Mesh Topology and Path Analysis

Knowing *how* your messages travel through the mesh is essential for identifying bottleneck nodes, optimizing relay placement, and diagnosing delivery failures. Meshtastic exposes hop-count information in every received packet, and community tools like meshmap.net provide geographic visualization of the whole network. This page explains how to read that data and use it to understand your mesh's topology.

How Meshtastic Routes Messages

Meshtastic uses a *managed flood* routing algorithm (not a distance-vector or link-state protocol). When a node transmits a packet:

1. The originating node sets the `hop_limit` field (default 3, maximum 7).
2. Every node that receives the packet and has not seen this packet ID before will relay it after a random back-off delay, decrementing `hop_limit` by 1.
3. A node will *not* relay a packet whose `hop_limit` has reached 0.
4. Each node keeps a recent-packet cache (typically 256 entries) to drop duplicates it has already forwarded.

This means there is no explicit source-routed path - messages can arrive via multiple routes simultaneously. The first copy to arrive is delivered; subsequent copies are dropped by the destination's duplicate cache.

Reading Hop Count in Received Packets

When a node receives a packet the `hop_start` and current `hop_limit` fields tell you how many hops the packet took:

```
hops_taken = hop_start - hop_limit
```

For example, if `hop_start = 3` and the received `hop_limit = 1`, the packet took 2 hops to reach you. In the Meshtastic mobile app, the node detail view shows "Hops Away" which is this pre-computed value. A value of 0 means the node is a direct neighbor; 1 means one relay was used; and so on.

In the Python CLI `--listen` output, the raw `hop_limit` of each arriving packet is printed. Since the original `hop_start` is stored in the packet, you can compute hops taken from any arriving packet.

Traceroute Equivalent: meshtastic --traceroute

Meshtastic has included a built-in Traceroute Module since firmware 2.0.8 that discovers the actual relay path to a destination node. (Recording the full return route *and* the per-link SNR shown below requires firmware 2.5 or newer; on older firmware only the outbound path is reported.)

```
# Syntax: meshtastic --traceroute '!<destination_node_id>'
meshtastic --traceroute '!aabbccdd'
```

Example output:

```
Sending traceroute request to !aabbccdd (this could take a while)
Route: !11223344 → !55667788 → !99aabbcc → !aabbccdd
 !11223344 WB5ABC (this node)
 !55667788 K5ZZZ SNR: 3.25 dB
 !99aabbcc W5RELAY SNR: -2.50 dB
 !aabbccdd N5XYZ SNR: 1.75 dB
```

Each arrow represents one relay hop. The SNR shown next to each node is the signal quality measured at that receiving node for the link it just traversed; on firmware 2.5+ the return path is recorded the same way, so you get per-link SNR for both the outbound and return legs. Lower (more negative) SNR on a leg indicates a weaker, less reliable link. How negative is "too weak" depends on the spreading factor: the decode floor ranges from about −7.5 dB at SF7 to about −20 dB at SF12. On the default LongFast preset (SF11, floor around −17.5 dB) only links worse than roughly −14 dB are marginal, so a leg at, say, −7 dB on LongFast still has comfortable margin.

Important caveats for traceroute:

- The traceroute packet itself consumes channel airtime, so avoid running it repeatedly on a congested mesh.

- The path shown is the path taken by the traceroute probe - actual message traffic may take a different path because nodes relay whichever copy arrives first.
- Traceroute requires the destination node to be online and reachable.

Visualizing the Network with meshmap.net

meshmap.net is a community-maintained map that ingests position and neighbor-info telemetry from Meshtastic nodes that have MQTT upload enabled. It shows:

- Node locations on a geographic map.
- Neighbor links drawn between nodes that have directly heard each other (neighbor-info packets).
- Signal quality (SNR) color-coded on each link.
- Node metadata: firmware version, hardware model, last seen time.

To appear on meshmap.net your node must:

1. Have GPS or a fixed position configured.
2. Have MQTT enabled pointing to `mqtt.meshtastic.org` or a community server that feeds into the map.
3. Have the *neighbor-info module* enabled so it reports detected neighbors.

Privacy note: enabling MQTT to the public broker so your node appears on meshmap.net also publishes your node's position and its neighbor list (which nodes you can hear) to the public internet. Consider whether that exposure is acceptable for your deployment before opting in.

Neighbor Info Module

The neighbor-info module (enabled in Settings → Modules → Neighbor Info) causes each node to periodically broadcast a list of all nodes it has directly heard, along with the SNR of the last packet from each. This data is the foundation of topology visualization:

```
meshtastic --get neighbor_info
# Shows:
# neighbor_info {
#   enabled: true
#   update_interval: 21600 # seconds between neighbor broadcasts (default 6 hours)
```

```
# }
```

The neighbor-info update interval **cannot be set lower than 4 hours (14400 seconds)** — the firmware rejects shorter values to protect channel utilization. The default is **6 hours (21600 seconds)**. Neighbor Info reports SNR only (no per-neighbor RSSI). Do not set it to short intervals such as 900 seconds; that value is invalid and will not take effect.

Identifying Bottleneck Nodes

A bottleneck node is one that a large fraction of the mesh depends on for connectivity. To identify them:

1. **On meshmap.net**: look for nodes with many link lines converging on them. A hub node with 6+ direct neighbors that are otherwise isolated from each other is a critical single point of failure.
2. **Via traceroute**: run traceroute to several different destination nodes. If the same relay node ID appears in every path, that node is a bottleneck.
3. **Via --nodes SNR analysis**: if several nodes only have high SNR to one other node (their only relay), that relay is a bottleneck.

Mitigation strategies for bottleneck nodes:

- Deploy a second relay node with overlapping coverage to provide redundancy.
- Move the bottleneck node to a higher elevation to increase its coverage radius.
- If — and only if — the bottleneck node is genuinely well-sited (excellent RF location, high elevation), consider configuring it as ROUTER role so the firmware prioritizes its rebroadcast duty. Note that on ESP32 boards the ROUTER role automatically enables power-saving sleep (it cannot be turned off) and disables BLE/WiFi/Serial by default — it does not stay awake more. Do not promote a poorly-sited node to ROUTER; add a second relay instead.

Understanding Hop Count in Received Packets - Practical Example

```
import meshtastic
import meshtastic.serial_interface
```

```
from pubsub import pub

def on_receive(packet, interface):
 hop_start = packet.get("hopStart", 0)
 hop_limit = packet.get("hopLimit", 0)
 hops_taken = hop_start - hop_limit
 sender = packet.get("fromId", "unknown")
 print(f"Packet from {sender}: {hops_taken} hop(s) away, hop_limit_remaining={hop_limit}")

pub.subscribe(on_receive, "meshtastic.receive")
iface = meshtastic.serial_interface.SerialInterface()
input("Listening... press Enter to exit
")
iface.close()
```

Running this script while observing mesh traffic gives a live view of hop distances for every packet, which is invaluable for understanding how your network topology performs in practice.

Revision #3
Created 2026-05-03 05:39:38 UTC by Mesh America Admin
Updated 2026-06-09 02:00:14 UTC by Mesh America Admin