10

EXPLORING MESH NETWORKING

As you learned in Chapter 1, Wi-Fi networks operate in one of two ways: infrastructure or ad hoc mode. The vast majority of wireless networks we interact with operate in infrastructure mode, where all devices communicate through a single access point. Ad hoc mode, by contrast, doesn't rely on a centralized router but rather distributes control to individual peers in the network.

Mesh networking, as its name implies, is meant to work via a large number of short-haul connections without any sort of centralized control. This is in contrast to traditional networks that follow a hub-and-spoke topology, where all devices connect to a central router or Wi-Fi base station. A proper mesh network should configure itself dynamically, responding to the addition and removal of nodes and changes in connectivity. In a well-functioning mesh, networking "just happens" without high-level coordination.

Mesh networks have earned a reputation as being difficult for do-it-yourself enthusiasts to implement. In this chapter, I'll dispel this myth by demonstrating a clear, methodical approach to building a resilient and scalable mesh network with inexpensive, off-the-shelf components. This recipe will work with two or more Raspberry Pi devices. Once you've successfully connected two nodes by using the ad hoc wireless routing protocol, the only

limitation is available hardware: You can apply the same configuration steps to subsequent nodes and expand the mesh with ease.

A Brief History

The decentralized and ad hoc nature of mesh networks affords them many advantages. A network of this type is often described as *self-healing*, meaning that if one node fails or becomes unavailable, data packets can still be routed through alternate paths. This redundancy and fault tolerance implies that mesh networks should, in theory, be very difficult to disrupt.

It should perhaps come as no surprise that the origins of wireless mesh networks can be traced back to military applications. Among the earliest known wireless mesh networks was the Packet Radio Network (PRNET) project initiated by the Defense Advanced Research Projects Agency (DARPA) in the late 1970s. This project involved the development of protocols and algorithms for routing data through a network of wireless nodes, paving the way for today's modern mesh networks.

MIT's Roofnet project in the early 2000s was an initiative to deploy a large-scale wireless mesh network to provide internet access to residents in Cambridge, Massachusetts. Roofnet's technology later formed the basis for Meraki, a mesh networking startup spun off by several MIT engineers. Meraki was subsequently acquired by Cisco Systems and is still in use today.

Around the same time in Europe, the Freifunk Paderborn project was started to deliver free and open wireless mesh connectivity to residents in the city of Paderborn, Germany. Participants install open source firmware on their wireless routers, which allows them to share a portion of their internet connection with others in the community. The Freifunk mesh network has been in operation continuously since 2002 (a section of the network is illustrated in Figure 10-1).

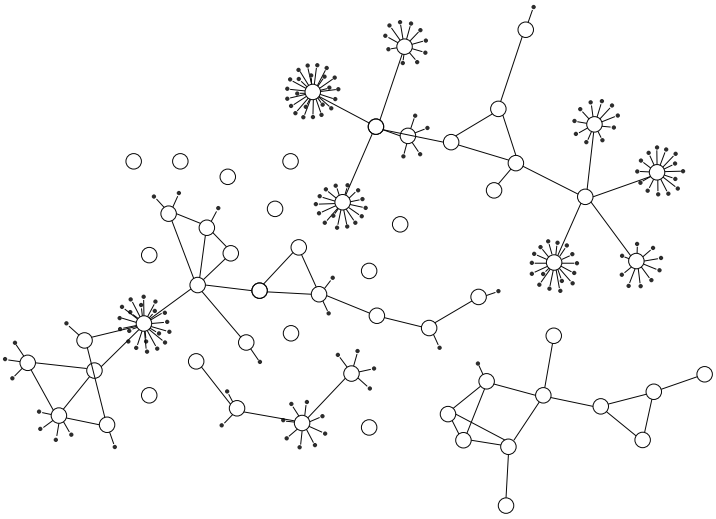


Figure 10-1: A small section of the Freifunk Paderborn wireless mesh network simulation

Since its introduction, Freifunk has grown into a global movement promoting community-driven mesh networking with open source software.

Use Cases

Initiatives like Freifunk are among the most popular examples of practical, community-based mesh network applications. These types of networks have also proven to be essential in providing communication lifelines when traditional networks have been either destroyed or severely disrupted. Since they can be deployed quickly, mesh networks have been used in the aftermath of numerous natural disasters in recent years, including earthquakes, hurricanes, floods, and wildfires.

Today, many commercial product offerings aim to replace your home Wi-Fi base station with a small constellation of wireless mesh nodes. A key value proposition to consumers is that these systems, while somewhat pricier on average than traditional home network router setups, require very little knowledge to deploy. A home user needn't know anything about what's happening under the hood; the mesh nodes dynamically and continuously adjust their routing parameters to optimize coverage and throughput.

Owing to the relatively small size, weight, and power requirements of wireless mesh hardware, NASA has identified this technology for use in both human and robotic space exploration. One recent area of practical research at NASA has focused on the Orion spacecraft's camera system. In fact, the mesh protocol evaluated by NASA is the very same one you'll be using in this chapter.

Hardware Required

This recipe can be implemented with any number of available Raspberry Pi devices. If my experience is any indication, you may find yourself eager to press additional devices into service after the mesh is up and running, making this recipe ideally suited as a group project.

Compatible Raspberry Pi Models

It's certainly possible to build a proof-of-concept mesh network with just two devices. However, things become more interesting as more nodes are added to the network. The only strict requirement is an available USB port that can accept an external wireless adapter. In addition, at least one device on your mesh network should be equipped with an Ethernet port, for reasons that will be discussed in the implementation steps.

You can use any Raspberry Pi model for this project, but its affordability and small form factor make the Zero 2 W an ideal choice. If you opt to use this model, bear in mind that you'll need a micro USB to USB-C adapter to connect an external wireless adapter, as the onboard WLAN chipset does not support the required mesh mode.

The Pi Zero 2 W is widely available from online retailers for around $15. It comes standard without a GPIO header, as pictured in Figure 10-2, so you'll need to obtain a 40-pin header and perform some basic soldering to attach it.

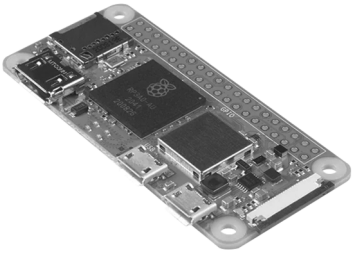


Figure 10-2: A standard Pi Zero 2 W

Alternatively, you can avoid soldering by purchasing a model with a presoldered header. Several retailers offer these, usually at little extra cost.

Mesh-Capable Adapter

If you've followed the recipe in Chapter 7, you'll already be familiar with the Edimax EW-7811Un Nano. This inexpensive yet capable USB wireless adapter, shown in Figure 10-3, has everything you need to begin exploring mesh networking, including in-kernel Linux driver support.

Figure 10-3: The Edimax EW-7811Un USB wireless adapter

At minimum, your Pi-based mesh network will comprise two nodes, so plan on budgeting for an equal number of these adapters. The Edimax adapter is available from various online retailers from about $13.

Squid RGB (Optional)

You'll acquire hands-on experience with several software tools for monitoring your mesh network in this chapter, but it can be quite useful to have at-a-glance visual indicators of network packet traffic on your mesh nodes. As an optional component, the Pi Hut's Squid RGB is an excellent addition to this project; you can pick up one for each node you want to monitor in this way.

As shown in Figure 10-4, the Squid RGB has separate leads that control the red, green, and blue channels of the attached LED.

Figure 10-4: The Squid RGB from the Pi Hut

The Squid RGB's lead sockets fit directly onto the Pi's GPIO header, so no soldering is required. It's available directly from the Pi Hut for about $6.

Uninterruptible Power Supply HAT

Optionally, you may choose to equip one or more of your Raspberry Pi Zero 2 Ws with an add-on battery HAT. This isn't strictly necessary, but you can often gain great insights if one or more of the nodes in your mesh network is completely untethered and free to move around a given service area. This is an excellent way to evaluate the self-healing nature of your mesh and to observe how a phenomenon known as *hopping* occurs in practice.

Many affordable options exist for the Raspberry Pi Zero and other models. A favorite in my testing lab is the Waveshare UPS HAT, shown in Figure 10-5.

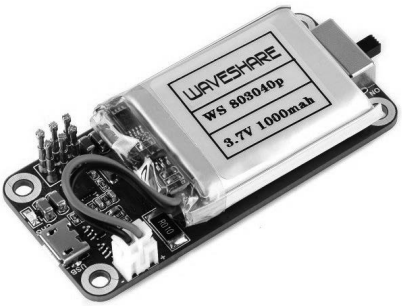


Figure 10-5: The Waveshare UPS HAT for Raspberry Pi Zero

It's available directly from Waveshare for about $24.

Software Used

The primary software component you'll be using for this recipe is the Better Approach to Mobile Ad hoc Networking (BATMAN) protocol. A BATMAN mesh is made up of a set of *originators*, which communicate via network interfaces such as standard wireless adapters. Periodically, each originator broadcasts an originator message (OGM) to all its neighbors to announce its presence. Each neighbor makes a note of the presence of the originator and rebroadcasts the message to its own neighbors. The net effect is that, over time, every node in the mesh receives the OGM (possibly via multiple

paths) and learns both that the originator exists and which of its neighbors provides the best path to reach it. Each node maintains a routing table that lists every node it has ever heard of, along with the best next hop to reach that node.

This protocol has the advantage of building and updating the routing tables on the fly, with no central coordination needed. It should also find near-optimal routes to each node. If a node goes offline, the routing tables will reconfigure themselves to maintain connectivity in its absence. In all but the smallest networks, no single node has a complete view of how the mesh is constructed; each node knows only which of its neighbors are available and which is the best next hop to get to a given node. This decentralized approach adds to the security and robustness of the mesh.

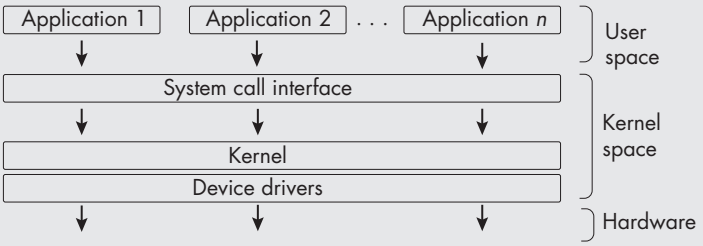
Implemented in Linux as the `batman-adv` kernel module, the BATMAN protocol has been part of the official kernel since version 2.6.38, released in early 2011. This module operates at Layer 2 of the OSI network model (the Data Link layer, discussed in the box titled "The OSI Network Model Explained" on page 322). As the module is already present in most Linux kernels, you shouldn't need to install it from scratch.

Because `batman-adv` operates in kernel land, a tool to manage the module and debug the network is required. The `batctl` utility was created to fill that role and has proven an indispensable companion to `batman-adv`.

KERNEL VS. USERLAND

In this context, *kernel land* (or *kernel space*) refers to the privileged area of virtual memory where the operating system's kernel is stored and executed. Modern operating systems, including Linux, use this separation to protect the kernel from malicious or errant software and to isolate hardware access.

By contrast, *userland*, also known as *user space*, is the area of memory where code that runs outside the operating system's kernel (such as application software, libraries that interact with the kernel, and some drivers) resides. This division is depicted in the following illustration, which shows a typical Linux operating system architecture.



Naturally, this recipe wouldn't be a complete without a dash of Python. Once you delve into the wilds of mesh networking, you'll likely want to add more devices to your network, and you'll want to be able to monitor their status at a glance. While not required for operation of the mesh network,

you can use a bit of Python code to flash an LED attached to your Pi. This will give you a simple mesh network traffic indicator.

This recipe will walk you through creating the mesh network step-by-step, so you can gain hands-on experience with the tools and processes involved. After that, you'll use a simple bash script to help you automate this process.

All the source code for this recipe is available at *https://github.com/wirelesscookbook/pi-mesh*.

Preparing the Nodes

To maximize the flexibility of your setup, you'll configure your mesh network nodes to operate without an Ethernet connection from the outset. With the nodes untethered by a network cabling, you'll be better able to move them around to test the resiliency of your mesh. You can, of course, make them entirely wireless by trading the power supplies for attached battery modules. Later, I'll discuss configuring one node as a gateway to provide internet connectivity to the rest of the network.

There is no hard-and-fast rule about using `wpa_supplicant` instead of Ethernet here. If you prefer to access your devices with `eth0` rather than a wireless interface, the outcome of the recipe will be the same.

For ease of setup later on, leave your Edimax EW-7811Un USB wireless adapter disconnected at this stage. You'll perform some basic configuration steps later to assist with streamlining the build process.

You'll need a minimum of two devices to create a basic mesh network. As noted in "Hardware Required," on page 311, you can use any Raspberry Pi model capable of supporting a USB wireless adapter. Repeat the steps described here for each device you plan to deploy on your network.

Begin by preparing the SD card. For this recipe, you'll use a legacy version of Raspberry Pi OS, Bullseye Lite (64- or 32-bit). The Bookworm distribution's kernel drivers for the wireless chipset used here require additional work to fully support the mesh networking mode, but Bullseye includes well-tested drivers that are known to work reliably with this recipe's hardware.

Flash a fresh SD card with Raspberry Pi OS (Legacy) Lite, available on the official download page, then create an empty file named *ssh* (no extension) and save it to the card's boot partition.

You'll use `wpa_supplicant` to connect to your existing wireless network. The built-in wireless interface will handle connecting your device to the network, while the interface provided by the USB wireless adapter will be used to create the mesh network. Using your preferred editor, create a file called *wpa_supplicant.conf* with the following contents:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=US

network={
    ssid="your_SSID"
```

```
    psk="your_wi-fi_password"
    key_mgmt=WPA-PSK
}
```

Substitute your two-letter ISO 3166-1 alpha-2 country code for `US` if you're not in the United States, and be sure both your network SSID and your password (PSK) are enclosed in double quotes. Copy this file to the card's boot partition. You'll need to complete this step before you boot the SD card for the first time, as that's the point at which the system checks for the presence of the *wpa_supplicant.conf* file.

Alternatively, you can perform this configuration step with the Raspberry Pi Imager tool while writing the OS to your SD card. Under **OS Customisation**, choose the **Wireless LAN** option and enter an SSID (name) and the password for your network. If the network doesn't broadcast an SSID publicly, enable the **Hidden SSID** setting.

Finally, ensure that your Pi is within range of your wireless router, then insert the SD card and connect it to power.

Creating the Mesh Network

At this stage, I'll assume that your required minimum two devices are booted and accessible on your WLAN. As with the previous section, repeat the steps described here for each device you plan to configure for your network.

The `batman-adv` driver is already present in the Linux kernel, but you'll need to install its companion userland tool, `batctl`. This will provide you with a full set of tools for creating, monitoring, and troubleshooting a single mesh node or the entire network. You'll also install `git` to support a later step. Since you're starting with a clean OS, begin with a full upgrade to be sure you have the latest kernel and packages, then install these prerequisites and reboot:

```
$ sudo apt update && sudo apt full-upgrade
$ sudo apt install batctl git
$ sudo reboot
```

You'll need to instruct the DHCP client daemon, `dhcpcd`, to ignore `wlan1` when discovering new interfaces. To do this, use `dhcpcd`'s `denyinterfaces` pattern. Open *dhcpcd.conf* in your editor with

```
$ sudo nano /etc/dhcpcd.conf
```

and add this directive to the end of the file:

```
denyinterfaces wlan1
```

Next, instruct `dhcpcd` to configure the `wlan0` interface with `wpa_supplicant`. Append the following below the line you added previously:

```
interface wlan0
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
```

Save the file and exit your editor. With these steps done, restart the `dhcpcd` service for the changes to take effect:

```
$ sudo systemctl restart dhcpcd.service
```

If you're using NetworkManager, use the following modification to stop it from managing the Wi-Fi adapter:

```
$ sudo touch /etc/network/interfaces
$ sudo echo "iface wlan1 inet manual" > /etc/network/interfaces
```

Now, connect your Edimax EW-7811Un wireless adapter to one of the Pi's USB ports. By connecting it after boot, you can more reliably have it bound to `wlan1`, which leaves it unused by `wpa_supplicant` and ready to be put into service as a mesh point. With the adapter connected to your device, run **`lsusb`** to confirm that it's recognized by the OS. You should see information such as the device ID, manufacturer, model name, and chipset.

If this is a fresh install, ensure that Wi-Fi isn't blocked by `rfkill`:

```
$ sudo rfkill unblock wlan
```

At this point, you can use `iw` to list the available wireless devices. For clarity, I'll refer to this first node as *Mesh-Pi #1*. Example output is shown here (yours may differ):

```
$ iw dev
phy#1
    Interface wlan1
        ifindex 4
        wdev 0x100000001
        addr 80:1f:02:9b:b0:c4
        type managed
        txpower 20.00 dBm
phy#0
    Interface wlan0
        ifindex 3
        wdev 0x1
        addr dc:a6:32:3d:ff:9d
        ssid Home-Router
        type managed
        channel 11 (2462 MHz), width: 20 MHz, center1: 2462 MHz
        txpower 31.00 dBm
```

Take note of the physical address (`phy#1`, in this example) for the adapter bound to the `wlan1` interface. You can inspect this wireless device's capabilities by running:

```
$ iw phy1 info | grep "Supported interface modes" -A10
    Supported interface modes:
        * IBSS
```

```
        * managed
        * AP
        * AP/VLAN
        * monitor
        * mesh point
        * P2P-client
        * P2P-GO
    --snip--
```

Notice that the Edimax adapter's `mesh point` mode is listed as an available option. Since you know the physical address (`phy#1`) and the interface (`wlan1`), you may now use `iw` to reconfigure this interface as a mesh point. Begin by removing the existing interface so you can define its type:

```
$ sudo iw dev wlan1 del
```

Now, redefine this interface as a `mesh` type, like so:

```
$ sudo iw phy phy1 interface add wlan1 type mesh
```

You can confirm the change by checking the output of `iw` again:

```
$ iw wlan1 info
Interface wlan1
    ifindex 5
    wdev 0x100000002
    addr 80:1f:02:9b:b0:c4
    type mesh point
    wiphy 1
    txpower 20.00 dBm
```

Next, use `ip` to set the MTU value for the interface, then call `iw` to join it to the `pi-mesh` network (you can change this name if you like, just be sure to use it consistently):

```
$ sudo ip link set mtu 1532 dev wlan1
$ sudo ip link set wlan1 up
$ sudo iw dev wlan1 mesh join pi-mesh
```

Now, you'll use `batctl` to instruct the `batman-adv` kernel module to create the new virtual `bat0` mesh interface:

```
$ sudo batctl if add wlan1
$ sudo ip link set up dev bat0
```

These commands will typically not produce any output in the terminal. However, you can use `dmesg` to view these events in the kernel's ring buffer by executing the following command:

```
$ dmesg | grep batman_adv
[  145.630607] batman_adv: B.A.T.M.A.N. advanced 2022.3 loaded
```

```
[  145.635757] batman_adv: bat0: Adding interface: wlan1
[  145.635886] batman_adv: bat0: Interface activated: wlan1
```

Each batman-adv node maintains a list of all single-hop neighbors it detects. Whether a single-hop neighbor is routed to directly or via another neighbor is determined based on the link quality. You can view the current node's neighbor table with:

```
$ sudo batctl neighbors
[B.A.T.M.A.N. adv 2020.4, MainIF/MAC: wlan1/80:1f:02:9b:b0:c4
(bat0/66:24:2a:20:a2:71 BATMAN_IV)]
IF             Neighbor              last-seen
```

At the moment, no other nodes are visible because you haven't yet enabled mesh support on your other devices.

Working with MTU Values

In TCP/IP networking, the *maximum transmission unit (MTU)* refers to the size, in bytes, of the largest datagram that a given layer of a communications protocol may pass in a single transaction. A large MTU value requires less overhead, while a smaller MTU has less delay. This value will vary depending on the most appropriate size for a given application.

The default MTU size for most Ethernet networks is 1,500 bytes. You can check the MTU setting for the interfaces on your device by running:

```
$ ip a | grep mtu
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 noqueue state UNKNOWN group default
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP group default
4: wlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1532 state UP group default
6: bat0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UNKNOWN group default
```

The Linux module batman-adv will advise you, via dmesg, when the MTU setting of your configured mesh interface falls outside its recommended range. For example, you may see a message such as:

```
The MTU interface wlan1 is too small (1500) to handle the transport of
batman-adv packets. Packets going over this interface will be fragmented
on layer2 which could impact performance. Setting the MTU to 1532 would
solve the problem.
--snip--
```

The MTU value is reflected in your configuration for the Edimax EW-7811Un wireless adapter's wlan1 interface. If you receive an error such as "MTU greater than device maximum," this usually indicates a limitation of the wireless adapter hardware. Helpfully, ip will report the minimum and maximum supported MTU values for a given interface. Use the -d or

`-details` option with `ip` and pipe it through `grep` to get detailed information about the interface:

```
$ ip -details link list | grep wlan1 -A1
5: wlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1532 bat0 state UP mode DEF...
    link/ether 74:da:38:ed:5e:7d... promiscuity 0 minmtu 256 maxmtu 2304
```

These values appear as `minmtu` and `maxmtu` in the output.

Adding More Nodes

To add more nodes to your mesh network, repeat the steps in the preceding section on your other device(s). When you're done, execute the following commands on each device to view the interface status and neighbor table, respectively. On Mesh-Pi #1, you should see output similar to this:

```
$ sudo batctl interface
wlan1: active

$ sudo batctl neighbors
[B.A.T.M.A.N. adv 2020.4, MainIF/MAC: wlan1/80:1f:02:9b:b0:c4
(bat0/d6:43:53:6c:61:88 BATMAN_IV)]
IF          Neighbor              last-seen
wlan1       74:da:38:ed:5e:94     0.570s
```

Now, execute the same commands on Mesh-Pi #2. Take note of the different MAC addresses:

```
$ sudo batctl interface
wlan1: active
$ sudo batctl neighbors
[B.A.T.M.A.N. adv 2020.4, MainIF/MAC: wlan1/74:da:38:ed:5e:94
(bat0/0e:36:d1:d2:dc:28 BATMAN_IV)]
IF          Neighbor              last-seen
wlan1       80:1f:02:9b:b0:c4     0.290s
```

As you can see, the MAC addresses of each device in the mesh is visible to the other node.

In these examples, the `batman-adv` interface `bat0` is used as a default parameter. You can specify the interface to use with the `meshif` option, and you can use the abbreviated form of the `neighbors` command, `n`, if you prefer:

```
$ sudo batctl meshif bat0 n
```

Diagnosing Mesh Network Connectivity

Most wireless routing protocols operate at Layer 3 of the OSI network model (see the box titled "The OSI Network Model Explained" on page 322 for a brief overview). This means they exchange routing information by

sending UDP packets and make routing decisions by manipulating the kernel routing table. By contrast, `batman-adv` operates entirely at Layer 2, meaning it handles not only the routing information but also the data traffic itself. In practice, `batman-adv` encapsulates and forwards all traffic until it reaches its destination, effectively emulating a virtual network switch with all nodes participating. This is illustrated in Figure 10-6.

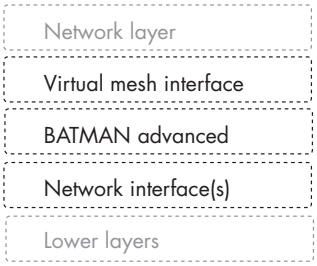


Figure 10-6: The batman-adv encapsulation structure

For this reason, all nodes appear to be *link-local* and are unaware of the network's topology. Similarly, mesh nodes are unaffected by changes within the network.

LINK-LOCAL ADDRESSES

In this context, *link-local* refers to addresses on a TCP/IP network that interfaces can automatically adopt if an address has not been manually configured for them or assigned by DHCP. These addresses are often used for automatic address configuration or neighbor discovery within a single link, or when no router is present. They also enable direct communication between nodes on the same network segment.

Traditional network debugging tools based on the Internet Control Message Protocol (ICMP), such as `ping` and `traceroute`, won't work as expected with `batman-adv`. This is because all traffic in the mesh is transported to its destination transparently at Layer 2, so higher-layer protocols have no visibility of hop counts or path details.

This transparency is one of the main reasons you can roam around without breaking your connection. To provide comparable diagnostic tools, `batman-adv` includes its own version of ICMP, which is integrated directly into the protocol. Here, I'll demonstrate using `batctl ping` to inject IMCP packets that behave very similarly to their Layer 3 network counterpart.

Execute the following command on Mesh-Pi #1, replacing the MAC address with the one for your own device. Your output will differ, but it should have a similar format to the example output shown here:

```
$ sudo batctl ping 74:da:38:ed:5e:94
PING 74:da:38:ed:5e:94 (74:da:38:ed:5e:94) 20(48) bytes of data
```

```
20 bytes from 74:da:38:ed:5e:94 icmp_seq=7 ttl=50 time=12.99 ms
20 bytes from 74:da:38:ed:5e:94 icmp_seq=8 ttl=50 time=18.18 ms
```

You can also perform this ping test on Mesh-Pi #2:

```
$ sudo batctl ping 80:1f:02:9b:b0:c4
PING 80:1f:02:9b:b0:c4 (80:1f:02:9b:b0:c4) 20(48) bytes of data
20 bytes from 80:1f:02:9b:b0:c4 icmp_seq=1 ttl=50 time=6.01 ms
20 bytes from 80:1f:02:9b:b0:c4 icmp_seq=7 ttl=50 time=13.71 ms
```

Interrupt `ping` with CTRL-C. This is just a basic network connectivity test. In the sections that follow, I'll show you additional steps you can use to verify the integrity of your new mesh network.

THE OSI NETWORK MODEL EXPLAINED

The *Open Systems Interconnect (OSI) model* is a conceptual framework that breaks down networking into seven layers, each with its own role and responsibilities. These layers help engineers visualize what's going on in their networks and pinpoint where issues occur. For our purposes, we're primarily concerned with Layers 3 (Network), 2 (Data Link), and 1 (Physical). Here's a brief overview of each:

Layer 3 (Network) At the Network layer, you'll find most of the routing functionality that networking professionals and hobbyists work with. In its most basic sense, this layer is responsible for packet forwarding, including routing traffic through different routers.

Layer 2 (Data Link) The Data Link layer provides node-to-node data transfer between directly connected devices and handles error detection and correction from the Physical layer. It has two sublayers: the Media Access Control (MAC) layer and the Logical Link Control (LLC) layer. In the networking world, most switches operate at Layer 2; however, some also operate at Layer 3 to support virtual LANs that span more than one switch subnet, which requires routing capabilities.

Layer 1 (Physical) At the bottom of the OSI model is the Physical layer, which represents the electrical and physical hardware. This can include everything from radio frequencies (as in a Wi-Fi network) and voltages to cables, hubs, repeaters, and other physical assets. When a networking problem occurs, engineers commonly start at the Physical layer, checking that all the hardware is properly connected and no components have lost power.

Creating Mesh Hostnames

Until now, you've referred to your mesh devices by their MAC addresses. Here, you'll create a file called */etc/bat-hosts* and define hostnames for each device. This step is not required, but it can make common tasks easier to perform and help clarify diagnostic output, as the symbolic names will be used instead of the MAC addresses in the output of many `batctl` commands.

Create the file with

```
$ sudo nano /etc/bat-hosts
```

and define your mesh hostnames as shown here, replacing the MAC addresses with those of your devices:

```
80:1f:02:9b:b0:c4   mesh-pi1
74:da:38:ed:5e:94   mesh-pi2
```

You can use different names if you prefer; they don't need to match the device hostnames or be consistent with DNS or any other naming schemes. When you're done, save the file and exit your editor. The next time you execute a `batctl` command, you should see the mesh node MAC addresses replaced with symbolic names. For example:

```
$ sudo batctl meshif bat0 neighbors
[B.A.T.M.A.N. adv 2022.3, MainIF/MAC: wlan1/74:da:38:ed:5e:7d
  (bat0/66:52:d4:22:36:67 BATMAN_IV)]
IF          Neighbor              last-seen
wlan1       mesh-pi2              0.820s
```

This can be helpful for tracking and diagnosing issues with individual nodes, particularly in larger mesh networks. You may even want to consider affixing physical labels to your devices that correspond to these symbolic names.

Running at Boot

The steps you've used to configure each mesh node will need to be performed each time the device is rebooted. To streamline this process, you can combine them into a bash startup script that you can run after a reboot. Begin by cloning the companion GitHub repository for this project:

```
$ cd $HOME
$ git clone https://github.com/wirelesscookbook/pi-mesh.git
```

At the next device boot, reconnect the Edimax adapter to a USB port if necessary, then run **`iw dev`** to confirm that it's available for use. With the presence of the `wlan1` interface verified, execute the startup script as shown here. The output should appear similar to the following:

```
$ cd $HOME/pi-mesh
$ ./startup.sh wlan1
Configuring wlan1 as mesh point
Physical wlan1 address is phy#1
Loading batman-adv kernel module
Releasing wlan1 interface
Adding wlan1 as mesh interface
Setting MTU value for batman-adv and joining pi-mesh
```

```
Adding wlan1 to batman-adv and bringing it up
Diagnostic output from batctl...
wlan1: active
[B.A.T.M.A.N. adv 2022.3, MainIF/MAC: wlan1/80:1f:02:9b:b0:c4
(bat0/ee:f8:60:47:7c:8a BATMAN_IV)]
IF          Neighbor              last-seen
wlan1       74:da:38:ed:5e:94     0.360s
```

If your mesh point interface is not `wlan1`, be sure to specify it when invoking the startup script. Otherwise, your device may become inaccessible when the interface is reconfigured. The script will automatically detect the physical interface, execute each of the required manual steps, and finish by performing a diagnostic with `batctl`, as indicated in the output, to check the status of the mesh interface and any connected nodes.

If you'd like this script to run automatically each time your mesh node boots, a *pi-mesh.service* systemd unit file is included in the GitHub repository. Begin by opening this file in your editor. If the path to *startup.sh* is different from */home/pi/pi-mesh*, be sure to adjust it accordingly in the unit file. Likewise, this service is configured to run as the *pi* user, but you can change this if necessary. When you're satisfied with the configuration, install and enable the service with:

```
$ sudo cp $HOME/pi-mesh/pi-mesh.service /etc/systemd/system/
$ sudo systemctl daemon-reload
$ sudo systemctl enable pi-mesh.service
```

Following a reboot, you can check its status with:

```
$ sudo systemctl status pi-mesh.service
```

You may have noticed that the `oneshot` type is defined in the systemd service unit file. This is used for services that perform a one-time task, then exit. In this case, the service handles initialization of the `wlan1` interface, brings up `bat0`, and joins the `pi-mesh` network. There's no need for the service to remain running in the background, so it appears as `inactive (dead)`. This is expected; you should see the status `(code=exited, status=0/SUCCESS)` returned in the output to indicate that it executed successfully.

Extending the Mesh Network

If you have additional devices to add to the mesh, repeat the previous steps for each one in turn. Verify each step, checking the kernel message log with `dmesg` to see if an error is thrown at any stage. Filtering kernel messages for `batman-adv` can be helpful for diagnosing problems. Example output is shown here:

```
$ dmesg | grep batman_adv
batman_adv: B.A.T.M.A.N. advanced 2022.3 (compatibility version 15) loaded
batman_adv: bat0: Interface deactivated: wlan1
```

```
batman_adv: bat0: Removing interface: wlan1
batman_adv: bat0: Adding interface: wlan1
batman_adv: bat0: Interface activated: wlan1
```

When you're satisfied with the state of your mesh network, move on to the next section, where you'll configure a gateway device to allow traffic to be routed to and from the internet.

Configure a Gateway and Allow Access

In the previous steps, you configured two or more Raspberry Pis to create a mesh network by using the `batman-adv` protocol. In this section, you'll ensure that you can access the devices forming the mesh and provide internet connectivity to them. To do this, you'll add a *gateway* node that allows mesh traffic to reach the internet, while maintaining a level of isolation and privacy for the mesh network itself. A visual representation is presented in Figure 10-7.

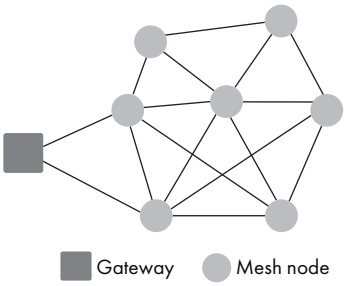


Figure 10-7: A conceptual view of a wireless mesh network with a single gateway

To make the internet uplink available to the mesh network, `batman-adv` lets you enable a so-called gateway mode. In this mode, the user-defined internet uplink bandwidth is propagated throughout the network. Since the gateway functionality is based on DHCP, as you'll see shortly, the protocol assumes that each gateway operates its own DHCP server and each client runs a DHCP client. When a DHCP client starts, it broadcasts a DHCP request to the entire network. Every available DHCP server sends a reply back to the client, which then chooses one of the responding DHCP servers as its gateway.

Depending on its topology, a mesh can have more than one gateway. Here, I'll demonstrate how to configure a single gateway for your mesh, but you can repeat the same process for any number of devices.

WHAT ROLE DOES A GATEWAY PLAY?

In the TCP/IP networking world, a gateway serves several key functions. First, it acts to hide or separate one network (your mesh, in this case) from another. Second, a gateway allows devices on a network to "see" devices on an external network by using *network address translation (NAT)*.

Simply put, this process maps multiple private, internal IP addresses to a unique public one. In this way, NAT allows a single device to act as an intermediary between a local, private network and the public internet. It achieves this by translating responses received from the internet back to the originating device on the local network.

A gateway also selectively forwards traffic from the public network to the private one. In a typical configuration, the gateway forwards only responses to traffic originating in the local (private) network. In this way, a gateway prevents internet traffic from "seeing" into your network, while still allowing devices on the network to access the internet.

If you've manually configured a routed access point, you'll already be familiar with NAT and IP forwarding. The technique explored in this section is functionally equivalent.

Among networking professionals, this gateway implementation is often referred to as a *Layer 3 routing solution* (in reference to the OSI model discussed in "Diagnosing Mesh Network Connectivity" on page 320).

Adding a Gateway

To create a gateway, you'll need to select one device from your mesh network and configure it accordingly. For simplicity, connect this device to your external network via Ethernet, so it can be accessed via `eth0`. The other devices in your mesh can remain connected via `wpa_supplicant`, if you opted to go with a fully wireless setup.

Since you're using IP routing to selectively move traffic between the mesh and your external network, you'll need to define a unique address range. The steps outlined here assume the following network details:

```
Gateway address 192.168.99.1
Subnet mask     255.255.255.0
Address range   192.168.99.50 - 192.168.99.150
```

In the previous section, you used the DHCP client daemon, `dhcpcd`, to configure the wireless interfaces for individual nodes in your mesh. `batman-adv` doesn't assign IP addresses itself; to do this, you'll need to host a DHCP server on your gateway. A separate software package called `dnsmasq` will handle automatic IP address assignment within the mesh. Installing it is straightforward:

```
$ sudo apt update
$ sudo apt install dnsmasq iptables
```

With the `dnsmasq` package installed, you can configure it to provide IP addresses to client nodes on the `bat0` mesh interface. To do this, you'll define a pool of available addresses for DHCP to draw from and set a lease time of 12 hours. This ensures that DHCP leases for devices that are no longer active on the network are freed, or released, after a predefined time. This helps make your mesh network more efficient, particularly if it's a dynamic one where nodes frequently go into and out of range.

Rather than editing the default `dnsmasq` configuration file (located at */etc/dnsmasq.conf*) directly, you can drop in a stand-alone configuration to keep things neater. Create the file in your editor with

```
$ sudo nano /etc/dnsmasq.d/bat0.conf
```

and add the following contents to it:

```
interface=bat0
dhcp-range=192.168.99.50,192.168.99.150,255.255.255.0,12h
```

Save the file and exit your editor. Next, you'll need to use a modified version of the startup script you created earlier for the gateway node. Examine the contents of this file by running:

```
$ cat $HOME/pi-mesh/startup-gw.sh
--snip--
iface=${1:-wlan1}
networkid="pi-mesh"

echo "Configuring ${iface} as mesh point..."

# get physical address of wlan1 adapter
addr=$(iw dev ${iface} info | awk '$1=="wiphy"{print $2}')
echo "Physical ${iface} address is phy#${addr}"

# load the module
echo "Loading batman-adv kernel module"
sudo modprobe batman-adv
--snip--

# tell batman-adv this is a gateway node
sudo batctl gw_mode server

# enable port forwarding
sudo sysctl -w net.ipv4.ip_forward=1
sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
sudo iptables -A FORWARD -i eth0 -o bat0 -m conntrack --ctstate RELATED,
  ESTABLISHED -j ACCEPT
sudo iptables -A FORWARD -i bat0 -o eth0 -j ACCEPT
--snip--
```

The key changes here are using `batctl` to tell `batman-adv` that this is a gateway node, assigning a static IP address to the gateway interface, and defining some NAT rules (as described earlier) with `iptables`. To ensure the `iptables` rules persist across reboots, install the following package:

```
$ sudo apt install iptables-persistent
```

At this point, your mesh node has all the required components to function as a gateway for the network. All that's left to do now is reboot the nodes and verify that IP addressing and internet access are working across your mesh network.

Rebooting the Mesh Network

Rebooting your gateway and the other nodes in the mesh will ensure that IP addressing and NAT are functioning as they should. Check that the gateway node is connected to your router via Ethernet, then reboot each of the Pis in your mesh network, starting with the gateway. Next, access the gateway node via `ssh` and run:

```
$ cd $HOME/pi-mesh
$ ./startup-gw.sh wlan1
--snip--
Configuring wlan1 as mesh point...

Physical wlan1 address is phy#0
Loading batman-adv kernel module
Releasing wlan1 interface
Adding wlan1 as mesh interface
Setting MTU value for batman-adv and joining pi-mesh
Tell batman-adv this is a gw_mode server
Enabling IP forwarding and NAT
net.ipv4.ip_forward = 1
Adding wlan1 to batman-adv and bringing it up
```

If you've installed the `pi-mesh` systemd service, you can skip the following step. Otherwise, connect to each of the other nodes via `ssh` and execute the "normal" (non-gateway-enabled) startup script:

```
$ cd $HOME/pi-mesh
$ ./startup.sh wlan1
```

When this process is complete, your mesh should be fully formed, and each node on the network should have an IP address assigned by the gateway. You'll confirm this in the next section.

Verifying Your Mesh

There are many tools and techniques you can use to verify the integrity of your mesh network. I'll demonstrate several of these here, starting with the gateway node. To begin, connect to the gateway via `ssh` and execute `ip` to inspect the configured network interfaces. The `bat0` interface is isolated here for clarity:

```
$ ip a
--snip--
bat0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 192.168.199.1  netmask 255.255.255.0  broadcast 192.168.199.255
--snip--
```

Notice that the `bat0` interface is configured with the gateway's static IP address, which you defined earlier. You should also see the `eth0` interface configured with an IP address assigned by your router.

Next, use `batctl` to check the status of neighboring nodes as seen by the gateway:

```
$ sudo batctl neighbors
[B.A.T.M.A.N. adv 2022.3, MainIF/MAC: wlan1/80:1f:02:9b:b0:c4
(bat0/5e:26:41:3e:fd:7e BATMAN_IV)]
IF            Neighbor              last-seen
wlan1         mesh-pi#2             0.930s
wlan1         mesh-pi#3             0.070s
wlan1         mesh-pi#4             0.610s
wlan1         mesh-pi#5             1.780s
wlan1         mesh-pi#6             0.820s
```

The example output shows that several new mesh nodes have been added (symbolic names have been mapped to their MAC addresses in */etc/bat-hosts*, as discussed earlier).

Now, verify the nodes in your mesh. Choose one of the `mesh-pi` nodes and connect to it via `ssh`. Then use `ip` to inspect its `bat0` interface, as you did with the gateway node:

```
$ ip a | grep bat0
bat0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
inet 192.168.99.133/24 brd 192.168.99.255 scope global dynamic noprefixroute bat0
--snip--
```

Notice that this node's `bat0` interface has been assigned an IP address, 192.168.99.133, from the range you defined in the gateway node's `dnsmasq` service. If the `bat0` interface doesn't have an IP address assigned, you can run `dhclient bat0` to request one. Make note of this address for a later step.

Next, using the `batctl` commands you explored earlier, try inspecting the `batman-adv` interface status and neighboring mesh nodes.

Up to this point, you've been accessing your nodes via the onboard wireless interface, `wlan0`, configured with `wpa_supplicant`. The wireless connection to your router provides internet access in this configuration. To verify connectivity within your mesh and to the wider internet via the gateway, you can disable this interface. First, list your configured wireless devices:

```
$ iw dev
phy#1
    Interface wlan1
        ifindex 5
        wdev 0x2
        addr 74:da:38:ed:5e:94
        type mesh point
        channel 1 (2412 MHz), width: 20 MHz (no HT), center1: 2412 MHz
        txpower 20.00 dBm
phy#0
    Interface wlan0
        ifindex 3
        wdev 0x1
        addr dc:a6:32:3d:ff:9d
        ssid Home-Router
        type managed
        channel 11 (2462 MHz), width: 20 MHz, center1: 2462 MHz
        txpower 31.00 dBm
```

Note that bringing down the interface you're currently connected to via `ssh` will cause your terminal session to freeze. Don't worry; this is expected. I'll demonstrate an alternative method to access your device remotely.

In the preceding output, you can see that the `wlan0` interface is connected to `Home-Router`. Execute the following command to bring down this interface:

```
$ sudo ip link set down dev wlan0
```

At this point, your `ssh` session will become unresponsive because your device is no longer connected to your home network. Reconnect to your gateway node via `ssh`, using the IP address you noted earlier in place of `192.168.99.133`:

```
$ ssh pi@192.168.99.133
pi@192.168.99.133 password:
--snip--
```

If your username is different, you'll also need to replace `pi` with the name of your user. If you've set up key-based authentication, as described in "Key-Based Authentication" on page 44, you should now be connected remotely to this node over the mesh network via `ssh`. To verify that the gateway is providing internet access via IP forwarding and NAT, you can perform a basic ping test:

```
$ ping nostarch.com
PING nostarch.com (104.20.17.121): 56 data bytes
64 bytes from nostarch.com (104.20.17.121): icmp_seq=1 ttl=47 time=21.9 ms
64 bytes from nostarch.com (104.20.17.121): icmp_seq=2 ttl=47 time=14.6 ms
--snip--
```

Interrupt the ping test with CTRL-C. You may also want to use `ip` again to check the status of this mesh node's interfaces:

```
$ ip a
--snip--
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN
    group default qlen 1000
    link/ether b8:27:eb:f9:39:a8 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast state DOWN
    group default qlen 1000
    link/ether b8:27:eb:ac:6c:fd brd ff:ff:ff:ff:ff:ff
5: wlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1532 qdisc mq master bat0 state UP
    group default qlen 1000
    link/ether 74:da:38:ed:5e:94 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::76da:38ff:feed:5e94/64 scope link
    valid_lft forever preferred_lft forever
6: bat0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN
    group default qlen 1000
    link/ether be:54:cb:4a:34:d8 brd ff:ff:ff:ff:ff:ff
    inet 192.168.99.133/24 brd 192.168.99.255 scope global dynamic noprefixroute
--snip--
```

Notice that the `eth0` and `wlan0` interfaces appear as `DOWN`, while the `wlan1` interface is `UP` and associated with the virtual `bat0` interface. Congratulations! Your node is successfully communicating on the mesh network, with internet connectivity provided by the gateway. While you're connected, you may also execute any of the `batctl` commands covered earlier to monitor the status of the mesh.

Managing Nodes with the Gateway

You may have noticed that the gateway is the only node that's directly accessible from your external network. The other mesh nodes remain hidden behind it. To access them, simply log in to the gateway node; you can then manage these nodes from the command line or access them directly via `ssh`. Bear in mind, however, that any external services that rely on network broadcast traffic, such as media servers or wireless printers, won't be passed through to the mesh network.

There are several ways you can survey nodes on the mesh, apart from using `batctl` and the `batman-adv` protocol. One method is to use the mesh

gateway's `dnsmasq` service to check the active DHCP leases. You can do this by inspecting the contents of the leases file, like so:

```
$ cat /var/lib/misc/dnsmasq.leases
1640415313 be:54:cb:4a:34:d8 192.168.99.133 raspberrypi 01:be:54:cb:4a:34:d8
--snip--
```

Recall that you've set leases to expire after 12 hours, so this may not accurately reflect all active nodes. Another handy utility for this scenario is `nmap` (or *network mapper*), a staple in every network administrator's toolbox. Install it and use it to explore your mesh network by executing the following commands:

```
$ sudo apt install nmap -y
$ sudo nmap -sn 192.168.99.0/24
Starting Nmap 7.80 ( https://nmap.org ) at 2021-12-24 20:18 GMT
Nmap scan report for 192.168.99.133
Host is up (0.0014s latency).
--snip--
```

Here, the `-sn` option tells `nmap` to list available hosts that respond to discovery probes, without performing a port scan. Often called a *ping scan*, this will provide you with a list of all the active IP addresses belonging to nodes on your mesh network. As a reminder, these services and tools operate at Layer 3 of the mesh. You can, of course, inspect the single-hop mesh neighborhood at the protocol (Layer 2) level with `sudo batctl neighbors` periodically as well, and compare the results.

`batman-adv` will proactively do all the dynamic packet routing and "dead node" detection in your mesh for you. You can monitor all aspects of the network by using the suite of tools available from `batctl`, including `tcpdump`, `traceroute`, and more, as well as analyzing logfiles and debug tables, if needed. Execute **`batctl -h`** for a complete list of options.

In most cases, a `batman-adv` wireless mesh will optimize itself without your intervention as nodes enter and leave the network. There are still ways you can tweak it to better suit your specific requirements, however. These are discussed in "Pointers on Fine-Tuning" on page 333.

A Closer Look at the Protocol

The `batman-adv` algorithm works by dispersing knowledge about the optimal end-to-end paths between nodes in the mesh across all participating nodes. Each node maintains information only about the best next hop to every other node, making global knowledge about network topology unnecessary. This might sound like voodoo, but the inner workings of the algorithm will become clearer as you explore it further.

On a practical level, each node in a `batman-adv` mesh periodically broadcasts a *hello* signal, also known as an *originator message*, or *OGM*, to inform neighboring nodes of its existence. An OGM consists of an originator address, a sender address, and a unique sequence number. When a node receives an OGM, it changes the sending address to its own address and

rebroadcasts the message according to specific rules. The sequence number is used to identify which of a pair of messages is newer. Through this process, each node in the network becomes aware of its own direct, or single-hop neighbors. At the same time, a node also learns about other nodes that aren't in range through a direct link but can be reached by hopping through a neighbor.

While node *A* is moving through the mesh, the route between *A* and *B* should recover as fast as possible. This process is illustrated in Figure 10-8, where *N1* and *N2* are intermediate nodes.

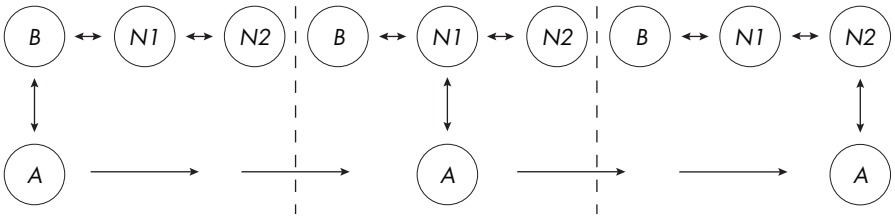


Figure 10-8: A visual representation of node mobility and hopping in a mesh network

OGMs that follow a path where the quality of wireless links is poor or where links are saturated will suffer from packet loss or delays on their way through the mesh. OGMs that travel along better routes will propagate faster and more reliably. The routing algorithm uses this information when selecting which neighbor to use as the next hop and updates its routing table accordingly.

Pointers on Fine-Tuning

Given this high-level view of how OGMs are used to build mesh routing tables, let's consider some scenarios in which you might tweak the network configuration to better suit your needs. To reduce the overhead of discovering all participants in the mesh, `batman-adv` can aggregate OGMs into a single packet rather than sending several smaller ones. This feature is enabled by default because it's helpful in the vast majority of setups. However, if you're operating `batman-adv` in a highly mobile environment (such as a fleet of vehicles, unmanned aerial vehicles, or robots), you may want to disable it, as it introduces a slight delay while updating the network. You can check and modify this setting as follows:

```
$ sudo batctl meshif bat0 aggregation
enabled
$ sudo batctl meshif bat0 aggregation 0
disabled
```

A related setting you can adjust for mobile environments is the *hop penalty*, which influences `batman-adv`'s preference for multihop versus direct routes. This value is applied to each forwarded OGM. A higher hop penalty makes it less likely that other nodes will choose this node as an intermediate

hop to any given destination, while a lower hop penalty may result in longer routes because retransmissions are not penalized.

The default hop penalty of 15 is a good balance for most mesh configurations and in most cases may be left as is. In mobile deployments, you might choose a higher value (up to a maximum of 255) to discourage other nodes from routing traffic through certain nodes. To view the current setting on any mesh node, run:

```
$ sudo batctl meshif bat0 hop_penalty
```

To change it, add the new value after the `hop_penalty` option. For example, to make sure a mobile node won't be chosen as a router, you might use:

```
$ sudo batctl meshif bat0 hop_penalty 255
```

As you saw in the previous section, a `batman-adv` node can operate in a gateway server mode in which it shares its internet connection with the rest of the mesh. One of the features of a gateway is its ability to announce its available internet bandwidth. The default gateway bandwidth is 10Mbps for downloads and 2Mbps for uploads. You can confirm this setting by running the following command on your gateway node:

```
$ sudo batctl meshif bat0 gw_mode
server (announced bw: 10.0/2.0 MBit)
```

As your mesh grows, it can be advantageous to configure more than one gateway. In these scenarios, gateways can adjust their announced bandwidth to more accurately represent their capabilities. To do this, enter the desired numbers separated with a forward slash (`/`), optionally followed by `kbit` or `mbit`):

```
$ sudo batctl meshif bat0 gw_mode server 20mbit/5mbit
```

The protocol will propagate these values throughout the mesh, allowing client nodes to select the best gateway based on criteria such as link quality and announced bandwidth.

These are a few of the common settings you can use to fine-tune your mesh network. Of course, `batman-adv` offers many more configuration options not covered here. If you'd like to make further adjustments to your mesh, a good practice is to establish a baseline by measuring latency between nodes (using the tools we've explored), then modify the settings on one or more nodes and observe how your metrics change.

For insights into additional mesh tuning parameters, visualized in real time, see "Using LED Activity Indicators" on page 336.

A Word on Security

One of the most common questions asked about `batman-adv`, and about wireless mesh networks in general, is: "How do I secure the mesh?" The short answer is, it depends on the type of security you need. Security is a broad

topic. Most often, it involves authentication, encryption, or (probably) some combination of both. The environment in which you deploy your mesh will also shape its security requirements. In real-world implementations, achieving strong security always involves trade-offs, and even the most secure systems usually have vulnerabilities. You might opt for making your community wireless mesh secure enough for most clients to join, or for use in an emergency, but not so bulletproof that the system becomes unusable. After all, the plug-and-play nature of mesh networks is one of their greatest selling points.

In the case of authentication, the lack of central infrastructure in an ad hoc wireless mesh presents special challenges when it comes to establishing trust between nodes. Contrast this to the public internet, where we can use a public key infrastructure (PKI) with certificates and keys to secure data transfers between a client device, such as a web browser, and a server.

In the context of community wireless projects, the lack of a central authority in a self-organizing mesh means that it's often simpler to forgo encryption at the Wi-Fi link layer and implement security at higher layers. This means using encrypted channels with tools such as `ssh` and `sftp` rather than insecure protocols like `telnet`, for example.

One proposed approach for mesh authentication is to use special public key certificates called proxy certificates, combined with a neighbor trust mechanism. Together, these allow authentication and access control to be managed in a secure manner.

Monitoring the Mesh Network

You should now have a minimum of two active mesh nodes in your network. You may also have configured a third node as a gateway to provide internet connectivity to the network and to give you an entry point to manage your mesh nodes. If you haven't set up a gateway, you can continue to provide WLAN connectivity to your nodes via the onboard wireless adapter's `wlan0` interface.

As noted in "Diagnosing Mesh Network Connectivity" on page 320, because `batman-adv` operates at Layer 2, traditional ICMP-based tools like `ping` and `traceroute` aren't able to provide their usual insights within the mesh. However, several other network tools can help you monitor the activity and overall health of your mesh nodes. These generally work by observing the flow of network packets transmitted and received over the mesh interface, then collecting and displaying various metrics. In addition to these terminal utilities, this section will demonstrate a monitoring technique that uses an optional hardware component and accompanying Python source code.

Using the Terminal

There are many command line tools you can use to monitor traffic over a `batman-adv` mesh interface. Several of the more popular utilities are described

here. The one you choose will largely depend on your specific network diagnostic goals, as well as personal taste.

One popular tool is `iftop`, a real-time bandwidth monitor for the terminal. It listens to network traffic on a specific interface, or on the first interface it can find if none is specified. By default, `iftop` counts all IP packets that pass through its filter and tracks their direction as they transit the interface. It then displays a summary of current bandwidth usage. To install and run `iftop` on the `bat0` interface, use the following commands:

```
$ sudo apt install iftop -y
$ sudo iftop -i bat0
```

After a few moments, you should see traffic between your node's `bat0` interface and other nodes in the mesh. Note the directional traffic indicators (`=>` and `<=`) between the hosts. To exit the monitor, press Q.

Another popular tool is `bmon` (bandwidth monitor), which can be used to monitor network traffic over a mesh interface and its associated wireless interface simultaneously. Install it with:

```
$ sudo apt install bmon -y
```

Then launch it, specifying the `bat0` and `wlan1` interfaces for monitoring:

```
$ bmon -p bat0,wlan1
```

You can use the up and down arrow keys to toggle between the interfaces. To exit `bmon`, press Q.

The final network monitor tool I'll mention is `slurm`. Install it with:

```
$ sudo apt install slurm -y
```

Then run it, specifying the `bat0` mesh interface with the `-i` option. The `-L` option enables software-based transmit/receive LED indicators:

```
$ slurm -i bat0 -L
```

Exit the program with a Q keystroke. In the next section, you'll implement a hardware-based monitoring technique.

Using LED Activity Indicators

Another way of visualizing how network traffic propagates through your mesh is by adding LED indicators to your nodes. This can be done with one or more Squid RGB add-ons and a dash of Python. As you saw in "Hardware Required" on page 311, this component provides a single LED with several fully controllable color channels.

Begin by connecting the Squid RGB to the GPIO header of the mesh node (or nodes, if you have multiple Squids) you want to monitor. If you're unsure of the GPIO pin numbering, execute **`pinout`** from the terminal for a handy reference. The black lead serves as a ground; connect this to one of

the GND pins on the header. The GND pin between GPIO pins 18 and 23 is most convenient, as it permits you to connect the Squid's leads in a series. Next, connect the color-coded leads as follows:

- Red Squid lead to GPIO 18
- Green Squid lead to GPIO 23
- Blue Squid lead to GPIO 24

With that done, access the node and install the necessary dependencies:

```
$ sudo apt install python3-rpi.gpio python3-pip python3-psutil
$ pip3 install psutil
```

If you haven't done so already, clone this chapter's companion GitHub repository to your home directory. Then change to the project directory and open *netactivity.py* in your editor:

```
$ cd $HOME
$ git clone https://github.com/wirelesscookbook/pi-mesh.git
$ cd pi-mesh
$ nano netactivity.py
```

The *netactivity.py* Python script uses the `Squid` class, which provides a convenient interface to the standard RPi.GPIO library. Each of the Squid RGB's three color channels is controlled by a separate GPIO pin. These pins and their corresponding colors are defined in the following code sample:

```
from squid import Squid, RED, GREEN, OFF
from time import sleep
import psutil
import time
import sys

# GPIO pins for RED, GREEN, BLUE channels (BCM numbering)
RED_PIN = 17
GREEN_PIN = 27
BLUE_PIN = 22

led = Squid(RED_PIN, GREEN_PIN, BLUE_PIN)

# Default network interface
interface = 'wlan1'

def flash(color):
    led.set_color(color)
    time.sleep(0.01)  # Flash duration
    led.set_color(OFF)
```

```
try:
    while True:
        net_io = psutil.net_io_counters(pernic=True)
        interface_io = net_io.get(interface)
--snip--
```

Inside the main `while` loop, the `psutil` (process and system utilities) Python package is used to fetch information on system utilization—in this case, network I/O, or input and output, statistics. By comparing the current number of bytes sent and received over the selected interface to their previous values, the script can flash the corresponding LED color to indicate activity. This is handled by `flash(GREEN)` and `flash(RED)`, respectively. A short `time.sleep()` interval is added so each flash is visible. Execute the script with:

```
$ python netactivity.py wlan1 &
```

You can omit the `wlan1` argument if you like, as this is the default interface. If your mesh interface is something other than `wlan1`, be sure to specify it here.

Appending an ampersand (`&`) to the command line instructs the shell to run the command as a background process. The shell will respond with the process ID (PID) of the running program, followed by a message indicating the monitored interface. Pressing ENTER will return you to the bash command prompt, allowing you to perform other operations while the Python script is running. At any time, you can use the `jobs` command to view a list of background jobs. Use `fg %n` to bring a background job (job number *n*) to the foreground. To exit the Python script, press CTRL-C.

The first thing you may notice upon running the script is that the LED flashes green (send, or transmit) on your mesh nodes at regular intervals. This activity corresponds to the *hello* originator messages sent to inform neighboring nodes of their presence, as discussed in “A Closer Look at the Protocol” on page 332. You can determine the current setting for this interval by running:

```
$ batctl meshif bat0 orig_interval
```

The default interval value is 1,000 ms (1 s). This is generally suitable for small, relatively stable mesh networks. In larger or highly dynamic networks (where nodes move frequently or change often), decreasing this interval can help the network adapt more quickly to topology changes. However, bear in mind that this also increases overhead. To adjust the interval, specify a new parameter value in milliseconds:

```
$ sudo batctl meshif bat0 orig_interval 500
```

After making this change, restart the *netactivity.py* script and check that the green send LED flashes to confirm that it has taken effect. As you tune your mesh, monitor the network's performance using `batctl` and the command line tools discussed in the previous section. A good rule of thumb is to make incremental, rather than drastic, changes while fine-tuning the network.

Troubleshooting

If a node isn't able to join your mesh network, start by verifying that the `batman-adv` module is loaded. Then check the installed version using `batctl`, like so:

```
$ lsmod | grep batman_adv
$ batctl -v
batctl debian-2020.4-2 [batman-adv: 2022.3]
```

The example output indicates that the installed `batman-adv` version is 2022.3. In practice, different minor versions usually interoperate without issues, but major version differences (for example, 2020.1 versus 2023.1) can cause protocol compatibility problems. To ensure a stable and reliable mesh, I recommend that all nodes run the same version of the protocol.

If you suspect a node has lost connectivity to the mesh, a best practice is to begin by conducting basic ping and `traceroute` tests. As mentioned in "Diagnosing Mesh Network Connectivity" on page 320, you'll need to use `batctl`'s versions of these tools. You can identify a node by its MAC address or hostname, using commands like the following:

```
$ sudo batctl ping node_mac_address
$ sudo batctl traceroute hostname
```

Make sure to substitute the actual MAC address or hostname of the node you want to test in place of the placeholder values. If your node responds to these tests, you can further examine your mesh with these `batctl` diagnostics:

```
$ sudo batctl neighbors
$ sudo batctl originators
$ sudo batctl gateways
```

As a next step, you can check the global translation table with:

```
$ sudo batctl transglobal
```

This table maps client MAC addresses to the appropriate mesh nodes and ensures that packets are correctly routed to their destination nodes within the network. This routing information is especially vital in dynamic environments where nodes frequently change position. If your mesh is highly dynamic, consider tuning the `orig_interval` parameter value as described in "Using LED Activity Indicators" on page 336.

All `batman-adv` error messages, warnings, and information messages are written to the kernel log. To see these messages in the system logs, use the following commands:

```
$ journalctl -xe | grep batman
$ dmesg | grep batman
```

As a last measure, restarting a node will often permit it to rejoin the network. The mesh node startup scripts described in “Running at Boot” on page 323 are useful here.

Going Further

In the previous sections, you looked at several methods to verify and monitor connectivity between nodes in your mesh network, including optionally integrating an LED activity indicator. In terms of where to go from here, the direction you choose will be guided by your (or your future mesh community's) specific requirements, goals, and ambitions. Mesh networks afford many unique possibilities that set them apart from typical Wi-Fi networks. To provide some inspiration, I'll describe a few popular use cases drawn from existing mesh implementations.

The Freifunk Paderborn project, discussed at the beginning of this chapter, is one of the best-known and most successful community-based mesh networks worldwide. This project, which is part of the larger Freifunk (“free Wi-Fi”) initiative in Germany, has the egalitarian aim of providing free and open internet access to its members and visitors. The project's open source ethos has inspired and enabled the growth of mesh-based free wireless connectivity in many other regions. Organized in a grassroots manner, local communities have formed their own networks and connected them together with wireless backbones, with uplinks to the wider internet established at several locations via secure VPN tunnels. A real-time map of one of these Freifunk community networks, connecting the municipalities of Mainz, Wiesbaden, and Umgebung, is shown in Figure 10-9.

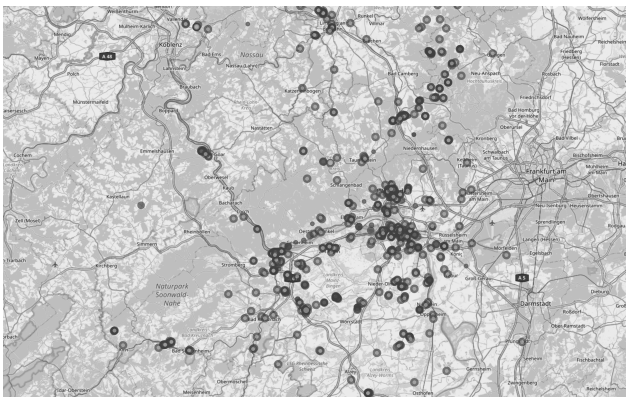


Figure 10-9: A real-time map of Freifunk community mesh nodes (OpenStreetMap, CC BY-SA 2.0)

As of this writing, more than a thousand mesh nodes are participating in this Freifunk region alone. Across Germany as a whole, this number exceeds many tens of thousands of nodes. All the devices in Freifunk operate as 802.11s mesh nodes using the same `batman-adv` protocol you've implemented here. Connecting to the network is done simply by flashing a supported router or access point with the Freifunk mesh firmware and powering it up. The device will then automatically mesh with other Freifunk router nodes that are within range.

Beyond free Wi-Fi, the Freifunk network provides many services to its participants, including chat via IRC and Mumble, radio and podcasts, collaborative writing, community calendars, and more. Perhaps most interestingly, the Freifunk Community API provides a mechanism for each community to make its resources known in a structured way. Powered by this API, a map of active Freifunk communities across Germany is available at *https://api-viewer.freifunk.net*.

The Freifunk initiative is an exceptional success story highlighting the potential of community-owned mesh networks. While a Raspberry Pi–based mesh network operating at this scale is theoretically possible, dedicated router hardware is generally better suited for this purpose. That said, I certainly encourage you to expand the `pi-mesh` network you've implemented here. To make it easier for others to join your `pi-mesh`, consider creating a baseline configuration based on the example in this chapter, then cloning the OS to create your own custom image. You can then distribute this image to other users so that they can more easily participate in the network.

If you're intrigued by the visual LED status indicators but find them lacking, the recipe from Chapter 3 can be modified to output any number of metrics associated with your mesh nodes. For example, you could parse the output of `batctl` to obtain a list of neighbors, originators, and gateway nodes within range of your node and output this to the TFT display. By querying these values at regular intervals, you can experimentally observe the process of hopping as nodes transit through the mesh.

Wrapping Up

In marked contrast to most of the other Wi-Fi networks covered in this book, this chapter has focused on implementing wireless mesh nodes that rely on an ad hoc routing protocol. The approach you followed uses low-cost, readily available components to create a mesh network that's both resistant to disruption and capable of scaling up as more nodes are added.

To achieve this, you interacted with network interfaces and the mesh protocol directly via the shell, and you used bash scripts from the companion GitHub repository to automate joining nodes to the mesh at system startup. Throughout this process, you saw how the mesh protocol operates entirely at Layer 2 of the OSI model, effectively emulating a virtual network switch to route traffic between nodes. You also learned about the vital role gateways play in providing an internet uplink to your network, as well as

serving as an entry point for managing nodes in the mesh. To facilitate this, you used the mesh protocol and DHCP to designate a gateway node.

You examined the protocol itself to learn how messages are propagated between nodes, and you saw how nodes use this information to discover neighbors and route traffic efficiently. With this understanding of how mesh routing tables are created, you explored techniques for fine-tuning your mesh network. Finally, you looked at various methods for monitoring your mesh network, both via the terminal and by using hardware LED activity indicators.