Network Diagnostics and Health Monitoring

Tools and techniques for understanding the health of your Meshtastic mesh.

- Reading Network Statistics in the Meshtastic App
- Using the Meshtastic Python CLI for Diagnostics
- Mesh Topology and Path Analysis
- Common Network Problems and Solutions
- Building a Meshtastic Network Map

Reading Network Statistics in the Meshtastic App

Understanding what your Meshtastic network is actually doing requires knowing how to read the statistics the app surfaces. The mobile app (Android and iOS) and the web client all expose channel utilization, airtime, packet counters, and radio-level metrics. This page explains every number, what healthy ranges look like, and how to spot a congested network before it starts dropping messages.

Where to Find Network Statistics

Android App

Open the Meshtastic Android app and open a node's **Device Metrics** (telemetry) view — tap a node in the node list, then open its Device Metrics / telemetry panel. Channel utilization and air-utilization metrics are shown there (the live stats live in the telemetry view, not in *Settings → Radio Configuration → Device*, which is the configuration editor). Individual node signal metrics (SNR and RSSI) are shown in the node list by tapping any node row and selecting *Node Info*.

iOS App

Tap the **Nodes** tab, then tap any node to see its detail card. Signal strength, last heard time, hop count, and telemetry battery readings are all shown here. The channel utilization percentage is displayed on the **Mesh** tab's header area when connected to a local node.

Web Client (meshtastic.local or IP)

On the web client served directly from the device, the **Dashboard** section shows live channel utilization, air utilization, and packet statistics. This is the most detailed view and updates periodically as new telemetry arrives.

Channel Utilization %

Channel utilization measures what fraction of airtime on the LoRa channel has been consumed by transmitted packets, averaged over a rolling **1-minute window**. The firmware computes it by summing on-air milliseconds across six 10-second sub-windows. (This is a separate metric from the per-hour duty cycle, which is the regulatory airtime percentage discussed below.)

Utilization Range	Network Health	Interpretation
0 - 25%	Healthy (green)	Plenty of spare capacity. All traffic should get through. This is the firmware's green/throttle boundary — above ~25% the firmware begins delaying transmissions.
25 - 50%	High (orange)	Collisions increasing. The firmware delays TX above ~25%; routers cut back telemetry around 40%. Expect occasional packet loss.
50 - 100%	Congested (red)	Severe contention. Most non-acked packets will be lost.

The underlying metric is computed by the device firmware using the LoRa modem's receive statistics. Every received or transmitted packet's on-air time is summed and divided by the window period. The default LongFast preset uses SF11/BW250/CR4/5; a small (~50-byte) packet at this configuration takes roughly **400–700 ms** on air (the exact figure depends on payload size and preamble length). As an illustrative estimate, that means only a handful of such packets per minute will push utilization toward 25% — a level easily exceeded on a busy community mesh. (Note: SF11/BW250 LongFast is much slower than shorter presets — for example MediumFast at SF9 transmits roughly 3× faster than LongFast at SF11.)

Air Utilization

Air utilization is closely related but specifically counts the fraction of time the radio was *actively transmitting* (TX duty cycle). Some regulators impose per-hour duty-cycle limits (for example the EU 868 MHz sub-bands have 1% or 10% duty-cycle limits under ETSI EN 300 220). **These per-hour duty-cycle limits apply to EU 868 MHz sub-bands.** The US/Canada 902–928 MHz band (47 CFR 15.247 / ISED RSS-247) imposes **no duty-cycle limit** on digital-modulation LoRa systems — the governing limits there are conducted power (1 W / 30 dBm), EIRP, and frequency-hopping/dwell rules, not an airtime percentage. So in the US/Canada, watching air utilization is about avoiding mesh congestion, not staying under a legal airtime cap. Where a regional duty-cycle limit does apply, the firmware will delay outgoing packets to help stay compliant; if you see air utilization climbing, reducing your transmit frequency reduces congestion either way.

Packet Counters

The firmware maintains several packet counters that help diagnose forwarding behavior:

- **Packets Rx**: Total valid LoRa packets received (passed CRC check).
- **Packets Rx Bad**: Packets received with a failed CRC. A high ratio of bad packets to good packets indicates RF interference or a node with a damaged antenna.
- **Packets Tx**: Packets this node has transmitted (original or forwarded).
- **Packets Tx Relay**: Subset of Tx that were relayed on behalf of another node. A router node should have a high relay ratio. A client node with a very high relay count may be flooded because its hop limit is too generous.
- **Packets Rx Duplicate**: Packets seen more than once. Some duplication is normal in mesh (same packet arrives via multiple paths); more than 10 - 20% of total Rx suggests a routing loop or misconfigured hop limits.

SNR - Signal-to-Noise Ratio

SNR is the most important single-number indicator of link quality. It is reported in **decibels (dB)** and expresses how far above the noise floor the received signal sits. LoRa can decode packets at negative SNR values because spread-spectrum processing gain allows it to recover signal from noise. **The decode floor is spreading-factor-dependent**, so the bands below are rules of thumb that shift with your preset (see the note after the table).

SNR (dB)	Link Quality	Notes
≥ 5	Excellent	Strong link (rule of thumb). Consider a lower spreading factor for shorter air time.
0 to 5	Good	Reliable communications. Typical for in-range nodes.
−5 to 0	Marginal	Some packet loss possible at low SF; still well within the decode floor on long presets.
−10 to −5	Weak (low SF) / usable (long presets)	On short presets (low SF) packet loss begins here; on the default LongFast (SF11) preset this range still decodes reliably.
< −10	SF-dependent	Below the floor at low SF, but on LongFast (SF11/BW250) links down to about −15 dB are still usable and the floor is around −17.5 dB. Only near each preset's own floor will most packets be lost.

The minimum decodable SNR depends on the spreading factor: the floor ranges from approximately −7.5 dB at SF7 to about −20 dB at SF12 (LongSlow), stepping roughly −2.5 dB per SF (per the SX1276 datasheet / Semtech AN1200.22). The default LongFast preset uses SF11/BW250, giving a floor around −17.5 dB — so a −12 dB link on LongFast is healthy, not weak. Interpret the table above relative to your configured preset.

RSSI - Received Signal Strength Indicator

RSSI measures the absolute power level of the received signal in **dBm** (decibels relative to 1 milliwatt). A more negative number means a weaker signal. Unlike SNR, RSSI alone does not tell you whether a packet can be decoded - a −120 dBm signal in a quiet environment may decode fine while a −100 dBm signal swamped by noise will not. The bands below are rules of thumb, not published Meshtastic specifications.

RSSI (dBm)	Interpretation
−60 to −80	Very strong (rule of thumb). Nodes are physically close or have excellent antennas.
−80 to −100	Normal operational range for most deployments.
−100 to −115	Weak but decodable at high spreading factors.
< −120	Conservative low-signal threshold. Note the datasheet sensitivity floor is preset-dependent and can reach below −140 dBm at high SF/narrow BW, so −120 dBm is a cautious cutoff rather than an absolute floor.

What Healthy Numbers Look Like

A well-configured community mesh with 10 - 30 nodes should exhibit:

- Channel utilization consistently below ~25% (the firmware's green/throttle boundary).
- SNR between −5 and +10 dB on all primary relay links (and remember the usable floor is lower on long presets).
- RSSI above −115 dBm on relay links.
- Duplicate packet ratio below 10%.
- Zero or near-zero "Rx Bad" packets (CRC failures) unless there is known interference.

Spotting a Congested Network

The most common congestion warning signs:

1. **Rising channel utilization**: If utilization creeps above 25% during normal operation (not an emergency event), the first remediation step is to increase the position broadcast interval for all nodes. The default 15-minute interval is reasonable for a small network but too aggressive for a mesh with 50+ nodes.
2. **High duplicate count**: More than 20% duplicates often indicates that too many nodes are configured as ROUTER or that hop limits are set too high, causing the same packet to traverse the network redundantly.
3. **Increasing Rx Bad rate**: If the ratio of CRC-failed packets is rising, a hidden-node problem or external interference source (other LoRa devices on the same frequency, industrial equipment) is likely the cause.
4. **Messages not delivered even to nearby nodes**: When a close-range node shows good SNR and RSSI but messages are still failing, the channel is likely saturated and ACKs are being lost.

Node Telemetry Metrics

Each node periodically transmits a *DeviceTelemetry* packet containing battery voltage, battery level percentage, and (if available) temperature. In the node list you can see these values next to each node. A battery level below 20% combined with low SNR from that node often means the node is running on backup power after a mains failure - an important operational signal in an emergency mesh.

Using the Meshtastic Python CLI for Diagnostics

The Meshtastic Python package ships both a library and a command-line interface (CLI). The CLI is the fastest way to interrogate a connected node, export its configuration, watch live packet traffic, and run one-off diagnostic commands from a laptop. This page covers installation, every useful diagnostic command, how to read the output, and the difference between serial and TCP connections.

Installation

```
# Requires Python 3
pip3 install --upgrade "meshtastic[cli]"

# Verify installation
meshtastic --version
# Output: meshtastic v2.x.x
```

On Linux you may need to add your user to the `dialout` group to access serial ports without sudo:

```
sudo usermod -aG dialout $USER
# Log out and back in for the change to take effect
```

Connecting to a Node

Serial (USB)

The most common connection method. Plug the device in via USB and run any command without extra flags - the CLI auto-detects the first available serial port:

```
meshtastic --info
```

To specify a port explicitly:

```
meshtastic --port /dev/ttyUSB0 --info # Linux
meshtastic --port /dev/cu.usbserial-* --info # macOS
meshtastic --port COM4 --info # Windows
```

TCP (Wi-Fi)

Nodes running ESP32 with Wi-Fi enabled expose a TCP port (default 4403). Use `--host` instead of `--port`:

```
meshtastic --host 192.168.1.42 --info
meshtastic --host meshtastic.local --info # mDNS name on the local LAN
```

BLE

BLE support is included by default — `bleak` is now a core dependency of the `meshtastic` package, so the old `[ble]` extra is no longer required:

```
pip3 install --upgrade "meshtastic[cli]"
meshtastic --ble-scan # lists visible Meshtastic BLE devices
meshtastic --ble <MAC-or-name> --info
```

--info: Device Summary

```
meshtastic --info
```

This is the single most useful diagnostic command. It prints a comprehensive summary of the connected node's radio configuration and node database (largely human-readable text with some structured sections), including:

```
Connected to radio
Owner: WB5ABC (4 char id: !aabbccdd)
My info:
{
 "myNodeNum": 2864434397,
 "hasGps": true,
```

```
 ...
}
Metadata: {
 "firmwareVersion": "2.5.3.abcdef",
 "deviceStateVersion": 23,
 "hasWifi": true,
 "hasBluetooth": true,
 "hasEthernet": false,
 "hwModel": "TLORA_V2_1_1P6",
 "hasRemoteHardware": false,
 "canShutdown": false
}
Nodes in mesh: {
 "!aabbccdd": {
 "num": 2864434397,
 "user": { "id": "!aabbccdd", "longName": "WB5ABC", "shortName": "W5", "hwModel":
"TLORA_V2_1_1P6", "role": "CLIENT" },
 "position": { "latitudeI": 323456789, "longitudeI": -970123456, "altitude": 312, "time":
1714000000 },
 "snr": 6.5,
 "lastHeard": 1714010000,
 "deviceMetrics": { "batteryLevel": 87, "voltage": 3.98, "channelUtilization": 4.25,
"airUtilTx": 0.81 }
 },
 ...
}
Preferences: { ... full radio config (role, positionFlags, etc. live here) ... }
Channels: [ { "index": 0, "role": "PRIMARY", "settings": { ... } }, ... ]
```

Note that the fields live in different objects: device capabilities and firmware version are under `Metadata`, while the node's `role` and `positionFlags` live in the radio config / `Preferences` (localConfig), not in `Metadata`. Script your lookups against the correct object.

Key fields to check during diagnostics:

- `channelUtilization`: percentage of channel airtime used over the last ~1 minute (the firmware sums on-air time across six 10-second sub-windows).
- `airUtilTx`: TX duty cycle percentage.
- `snr`: last-heard SNR from each remote node (dB).
- `lastHeard`: Unix timestamp. Subtract from current time to see how stale a node entry is.
- `firmwareVersion`: compare against the latest release to check if updates are needed.

--nodes: Node Table

```
meshtastic --nodes
```

Prints a compact tabular summary of all nodes in the mesh database:

```
 N Num User AKA Hardware Latitude Longitude Altitude Battery SNR LastHeard
1 2864434397 WB5ABC W5 TLORA_V2 32.3456 -97.0123 312m 87% 6.5 2024-04-25 10:31:22
2 3109276812 K5ZZZ K5 RAK4631 32.3512 -97.0099 289m 72% 2.1 2024-04-25 10:29:55
3 4012345678 N5XYZ N5 HELTEC_V3 32.3390 -97.0210 278m -- -4.7 2024-04-25 09:15:10
```

The `LastHeard` column quickly shows stale nodes (entries in the node DB that haven't been heard in hours or days). These are candidates for manual removal if you are troubleshooting ghost-node problems. The SNR column shows the signal quality of the last packet heard *from* that node (not necessarily a direct link - the packet may have been relayed).

--export-config: Full Configuration Export

```
meshtastic --export-config > node_backup_$(date +%Y%m%d).yaml
```

Exports the complete device configuration as YAML. This includes radio settings, channel definitions, module configs (telemetry intervals, MQTT settings, serial module, etc.), and device metadata. Use this to:

- Back up a node before a firmware update.
- Clone configuration from one device to another with `meshtastic --configure config.yaml`.
- Audit configuration differences between nodes in a community mesh.

Treat the exported file as secret. The YAML contains channel PSKs and any MQTT username/password in recoverable form. Store the backup encrypted and do not commit it to a public repository.

--listen: Live Packet Watch

```
meshtastic --listen
```

Subscribes to all decoded packets from the connected node and prints them as they arrive. This is the equivalent of a packet sniffer for the mesh. Press `Ctrl-C` to stop. The CLI prints library log lines and packet dictionaries; the simplified example below illustrates the key fields you will see rather than the literal output format:

```
TEXT_MESSAGE_APP: from=!aabbccdd, to=^all, id=0x12345678, hop_limit=3, want_ack=False
 payload: b'Hello from WB5ABC'

POSITION_APP: from=!aabbccdd, to=^all, id=0xdeadbeef, hop_limit=3, want_ack=False
 payload: Position(latitudeI=323456789, longitudeI=-970123456, altitude=312, time=1714010500)

TELEMETRY_APP: from=!ccccdddd, to=^all, id=0x87654321, hop_limit=1, want_ack=False
 payload: Telemetry(deviceMetrics=DeviceMetrics(batteryLevel=72, voltage=3.91,
 channelUtilization=11.3, airUtilTx=2.1))
```

During a `--listen` session, watch for:

- Packets arriving with `hop_limit=0`. `hop_limit` is the number of *remaining* hops: it is set to the configured hop count when the packet is sent and is decremented by one at each relay. A `hop_limit` of 0 on arrival means the packet can no longer be rebroadcast — it used all of its allowed hops, so the hop budget may be too low.
- Repeated identical packet IDs arriving within seconds (duplicate storm).
- High-frequency position packets from a single node ID indicating that node's smart position interval is too aggressive.

Reading Telemetry from --listen

Every 30 minutes (1800 s) by default each node broadcasts device telemetry. While listening you can redirect output and grep for telemetry to build a quick health log:

```
meshtastic --listen 2>&1 | grep TELEMETRY_APP | tee mesh_telemetry.log
```

Practical Diagnostic Workflow

1. **Start with `--info`** to get baseline node count and channelUtilization. If utilization > 25%, immediately check individual node position intervals.
2. **Run `--nodes`** and flag any node where `LastHeard` is more than 2 hours old during an active session. These are likely ghost nodes.

3. **Run `--listen` for 5 - 10 minutes** during peak mesh activity to count duplicate packet IDs and identify high-frequency transmitters.
4. **Export config with `--export-config`** and compare `hop_limit` across nodes. A community mesh should generally use a maximum hop limit of 3 (the default).

Serial vs TCP: When to Use Each

Method	Best For	Limitations
Serial (USB)	Local bench testing, initial setup, firmware examination	Requires physical access; occupies USB port preventing simultaneous app connection
TCP (Wi-Fi)	Remote diagnostics on deployed nodes, scripted automation, Raspberry Pi gateways	Requires node to have Wi-Fi enabled and be on local network or accessible via port forward
BLE	Temporary field diagnostics without Wi-Fi	Short range, platform-specific BLE stack issues, slower data rate

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.

Common Network Problems and Solutions

Meshtastic networks are remarkably self-organizing, but they are not self-diagnosing. When the mesh stops working well - messages drop, nodes disappear, delivery becomes unreliable - there are a small number of root causes that account for the vast majority of problems. This page covers the four most common issues: high channel utilization, ghost nodes, routing loops, and clock skew, along with specific remediation steps for each.

Problem 1: High Channel Utilization

Symptoms

- Channel utilization percentage consistently above 25%.
- Messages sometimes fail to deliver even between nearby nodes.
- The node list shows many nodes as "last heard" minutes ago even though they are known to be online.
- ACKs are not received for sent messages despite short distances.

Root Cause

The main drivers of channel utilization are **position, nodeinfo, and telemetry broadcasts**. Every node broadcasts its GPS position on a smart interval (minimum interval) and an update interval, alongside periodic NodeInfo and telemetry. On a large mesh with 30+ nodes all broadcasting position every 15 minutes, the channel quickly fills. Each position packet is ~50 bytes; at LongFast (SF11/BW250/CR4-5) this is roughly 400 - 700 ms of airtime (use a LoRa time-on-air calculator for the exact value at your payload size). With 30 nodes broadcasting every 900 seconds, position alone consumes on the order of 30 * 0.5 / 900 ≈ 1.7% airtime continuously for a single transmission each - and that understates the real load, because each broadcast is rebroadcast by multiple nodes across the mesh, and additional overhead (ACKs, route discovery, telemetry) multiplies the effective utilization.

Solutions

1. **Increase minimum and maximum position broadcast intervals.**

   In the Meshtastic app: Settings → Radio Configuration → Position → *Broadcast Interval*. For a community mesh where nodes are mostly stationary, intervals of 30 - 60 minutes are appropriate. Mobile nodes can use smart position which triggers a broadcast only when the node has moved more than a threshold distance.

   ```
   meshtastic --set position.position_broadcast_secs 3600
   meshtastic --set position.position_broadcast_smart_enabled true
   meshtastic --set position.gps_update_interval 120
   ```

2. **Disable unnecessary telemetry modules.**
   Environment telemetry (temperature, humidity) and power telemetry broadcast at their own intervals. If sensors are not connected, disable the modules to stop unnecessary transmissions. Note: setting an update interval to 0 does *not* disable telemetry - a value of 0 reverts to the 30-minute (1800s) default. To actually stop the broadcasts, disable the measurement:

   ```
   meshtastic --set telemetry.environment_measurement_enabled false
   meshtastic --set telemetry.power_measurement_enabled false
   ```

3. **Reduce hop limit on client nodes.**
   Most messages on a well-designed mesh reach their destination in 1 - 2 hops. Reducing the hop limit lowers the number of times a packet is relayed across the mesh, reducing airtime. Tune carefully: setting it too low can cut off nodes that genuinely need 3 hops to reach the rest of the network.

   ```
   meshtastic --set lora.hop_limit 2
   ```

4. **Switch to a lower-duty-cycle modem preset for dense networks.**
   The MediumFast preset (SF9/BW250) offers significantly shorter air time per packet at the cost of reduced range. For a dense urban mesh where all nodes are within 2 - 3 km of a relay, this trade-off is often worthwhile.

Problem 2: Ghost Nodes

Symptoms

- The node list contains nodes with "Last Heard" timestamps hours or days in the past that never update.

- The node count shown in the app is higher than the number of known active devices.
- Sending a message to a ghost node never gets ACKed.

Root Cause

Meshtastic stores a local database of every node it has ever heard. Nodes that leave the area, run out of battery, or are turned off permanently remain in the database indefinitely until manually cleared or the database fills and the oldest entry is evicted. These stale entries are "ghost nodes."

Ghost nodes are not just a cosmetic issue. On a network with strict channel utilization budgets, any mechanism that sends directed packets to ghost nodes (e.g., automated pings or position requests) wastes airtime. Additionally, some firmware versions attempt to route through recently-known paths, which can cause messages to fail if those paths included a now-offline ghost.

Solutions

1. **Remove ghost nodes via the CLI:**

   ```
   # Remove a single node by its node ID. Replace !deadbeef with the
   # target node's actual ID (the !-prefixed hex shown in the node list):
   meshtastic --remove-node !deadbeef

   # Clear the entire node database (nuclear option)
   meshtastic --reset-nodedb
   ```

   Caution: `--reset-nodedb` removes all known nodes including active ones. The mesh will rebuild the database over the next few hours as nodes broadcast again. Only use this when the database is severely polluted.
2. **Remove ghost nodes via the app:**
   In the Android app, long-press a node in the node list and select *Remove from node list*. iOS uses a swipe-left gesture on the node row.
3. **Note on node-info broadcast cadence:**
   The `device.node_info_broadcast_secs` setting controls how often *this* node broadcasts its own NodeInfo (default 10800s / 3 hours, minimum 3600s). It does *not* configure automatic eviction of other nodes - there is no documented "not heard within this window, auto-evict" setting. Remove stale entries manually via the app or `--remove-node` as above.

   ```
   # Adjust how often THIS node advertises its NodeInfo (does not evict others):
   meshtastic --set device.node_info_broadcast_secs 10800
   ```

Problem 3: Routing Loops

Symptoms

- Channel utilization spikes suddenly and stays high.
- The `--listen` output shows the same packet ID appearing many times from many different node IDs in rapid succession.
- A high proportion of received packets are duplicates of ones already seen.
- Battery drain on relay nodes accelerates noticeably.

Root Cause

Meshtastic uses managed flooding: before rebroadcasting, a node listens briefly to see whether another node has already rebroadcast the packet, and it skips packets it has recently seen (tracked in a packet history). Each hop also decrements the hop limit, which bounds how long a packet can live. Apparent "loops" - the same packet circulating repeatedly - usually trace back to:

- **Congestion / collisions**: when the channel is busy, a node may not hear the earlier rebroadcast it would otherwise have suppressed against, so it rebroadcasts anyway.
- **Misconfigured infrastructure roles**: too many always-rebroadcasting nodes in one area increases redundant transmissions.
- **Firmware bug**: certain firmware versions had rebroadcast-suppression bugs that caused excess relaying. Updating firmware is the primary fix.

Solutions

1. **Update to the latest stable firmware.**
   Many loop-related bugs have been fixed in 2.3+ firmware. Check github.com/meshtastic/firmware/releases for the current stable release and update all nodes.
2. **Reduce hop limit to 2 or 1 for the duration of the incident.**
   A lower hop limit reduces the number of relays that participate, helping the rebroadcast-suppression mechanism contain the excess traffic:

   ```
   meshtastic --set lora.hop_limit 2
   ```

3. **Identify and temporarily disable the node that triggered the loop.**
   During a `--listen` session, the node ID that appears most frequently as the source of duplicate bursts is often the loop originator or an amplifying relay. Temporarily taking that node off the air allows the loop to die out.

4. **Do not cluster multiple infrastructure relay nodes in the same RF coverage area.**
   Infrastructure roles (ROUTER, and the deprecated REPEATER) rebroadcast with higher priority - they rebroadcast even when they hear another rebroadcast. Clustering several of them in the same area increases collision probability and redundant relaying. Note: the older ROUTER_CLIENT role was removed in firmware 2.3.15; for infrastructure use ROUTER (or ROUTER_LATE), and for ordinary nodes use CLIENT. ROUTER nodes appear in the node list and run a full client; REPEATER nodes do not appear in the node list (and REPEATER itself was deprecated in firmware 2.7.11).

Problem 4: Clock Skew

Symptoms

- Message timestamps in the app appear wrong (hours off or showing Unix epoch 0).
- Nodes show "Last Heard" timestamps in the future.
- Position packets are ignored or treated as stale even for active nodes.

Root Cause

Meshtastic nodes use GPS time when GPS is available. Without GPS or NTP, most ESP32 and nRF52 boards have no battery-backed real-time clock, so the clock resets entirely on every power loss (it does not merely drift) and timestamps cannot be trusted until re-synced from GPS, NTP, or a connected app/PC. A node with a wrong clock produces incorrect "last heard" calculations and misleading message timestamps. (Clock skew does *not* cause packet-ID collisions - packet IDs are 32-bit identifiers, not timestamp-derived.)

Solutions

1. **Enable GPS time synchronization.**
   Nodes with GPS hardware will sync their RTC from GPS time lock. Ensure GPS is enabled and the device has a clear sky view for at least 5 minutes after power-on to acquire a time fix.
2. **Use NTP time synchronization via Wi-Fi.**
   ESP32 nodes connected to Wi-Fi can sync time via NTP. This happens automatically when Wi-Fi is connected. Ensuring your gateway node has working Wi-Fi keeps its clock accurate and propagates timestamps to the mesh through its packets.
3. **Check and correct time via the Python CLI:**

```
import meshtastic
import meshtastic.serial_interface
import time

iface = meshtastic.serial_interface.SerialInterface()
# The library sets the RTC on connect if the local system clock is accurate
# You can also check the current node time:
info = iface.getMyNodeInfo()
print("Node time:", info.get("localConfig", {}).get("device",
{}).get("nodeInfoBroadcastSecs"))
iface.close()
```

Simply connecting with the Python library sets the node's time to the host computer's system clock, which is a quick fix for a clock-skewed node when you have USB access.

4. **For nodes without GPS or Wi-Fi: use the Meshtastic app to sync time on connect.**
   The mobile app automatically sends the phone's current time to the node when it connects over BLE. Briefly connecting the app to a clock-skewed node is often the easiest fix in the field.

Quick Reference: Problem-to-Solution Matrix

Symptom	Most Likely Cause	First Action
Channel utilization > 25%	Frequent position, nodeinfo, and telemetry broadcasts (plus message traffic and hop count)	Increase position broadcast interval to 1800 - 3600s; trim telemetry
Nodes in list never heard	Ghost nodes in DB	Remove stale nodes via app or `--remove-node`
Same packet seen 10+ times	Excess rebroadcasting (congestion or misconfigured roles)	Update firmware; reduce hop_limit to 2
Timestamps wrong / future dates	Clock skew (no GPS/NTP)	Connect app or CLI to sync time from phone/PC
ACKs missing, good SNR	Congested channel saturating ACK window	Reduce channel utilization; check for loops

Building a Meshtastic Network Map

A network map shows you which nodes can hear each other, the quality of each link, and how messages actually route through your network. Building and maintaining an accurate map is essential for optimizing coverage and identifying problem areas.

What to Map

A useful network map shows:

- Geographic position of all nodes (lat/lon from GPS or fixed position setting)
- Link quality between neighboring nodes (SNR)
- Hop counts between all node pairs
- Node roles — common roles include CLIENT, ROUTER, and ROUTER_LATE; the full set also includes CLIENT_MUTE, CLIENT_HIDDEN, TRACKER, SENSOR, TAK, TAK_TRACKER, LOST_AND_FOUND, and REPEATER (REPEATER is deprecated). Do not assume only three exist.
- Last-heard timestamps (to distinguish active vs. stale nodes)

Using meshmap.net

meshmap.net aggregates position data from Meshtastic nodes that have position reporting and the MQTT gateway enabled. It provides a global view of the Meshtastic community:

- Zoom to your region to see local node density
- Click a node for details: ID, last heard, firmware version, battery level (if reporting telemetry)
- Use the time filter to see nodes active in the last 24 hours vs. all time

Limitation: Only shows nodes that uplink to the public MQTT broker (GPS/position enabled and a reachable MQTT gateway). meshmap.net is a third-party, opt-in service. Many private community networks don't use the public MQTT server.

Privacy warning: enabling position reporting plus MQTT to the public broker publishes your node's location to the public internet (meshmap.net and any other broker subscriber), potentially indefinitely. Only do this for nodes whose location you intend to be public. For fixed sites you don't want pinpointed, use a coarsened or deliberately offset fixed position.

Building a Private Community Map

For a private community network not connected to the public MQTT, build your own map:

```
# Option 1: Self-hosted Meshtastic map
# Run a local MQTT broker + a self-hosted map application.
# See github.com/brianshea2/meshmap.net or github.com/liamcottle/meshtastic-map
# for self-hostable map projects (verify the repo before deploying).

# Option 2: Grafana Geomap panel with InfluxDB
# Telemetry data (positions) -> InfluxDB -> Grafana Geomap panel
# Shows all nodes with position and telemetry data

# Option 3: Simple Python script to generate KML
import json, paho.mqtt.client as mqtt

nodes = {}
def on_message(client, userdata, msg):
 # Parse position packets and build nodes dict
 # Export to KML for Google Earth or KMZ for Google Maps
 pass
```

Link Quality Analysis

Meshtastic's Neighbor Info module reports each node's heard neighbors and the SNR of those links (it carries SNR, not per-neighbor RSSI). Enable it on key backbone nodes — but note it is airtime-intensive and is discouraged on the default public channel:

```
# Enable Neighbor Info module:
meshtastic --set neighbor_info.enabled true
# Interval cannot be set below 14400 s (4 hours); default is 21600 s (6 hours).
meshtastic --set neighbor_info.update_interval 21600  # 6 hours (default)
```

With Neighbor Info data flowing through MQTT, you can build a heat map of link quality across your network. The usable SNR floor depends on the modem preset (roughly -7.5 dB at ShortFast/SF7 down to about -20 dB at SF12; around -17 dB on the default LongFast/SF11). A fixed "-10 dB" rule of thumb only fits mid presets — on LongFast a -10 dB link still has healthy margin. Treat links within a few dB of the preset's decode floor (on LongFast, roughly worse than -14 dB) as weak candidates for repeater insertion or antenna improvement.

Topology Visualization Tools

- **Gephi** - Open-source network visualization. Export node list and edge list from your MQTT data; import into Gephi for force-directed layout visualization of your mesh topology.
- **NetworkX (Python)** - Build a graph of your mesh programmatically; use matplotlib for visualization; calculate graph properties (diameter, clustering coefficient, cut vertices).
- **MeshView** - Community tool that parses Meshtastic position and neighbor data to produce a topology map. Check GitHub for current maintained versions.