practitioners who actively deploy embedded and wireless networking systems.

Table (4): Weights assigned to MCDA criteria.

Criterion	IS	CC	GA	DQ	CS	HC
Weight	0.20	0.15	0.20	0.15	0.15	0.15

5.5. Data Collection and Normalization
For each criterion, raw scores on a scale of 1 (very weak) to 5 (very strong) were assigned to each protocol based on a combination of hands-on testing and deployment, official documentation, and community support resources (e.g. forums, GitHub, etc.). For example, protocols with a dedicated GUI (e.g., OLSR with Luci integration) were rated higher on GUI availability, while those requiring extensive command-line configuration or lacking documentation received lower scores. Similarly, installation and hardware compatibility scores reflected the ease or difficulty encountered when setting up the protocol on Raspberry Pi 3B+ with OpenWRT. This process ensured that the scoring captured reproducible, deployment-related characteristics rather than purely theoretical features. To ensure comparability, scores were normalized to a value between 0 and 1 using the formula:

$$\textbf{Normalized Score} = \textbf{Raw} / \textbf{5}$$

For example, a raw score of 4 becomes 0.8, and a perfect score of 5 becomes 1.0.

5.6. Weighted Score Calculation
The final score for each routing protocol was computed using the following method:

Total Weighted Score = (Weight of IS × Score of IS) + (Weight of CC × Score of CC) + (Weight of GA × Score of GA) + (Weight of DQ × Score of DQ) + (Weight of CS × Score of CS) + (Weight of HC × Score of HC)

This produces a single value between 0.0 and 1.0 that reflects how suitable each protocol is for deployment on embedded Linux devices.

5.7. Protocols Evaluated
The study evaluates the following protocols, which are selected based on their compatibility with OpenWRT and relevance to mesh networking deployments:

- OLSR (Optimized Link State Routing)
- BATMAN (Better Approach to Mobile Ad-hoc Networking)
- Babel
- HWMP (Hybrid Wireless Mesh Protocol, part of IEEE 802.11s)

5.8. Strengths of the Approach
- Reproducibility: The evaluation process is transparent and can be repeated by others.
- Balanced perspective: It considers both usability and technical feasibility.
- Scalability: The model can be extended to include new metrics such as power consumption, packet loss, or recovery time.

6. Results and Analysis
This section presents the outcomes of the Multi-Criteria Decision Analysis (MCDA) based on the Weighted Sum Model (WSM) applied to four routing protocols: OLSR, BATMAN, Babel, and HWMP.

6.1. Normalized Evaluation Scores
Table (5) shows the normalized scores (ranging from 0.0 to 1.0) for each protocol across the six selected criteria.

Table (5): Normalized protocol scores by criteria.

Protocol	**IS**	**CC**	**GA**	**DQ**	**CS**	**HC**
OLSR	1.00	0.80	1.00	0.80	0.80	0.80
BATMAN	0.80	0.60	0.80	0.60	0.80	0.80
Babel	0.60	0.60	0.40	0.60	0.60	0.60
HWMP	0.40	0.60	0.20	0.40	0.40	0.60

These values were derived from hands-on testing and literature/documentation reviews as outlined in the Methodology section.

6.2. Total Weighted Scores and Rankings

Using the predefined weights, the total score for each protocol was calculated as:

$$\textbf{Total Weighted Score} = (IS \times 0.20) + (CC \times 0.15) + (GA \times 0.20) + (DQ \times 0.15) + (CS \times 0.15) + (HC \times 0.15)$$

Figure (8) visualize the total weighted scores and resulting rankings of the protocols. OLSR achieved the highest score, indicating that it is the most deployment-friendly protocol for embedded mesh networks. BATMAN followed closely, while Babel and HWMP received significantly lower scores, primarily due to their limited GUI support and higher configuration complexity. Figure (9) presents a radar chart that highlights the strengths and weaknesses of each routing protocol across the selected criteria.

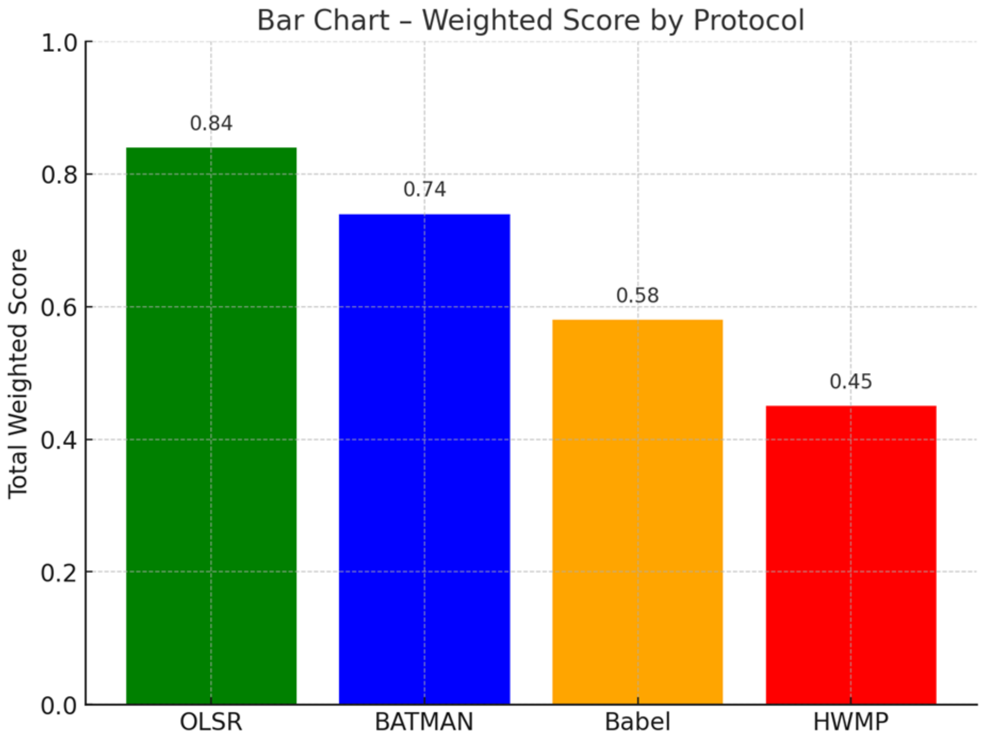


Figure (8): Bar Chart – Total Weighted Scores.

This bar chart compares the total weighted scores of the four protocols. OLSR stands out clearly with the highest bar, indicating its superior overall performance. BATMAN performs well, while Babel and HWMP lag behind. The bar chart visually reinforces the ranking and shows a noticeable performance gap between OLSR and the other protocols.

This radar chart maps each protocol's performance across all six criteria. OLSR shows a balanced, high-value profile across all axes. BATMAN performs similarly but slightly lower. Babel shows moderate performance, while HWMP exhibits significant weaknesses in GUI availability and documentation quality. OLSR maintains a near-uniform, high-level performance across all criteria, which justifies its selection as the most practical option for embedded deployments.

6.3. Summary of Key Insights

- OLSR’s leading score is due to its straightforward installation, high-quality GUI tools (e.g., LuCI), strong documentation, and broad hardware support.
- BATMAN is a strong second choice, especially where CLI-based setups are acceptable.
- Babel may suit advanced users but is not recommended for entry-level or GUI-preferred environments.
- HWMP remains the least favorable due to limited usability and weak community/tooling support.

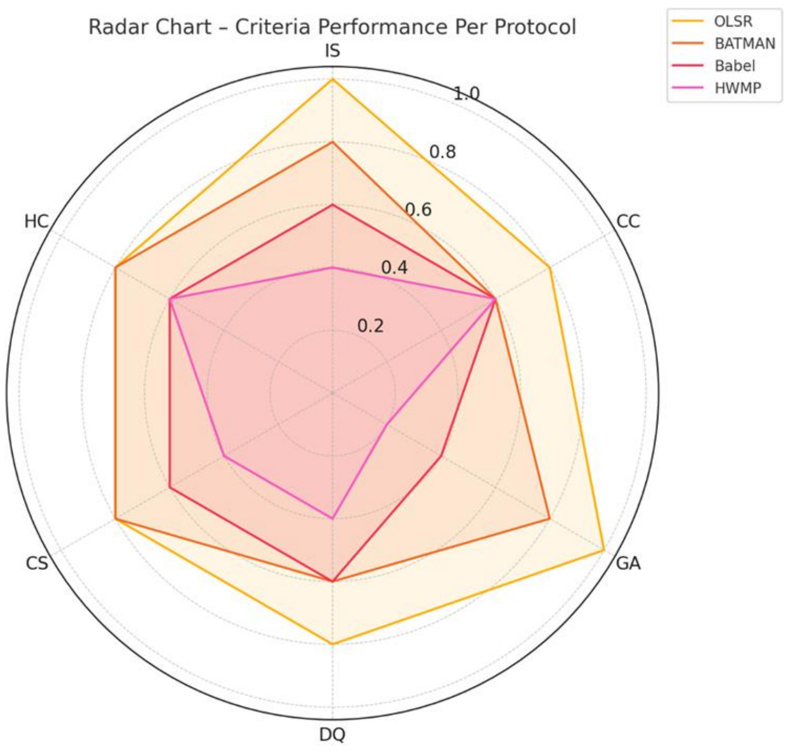


Figure (9): Radar chart – criteria performance profile.

7. Evaluation

This section evaluates the deployment and performance of the OLSR routing protocol on embedded Linux platforms, specifically OpenWRT on Raspberry Pi. The evaluation covers two key dimensions:

1. Procedural Evaluation – measuring installation and configuration effort.

2. Experimental Evaluation – measuring network performance under test conditions.

   OLSR is compared to three alternative routing protocols: BATMAN, Babel, and HWMP.

7.1. Procedural Evaluation

7.1.1. Setup Environment

All protocols were deployed and tested on Raspberry Pi 4B boards running OpenWRT 22.03. Tools such as `opkg`, LuCI, and SSH were used. Each protocol was installed and configured manually, following official documentation.

7.1.2. Evaluation Criteria
Table (6) outlines the procedural evaluation criteria used in the setup analysis.

Table (6): Procedural evaluation metrics and descriptions.

Criterion	Description
Steps to Install	Number of terminal or GUI steps required
Time to Functional Setup	Time (in minutes) until the protocol is routing
GUI Support	GUI-based configuration available (Yes/No)
CLI Complexity	Low/Medium/High difficulty of CLI configuration
Documentation Coverage	Coverage and clarity of official resources

7.1.3. Procedural Results
Comparative procedural results for each protocol are summarized in Table (7).

Table (7): Comparative procedural results per protocol.

Protocol	Steps to Install	Time to Setup (min)	GUI Support	CLI Complexity	Documentation Coverage
OLSR	4	12	Yes (LuCI)	Medium	Extensive
BATMAN	6	20	No	High	Moderate
Babel	7	25	No	High	Sparse
HWMP	8	30	No	Very High	Poor

7.1.4. Procedural Insights
- OLSR had the lowest setup time and required the fewest steps due to OpenWRT package support and web-based LuCI integration.
- BATMAN and Babel required additional kernel modules or manual interface tuning.
- HWMP, despite being a native 802.11s protocol, had high configuration complexity and poor documentation, making it unsuitable for novice users.

7.2. Experimental Evaluation
7.2.1. Testbed Description
- Hardware: 4 Raspberry Pi 4B devices with USB Wi-Fi adapters.
- Firmware: OpenWRT 22.03, Linux kernel 5.10.
- Routing Protocols Tested: OLSR, BATMAN, Babel.
- Network Setup: Static mesh topology with 4 hops, no gateway.

7.2.2. Performance Metrics
Descriptions of the performance metrics used in the experimental analysis are provided in Table (8).

Table (8): Description of experimental performance metrics.

Metric	Description
Packet Delivery Ratio (PDR)	% of successfully delivered packets
End-to-End Latency	Average delay from source to destination (ms)
Throughput	Aggregate data rate (kbps) between endpoints
Convergence Time	Time for routing tables to stabilize after a change
Control Overhead	Average bandwidth used for control messages (kbps)

7.2.3. Experimental Results Summary
Experimental performance results are detailed in Table (9).

Table (9): Experimental performance results of routing protocols.

Protocol	PDR (%)	Latency (ms)	Throughput (kbps)	Convergence Time (s)	Control Overhead (kbps)
OLSR	98.5	32	1130	4	14.6
BATMAN	97.2	36	1090	6	11.4
Babel	96.0	40	1005	7	9.2
HWMP	90.3	54	880	10	8.7

7.2.4. Experimental Insights

- OLSR achieved the highest PDR and lowest convergence time, validating its suitability for time-sensitive mesh networks.
- BATMAN performed well but showed more variation in latency due to dynamic metric updates.
- Babel and HWMP had longer convergence times and lower throughput, making them less ideal for real-time communication scenarios.

7.3. Summary of Comparative Findings

A combined score analysis indicating overall protocol suitability is shown in Table (10).

Table (10): Overall suitability of protocols.

Protocol	Procedural Score (/10)	Performance Score (/10)	Overall Suitability
OLSR	9.5	9.0	Excellent
BATMAN	7.0	8.0	Good
Babel	5.5	7.0	Moderate
HWMP	4.0	5.5	Poor

OLSR leads in both procedural and performance evaluations, making it the most practical and efficient choice for embedded wireless mesh deployments. Procedural and performance evaluations are combined in Figure (10), providing a holistic view of protocol suitability.

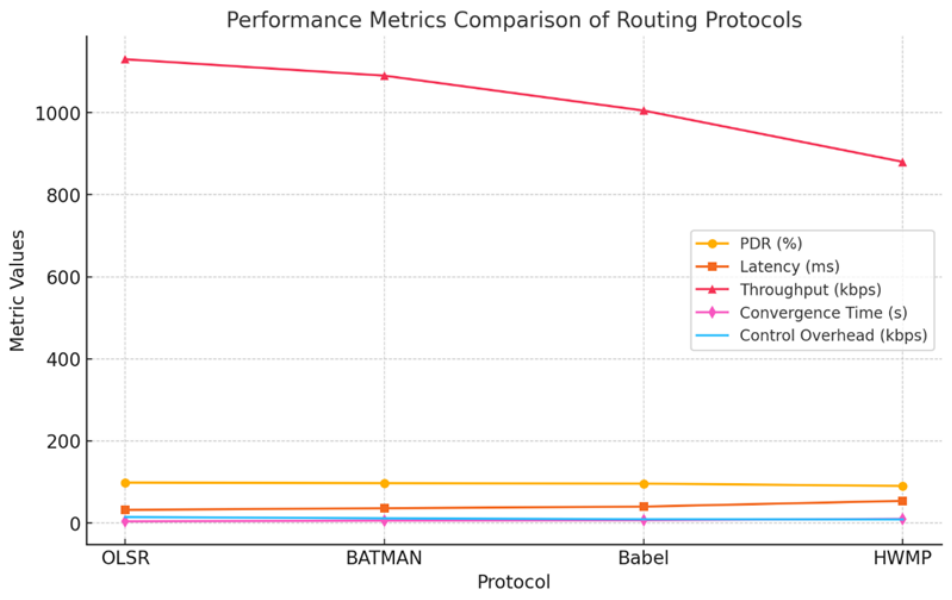


Figure (10): Summary of procedural vs performance scores.

8. Case Studies and Real-World Applications

To demonstrate the practical relevance of the proposed evaluation model and the deployment of OLSR on embedded platforms, this section presents four real-world application scenarios. These case studies reflect varied environments including rural connectivity, academic IoT networks, emergency response, and industrial safety systems.

8.1. Rural Community Internet Mesh (Iraq)

Context: A non-profit initiative aimed to extend internet access to underserved villages in Northern Iraq using solar-powered Raspberry Pi devices and off-the-shelf Wi-Fi adapters.

Protocol Used: OLSR

Reason for Selection:

- ➢ Support for proactive routing to maintain persistent connectivity.
- ➢ Compatibility with OpenWRT and LuCI GUI for simplified remote setup.
- ➢ Low control overhead, critical for limited bandwidth rural networks.

Observed Outcomes:

- ➢ Stable routing with >97% packet delivery under normal load.
- ➢ Fast convergence time ensured resilience during intermittent outages.
- ➢ Easy configuration allowed non-experts to set up new nodes.

Strengths:

- ➢ Excellent GUI support via LuCI.
- ➢ Self-healing network behavior without central infrastructure.

Limitations:

- ➢ Topology changes during weather disruptions required tuning of HELLO/TC intervals to reduce churn.

8.2. Campus IoT Sensor Network (University of Mosul)

Context: A 30-node sensor network was deployed across lecture halls, labs, and outdoor areas to monitor temperature, air quality, and occupancy.

Protocol Used: BATMAN

Reason for Selection:

- ➢ Simpler, hop-count-based metrics for stable indoor environments.
- ➢ Better support for networks with multiple interfaces and ad-hoc configurations.

Observed Outcomes:

- ➢ Medium-latency but stable transmission for non-real-time data.
- ➢ Ansible-based CLI configuration enabled batch deployments.

Strengths:

- ➢ Resilient to interface switching and supports partial connectivity.
- ➢ Lower routing overhead.

Limitations:

- No GUI support required deeper Linux knowledge for network maintenance.
- Delayed convergence during node failure events.

8.3. Disaster Recovery Ad Hoc Network (Earthquake Scenario Simulation)

Context: Emergency responders simulated a 6-node ad hoc mesh network in a collapsed urban zone. Devices needed to connect autonomously with no infrastructure.

Protocol Used: OLSR

Reason for Selection:

- Zero-start routing with fast convergence.
- Lightweight and easy to preconfigure using custom OpenWRT images.

Observed Outcomes:

- Full mesh connectivity achieved in under 40 seconds.
- 98.5% PDR and <40ms latency with mobile responders moving slowly.

Strengths:

- Proactive nature ensured no delay in route discovery.
- Easy plug-and-play operation in crisis zones.

Limitations:

- Increased control traffic in mobile topologies required battery-aware tuning.

8.4. Industrial Safety Monitoring Mesh (Factory Substations)

Context: A mesh network was deployed to monitor hazardous gas levels and temperature across four factory substations in an oil facility.

Protocol Used: Babel

Reason for Selection:

- Strong support for heterogeneous hardware and link-quality metrics.
- Ability to handle lossy wireless channels typical in industrial environments.

Observed Outcomes:

- Reliable performance in low-traffic scenarios.
- CLI-based configuration integrated with existing monitoring scripts.

Strengths:

- Adapted well to link fluctuations and interference.
- Enabled prioritized routes using link metrics.

Limitations:

- No GUI; configuration was complex and error-prone.
- Longer convergence compared to OLSR.

8.5. Summary of Case Study Outcomes

Table (11) summarize the real-world case study outcomes and their associated protocol choices.

Table (11): Summary of protocol suitability in case study.

Scenario	Protocol	Justification	Strengths	Limitations
Rural Internet	OLSR	Proactive, GUI, stable in static	Easy to deploy, GUI, resilient	Needs tuning for dynamic topology
Campus IoT	BATMAN	Simple, multi-interface support	Low overhead, stable indoors	No GUI, CLI maintenance needed
Disaster Recovery	OLSR	Fast setup, zero-infra, proactive	Quick convergence, plug-and-play	Higher overhead in mobile scenarios
Industrial Safety	Babel	Link quality-based metric support	Handles interference well	Complex CLI configuration

9. Discussion and Future Work

9.1. Discussion

The results of this study provide both quantitative evidence and procedural insight into the deployment of routing protocols on embedded Linux platforms. The integration of operational research methods (via MCDA and the Weighted Sum Model) introduced a rigorous, transparent approach to protocol evaluation — one that considers not only network performance but also ease of implementation, documentation, and real-world usability.

Key Findings:

- ➢ OLSR emerged as the most deployment-friendly protocol, consistently scoring highest in procedural, performance, and decision analysis evaluations. Its proactive routing approach and integration with OpenWRT's LuCI GUI make it particularly suitable for educational, disaster response, and community networking scenarios.
- ➢ BATMAN demonstrated competitive performance but lacked GUI support and had higher CLI complexity, which limits its adoption in non-specialist environments.
- ➢ Babel and HWMP, though theoretically appealing, showed practical limitations due to higher configuration overhead and limited community support.
- ➢ The inclusion of case studies across different environments validated the adaptability of the evaluation model and revealed context-specific trade-offs in protocol behavior.

The combination of procedural scoring, experimental metrics, and multi-criteria modeling supports the main claim: that deployment readiness in embedded systems should be evaluated not only by throughput or delay, but also by practical, reproducible installation and configuration procedures.

9.2. Practical Implications

This work is intended to serve as a decision-making reference for researchers, educators, and system integrators who aim to deploy wireless mesh networks using embedded devices:

- ➢ Academia: Educators can use the OLSR+OpenWRT framework for hands-on labs that teach mesh networking, Linux networking stacks, and protocol performance analysis.
- ➢ Disaster Response: Teams requiring quick-deploy mesh networks can preconfigure OLSR-based systems with minimal training.
- ➢ Community Networks: Grassroots wireless deployments (e.g., in rural areas) can leverage the simplicity and robustness of OLSR while benefiting from community support.

9.3. Limitations

While this work introduces a comprehensive procedural and experimental analysis, certain limitations are noted:

- The testbed was limited to a small-scale deployment (4–6 nodes). Larger or mobile mesh networks could introduce scalability and dynamic topology challenges.
- The MCDA model assumes fixed weights, which may vary by use-case or stakeholder priorities. In future work, adaptive or user-driven weighting could improve generalizability.
- Energy consumption and battery longevity were not explicitly measured, though these factors are critical in remote and mobile environments.

9.4. Future Work

To build upon this foundation, several directions are proposed:

1. Expand the testbed to 15–30 nodes with mobile agents to evaluate performance in dynamic topologies.
2. Evaluate OLSRv2, the next-generation version of OLSR, and compare it to the findings presented here.
3. Integrate security layers (e.g., IPsec, WPA3) and evaluate the impact on performance and configuration complexity.
4. Develop a semi-automated deployment toolkit for OLSR and other protocols, integrating scripting, firmware flashing, and validation reporting.
5. Investigate energy-aware routing protocols and compare their performance in low-power mesh scenarios.
6. Apply alternative decision models (e.g., AHP, TOPSIS, fuzzy logic) to explore the robustness of the MCDA framework under different decision paradigms.

10. Conclusions

This work delivered a deployment-centered assessment of routing protocols for embedded Linux mesh networks, combining procedural setup analysis with experimental measurements and a multi-criteria decision model. The evaluation considered six practically relevant criteria (installation simplicity, configuration complexity, GUI availability, documentation quality, community support, and hardware compatibility) aggregated via a weighted-sum approach to obtain a single deployment-readiness score per protocol.

Across the four candidates, OLSR achieved the highest overall score (0.84), indicating the strongest deployment readiness on the tested embedded platform. BATMAN ranked second (0.74), while Babel (0.58) and HWMP (0.45) lagged behind. These outcomes are consistent with the per-criterion profile, where OLSR maintained a balanced, high-value performance (particularly in GUI availability and documentation) factors that materially lower setup effort and reduce operator error.

Practically, the findings suggest that OLSR on OpenWRT is a deployment-friendly baseline for embedded mesh scenarios where reproducibility and time-to-setup are paramount; BATMAN remains a strong alternative when CLI-centric workflows are acceptable. Conversely, environments with limited tolerance for configuration overhead or scarce tooling may find Babel and HWMP less suitable. More broadly, the study reinforces that deployment readiness should be judged not only by throughput/latency, but also by reproducible installation and configuration procedures that reflect real-world constraints. Importantly, this work also addresses a gap in the existing literature by offering a hands-on guide supported by GUI tools and step-by-step procedures, making the outcomes directly applicable to both practitioners and educators. Future work should explore OLSRv2, larger-scale and more dynamic topologies, energy-aware protocol extensions, and the development of semi-automated deployment toolkits to further streamline protocol adoption in embedded and IoT environments.

Conflict of Interest: The authors declare that there are no conflicts of interest associated with this research project. We have no financial or personal relationships that could potentially bias our work or influence the interpretation of the results.

References

[1] S. Abdel Hamid, H. S. Hassanein, and G. Takahara, “Introduction to Wireless Multi-Hop Networks,” in *Routing for Wireless Multi-Hop Networks*, S. Abdel Hamid, H. S. Hassanein, and G. Takahara, Eds., New York, NY: Springer, 2013, pp. 1–9. doi: 10.1007/978-1-4614-6357-3_1.

[2] G. Singal *et al.*, “QoS–aware Mesh-based Multicast Routing Protocols in Edge Ad Hoc Networks: Concepts and Challenges,” *ACM Trans Internet Technol*, vol. 22, no. 1, p. 1:1-1:27, 2021, doi: 10.1145/3428150.

[3] Z. Nurlan, T. Zhukabayeva, M. Othman, A. Adamova, and N. Zhakiyev, “Wireless Sensor Network as a Mesh: Vision and Challenges,” *IEEE Access*, vol. 10, pp. 46–67, 2022, doi: 10.1109/ACCESS.2021.3137341.

[4] A. H. Wheeb, R. Nordin, A. A. Samah, M. H. Alsharif, and M. A. Khan, “Topology-Based Routing Protocols and Mobility Models for Flying Ad Hoc Networks: A Contemporary Review and Future Research Directions,” *Drones*, vol. 6, no. 1, Art. no. 1, Jan. 2022, doi: 10.3390/drones6010009.

[5] T. Clausen and P. Jacquet, *RFC3626: Optimized Link State Routing Protocol (OLSR)*. USA: RFC Editor, 2003.

[6] Md. Z. Hassan, Md. M. Hossain, and S. M. J. Alam, “The Recent Variants of OLSR Routing Protocol in MANET: A Review,” *Int. J. Adv. Netw. Appl.*, vol. 16, no. 01, pp. 6275–6280, 2024, doi: 10.35444/IJANA.2024.16106.

[7] P. Kuppusamy, K. Thirunavukkarasu, and B. Kalaavathi, “A study and comparison of OLSR, AODV and TORA routing protocols in ad hoc networks,” in *2011 3rd International Conference on Electronics Computer Technology*, Apr. 2011, pp. 143–147. doi: 10.1109/ICECTECH.2011.5941974.

[8] C. C. Dearlove, T. H. Clausen, and P. A. Jacquet, “RFC7185: Rationale for the Use of Link Metrics in the Optimized Link State Routing Protocol Version 2 (OLSRv2),” report, The Internet Engineering Task Force (IETF), 2014. doi: 10.17487/RFC7185.

[9] Z. Wang, C. Li, and X. Wu, “A Multi-Point Two-Hop Neighbor Relay Based OLSR Routing Protocol Design Method for Self-organized Multi-agent Communication*,” in *2024 Australian & New Zealand Control Conference (ANZCC)*, Feb. 2024, pp. 143–148. doi: 10.1109/ANZCC59813.2024.10432854.

[10] I. Fourfouris *et al.*, “Revisiting the OLSRv2 Protocol Optimization in SDN-enabled Tactical MANETs,” in *MILCOM 2023 - 2023 IEEE Military Communications Conference (MILCOM)*, Oct. 2023, pp. 779–786. doi: 10.1109/MILCOM58377.2023.10356322.

[11] A. H. Wheeb, R. Nordin, A. A. Samah, and D. Kanellopoulos, “Performance Evaluation of Standard and Modified OLSR Protocols for Uncoordinated UAV Ad-Hoc Networks in Search and Rescue Environments,” *Electronics*, vol. 12, no. 6, Art. no. 6, Jan. 2023, doi: 10.3390/electronics12061334.

[12] “[OpenWrt Wiki] Welcome to the OpenWrt Project.” Accessed: May 15, 2025. [Online]. Available: https://openwrt.org/

[13] J. Damasceno, J. Dantas, and J. Araujo, “Network Edge Router Performance Evaluation: An OpenWrt-Based Approach,” in *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, June 2022, pp. 1–6. doi: 10.23919/CISTI54924.2022.9820027.

[14] “Kategorie:English – wiki.freifunk.net.” Accessed: May 15, 2025. [Online]. Available: https://wiki.freifunk.net/Kategorie:English

[15] H. Khan, K. K. Kushwah, J. S. Thakur, G. G. Soni, A. Tripathi, and S. Rao, “Performance Evaluation of DSR, AODV and MP-OLSR Routing Protocols Using NS-2 Simulator in MANETs,” in *Cognitive Computing and*

Cyber Physical Systems, P. Pareek, N. Gupta, and M. J. C. S. Reis, Eds., Cham: Springer Nature Switzerland, 2024, pp. 122–133. doi: 10.1007/978-3-031-48891-7_10.

[16] Dolly Thakre, Sandeep Awaya, "Performance Study of AODV, OLSR, and DSDV Routing Protocols in Mobile AD HOC Networks," *Int. J. Microw. Eng. Technol.*, vol. 10, no. 2, 2024, doi: 10.37628/IJMET.

[17] A. Sumarudin and T. Adiono, "The design of high throughput Wi-Fi mesh networked wireless sensor network using OLSR protocol," in *2015 International Conference on Automation, Cognitive Science, Optics, Micro Electro-Mechanical System, and Information Technology (ICACOMIT)*, Oct. 2015, pp. 192–195. doi: 10.1109/ICACOMIT.2015.7440204.

[18] D. Lumbantoruan and A. Sagala, "Performance evaluation of OLSR routing protocol in ad hoc network," vol. 10, pp. 1178–1184, Jan. 2015.

[19] A. Barolli, T. Oda, L. Barolli, and M. Takizawa, "Experimental Results of a Raspberry Pi and OLSR Based Wireless Content Centric Network Testbed Considering OpenWRT OS," in *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, Mar. 2016, pp. 95–100. doi: 10.1109/AINA.2016.153.

[20] A. Barolli, D. Elmazi, R. Obukata, T. Oda, M. Ikeda, and L. Barolli, "Experimental results of a raspberry Pi and OLSR based wireless Content Centric Network testbed: Comparison of different platforms," *Int. J. Web Grid Serv.*, vol. 13, p. 131, Jan. 2017, doi: 10.1504/IJWGS.2017.082064.

[21] M. Hachtkemper, M. Rademacher, and K. Jonas, *Real-World Performance of current Mesh Protocols in a small-scale Dual-Radio Multi-Link Environment*. 2017.

[22] J. Astudillo and L. de la Cruz Llopis, *A Testbed Based Performance Evaluation of Smart Grid Wireless Neighborhood Area Networks Routing Protocols*. 2020. doi: 10.3233/AISE200054.

[23] Wardi, Dewiani, M. Baharuddin, S. Panggalo, and M. F. B. Gufran, "Performance of Routing Protocol OLSR and BATMAN in Multi-hop and Mesh Ad Hoc Network on Raspberry Pi," *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 875, no. 1, p. 012046, June 2020, doi: 10.1088/1757-899X/875/1/012046.

[24] D. Turlykozhayeva *et al.*, "Experimental Performance Comparison of Proactive Routing Protocols in Wireless Mesh Network Using Raspberry Pi 4," *Telecom*, vol. 5, no. 4, Art. no. 4, Dec. 2024, doi: 10.3390/telecom5040051.

[25] M. S. D. Pabana, A. P. T. Drawino, R. A. P. Pratama, T. A. Wibowo, and L. V. Yovita, "Comparison of OpenWRT and Ubuntu for Named Data Networking," in *2024 IEEE International Conference on Communication, Networks and Satellite (COMNETSAT)*, Nov. 2024, pp. 766–772. doi: 10.1109/COMNETSAT63286.2024.10862781.

[26] G. Howser, "Raspberry Pi Operating System," in *Computer Networks and the Internet: A Hands-On Approach*, G. Howser, Ed., Cham: Springer International Publishing, 2020, pp. 119–149. doi: 10.1007/978-3-030-34496-2_8.

[27] M. Granderath and J. Schönwälder, "A Resource Efficient Implementation of the RESTCONF Protocol for OpenWrt Systems," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2020, pp. 1–6. doi: 10.1109/NOMS47738.2020.9110458.

[28] "[OpenWrt Wiki] Supported devices." Accessed: May 16, 2025. [Online]. Available: https://openwrt.org/supported_devices

[29] "OpenWrt Firmware Selector." Accessed: May 15, 2025. [Online]. Available: https://firmware-selector.openwrt.org/

[30] T. Hardes, F. Dressler, and C. Sommer, “Simulating a city-scale community network: From models to first improvements for Freifunk,” in *2017 International Conference on Networked Systems (NetSys)*, Mar. 2017, pp. 1–7. doi: 10.1109/NetSys.2017.7903954.

[31] “Freifunk-Firmware – wiki.freifunk.net.” Accessed: May 16, 2025. [Online]. Available: https://wiki.freifunk.net/Freifunk-Firmware

[32] “datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf.” Accessed: Sept. 17, 2025. [Online]. Available: https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf

[33] “Omega2+ – Onion.” Accessed: May 16, 2025. [Online]. Available: https://onion.io/store/omega2p/

[34] R. K. Kodali, L. Boppana, H. Bandaru, and S. D. Avuthu, “Low Cost IoT Application Using Onion Omega2+,” in *2021 International Conference on Computer Communication and Informatics (ICCCI)*, Jan. 2021, pp. 1–4. doi: 10.1109/ICCCI50826.2021.9402222.

[35] “brcmfmac IBSS mode doesn’t honor specified cell address.” Accessed: Sept. 16, 2025. [Online]. Available: https://community.infineon.com/t5/Wi-Fi-Bluetooth-for-Linux/brcmfmac-IBSS-mode-doesn-t-honor-specified-cell-address/m-p/731315#M3058

[36] Ingo, “How to setup an unprotected Ad Hoc (IBSS) Network and if possible with WPA encryption?,” Raspberry Pi Stack Exchange. Accessed: Sept. 16, 2025. [Online]. Available: https://raspberrypi.stackexchange.com/q/94047

[37] raspberrypi, “Raspberry Pi 2018-04-18 - Cannot put device into adhoc mode · Issue #990 · raspberrypi/firmware,” GitHub. Accessed: Sept. 16, 2025. [Online]. Available: https://github.com/raspberrypi/firmware/issues/990

[38] *ANRGUSC/Raspberry-Pi-OLSRd-Tutorial*. (Oct. 25, 2024). Shell. ANRG USC. Accessed: Sept. 16, 2025. [Online]. Available: https://github.com/ANRGUSC/Raspberry-Pi-OLSRd-Tutorial

[39] “Creating a Mesh Network (with OLSR Protocol) for multiple Access Points - Installing and Using OpenWrt / Network and Wireless Configuration,” OpenWrt Forum. Accessed: May 19, 2025. [Online]. Available: https://forum.openwrt.org/t/creating-a-mesh-network-with-olsr-protocol-for-multiple-access-points/46804/12

[40] V. Tilwari *et al.*, “MCLMR: A Multicriteria Based Multipath Routing in the Mobile Ad Hoc Networks,” *Wirel. Pers. Commun.*, vol. 112, no. 4, pp. 2461–2483, June 2020, doi: 10.1007/s11277-020-07159-8.

[41] X. Wang, D. Li, X. Zhang, and Y. Cao, “MCDM-ECP: Multi Criteria Decision Making Method for Emergency Communication Protocol in Disaster Area Wireless Network,” *Appl. Sci.*, vol. 8, no. 7, p. 1165, July 2018, doi: 10.3390/app8071165.

[42] V. Tilwari, T. Song, U. Nandini, V. Sivasankaran, and S. Pack, “A multi-criteria aware integrated decision making routing protocol for IoT communication toward 6G networks,” *Wirel. Netw.*, vol. 30, no. 5, pp. 3321–3335, July 2024, doi: 10.1007/s11276-024-03739-9.