Network Monitoring and Management

- Building a Mesh Network Dashboard
- Using meshmap.net and Community Maps
- Mesh Network Change Management

Building a Mesh Network Dashboard

A community mesh network dashboard gives operators a real-time view of network health - which nodes are online, battery levels, channel utilization, and connectivity maps. This page covers building a monitoring stack for a Meshtastic network.

Architecture Overview

The standard monitoring stack for Meshtastic:

```
Meshtastic nodes
 ↓ (MQTT uplink)
MQTT Broker (Mosquitto)
 ↓
Telegraf (or Python consumer)
 ↓
InfluxDB (time-series database)
 ↓
Grafana (dashboard and alerting)
```

This stack can run on a Raspberry Pi 4 or any small Linux server. As a rough guide, budget at least 1 - 2 GB RAM (InfluxDB 2.x is memory-hungry and can exceed 500 MB under load) and on the order of 10 GB of storage per month of retained data for a ~50-node network. Actual usage varies with telemetry rate and retention - test on your own hardware before committing to a Pi.

Step 1: MQTT Broker Setup

Install Mosquitto on your monitoring server:

```
sudo apt install mosquitto mosquitto-clients
sudo systemctl enable mosquitto
```

Minimal config at `/etc/mosquitto/mosquitto.conf`:

```
listener 1883
allow_anonymous true
persistence true
persistence_location /var/lib/mosquitto/
```

For production, add TLS and password authentication.

Step 2: Configure Nodes to Uplink

On each node you want to monitor. Note that `uplink_enabled` is a per-channel setting, not an `mqtt.*` module key - there is no `mqtt.uplink_enabled`. Set the module-level keys first, then enable uplink on the channel:

```
# Module-level MQTT settings
meshtastic --set mqtt.enabled true
meshtastic --set mqtt.address "YOUR_SERVER_IP"
meshtastic --set mqtt.json_enabled true

# Per-channel: enable uplink on channel index 0 (repeat for other channels)
meshtastic --ch-index 0 --ch-set uplink_enabled true
```

JSON mode outputs human-readable JSON instead of protobuf binary - easier for custom consumers but includes less data. For full data including all telemetry, use protobuf mode and decode with the meshtastic Python library.

Step 3: InfluxDB

Install InfluxDB 2.x via the official InfluxData apt repository (the snippet below is abbreviated - follow the full add-key + add-repo steps in the InfluxData install guide at docs.influxdata.com/influxdb/v2/install/ before running `apt install`):

```
# Abbreviated - see docs.influxdata.com/influxdb/v2/install/ for the
# complete key import and repository setup steps
wget -q https://repos.influxdata.com/influxdata-archive_compat.key
# ... import the key and add the influxdata apt source, then:
sudo apt install influxdb2
```

```
sudo systemctl enable --now influxdb
```

Create a bucket named "meshtastic" via the InfluxDB UI at http://localhost:8086. Set the retention period to match your storage budget - the ~10 GB/month figure above assumes roughly 30-day retention, so a 90-day retention would store about three times as much.

Step 4: Python MQTT-to-InfluxDB Bridge

Note: the bridge below subscribes to `msh/+/2/json/#`, the JSON subtopic, so that `json.loads()` only ever sees JSON payloads. If you instead subscribe to `msh/#` you will also receive raw protobuf topics, and `json.loads()` will throw on those binary payloads - the `try/except` simply skips anything that is not valid JSON.

```
pip install paho-mqtt influxdb-client meshtastic

# bridge.py - subscribe to Meshtastic MQTT, write metrics to InfluxDB
import paho.mqtt.client as mqtt
from influxdb_client import InfluxDBClient, Point
from influxdb_client.client.write_api import SYNCHRONOUS
import json, time

INFLUX_URL = "http://localhost:8086"
INFLUX_TOKEN = "YOUR_TOKEN"
INFLUX_ORG = "meshamerica"
INFLUX_BUCKET = "meshtastic"

client = InfluxDBClient(url=INFLUX_URL, token=INFLUX_TOKEN, org=INFLUX_ORG)
write_api = client.write_api(write_options=SYNCHRONOUS)

def on_message(mqttc, userdata, msg):
 try:
 data = json.loads(msg.payload)
 except (ValueError, UnicodeDecodeError):
 # Non-JSON (raw protobuf) payload - skip it
 return
 try:
 node_id = data.get("from", "unknown")
```

```
        if "payload" in data:
            p = data["payload"]
            pt = Point("node_telemetry").tag("node_id", node_id)
            if "battery_level" in p:
                pt = pt.field("battery_pct", p["battery_level"])
            if "voltage" in p:
                pt = pt.field("voltage", p["voltage"])
            if "channel_utilization" in p:
                pt = pt.field("channel_util", p["channel_utilization"])
            write_api.write(INFLUX_BUCKET, record=pt)
        except Exception as e:
            print(f"Error: {e}")

mqttc = mqtt.Client()
mqttc.on_message = on_message
mqttc.connect("localhost", 1883)
# Subscribe only to the JSON subtopic so json.loads never sees protobuf
mqttc.subscribe("msh/+/2/json/#")
mqttc.loop_forever()
```

The telemetry JSON keys used above (`battery_level`, `voltage`, `channel_utilization`) correspond to the device-metrics telemetry fields; verify the exact key names and nesting against a live TELEMETRY_APP JSON sample from your firmware version, as the serializer can change.

Step 5: Grafana Dashboard

Install Grafana and add InfluxDB as a data source. Useful panels:

- **Battery Voltage by Node** - Time series, grouped by node_id tag
- **Online Node Count** - Count distinct nodes seen in last 30 minutes
- **Channel Utilization Heatmap** - Spot congestion patterns over time
- **Last Seen Table** - Node name and minutes since last packet

Alerting

Configure Grafana alerts to notify via email, Slack, or Telegram:

- Battery below 20% - node needs attention
- Node offline (no data in 2 hours) - repeater may be down

- Channel utilization above 30% - network congestion warning

Using meshmap.net and Community Maps

meshmap.net is the primary public Meshtastic network map. It shows only Meshtastic nodes that have enabled MQTT uplink to the public broker (mqtt.meshtastic.org) with position reporting - an opt-in subset, not all nodes on the mesh. MeshCore nodes are not shown there. It gives operators a quick view of community coverage without building their own infrastructure.

What meshmap.net Shows

- Node positions on an interactive map
- Node names, last seen timestamps, and hardware type
- Neighbor relationships - which nodes have heard each other directly
- Node telemetry where reported (battery, channel utilization)

Note: meshmap.net does **not** display per-link SNR/RSSI values. For per-hop signal quality, use Meshtastic's traceroute (which reports SNR per hop) on your own nodes.

Getting Your Node on the Map

Requirements to appear on meshmap.net:

1. Your node must have a GPS fix or fixed position configured
2. MQTT uplink must be enabled, pointing to mqtt.meshtastic.org (the default Meshtastic public broker)
3. Position reporting must be enabled (not position_precision = 0)

```
meshtastic --set mqtt.enabled true
meshtastic --set mqtt.address "mqtt.meshtastic.org"
meshtastic --ch-index 0 --ch-set uplink_enabled true
meshtastic --ch-index 0 --ch-set module_settings.position_precision 16
```

Note: `uplink_enabled` and `position_precision` are per-channel settings (set with `--ch-set`), not module-level `mqtt.*` keys. There is no `mqtt.uplink_enabled` key.

Nodes usually appear within a few minutes once a valid position is uplinked (the map data refreshes roughly every minute). Position is updated each time the node broadcasts a position packet.

Privacy Considerations

Your node's position is publicly visible on meshmap.net once MQTT uplink is enabled. Note that for `position_precision`, a **higher** number means MORE precise / a SMALLER obfuscation radius (less privacy), and a lower number means coarser / more privacy. Reducing position_precision rounds the broadcast location to a coarser grid. Consider:

- Use position precision to reduce location accuracy. Approximate radii: precision 32 = exact, 16 ≈ 364 m, 14 ≈ 1.5 km, 13 ≈ 2.9 km, 12 ≈ 5.8 km, 11 ≈ 11.7 km, 10 ≈ 23.3 km. For roughly 1 km of obfuscation use precision 14 (not 10); for ~10 km use precision 11. This shows the general area without revealing exact location.
- Fixed infrastructure repeaters: full precision is usually acceptable and helpful for community planning
- Personal/portable nodes: reduced precision (or disabling MQTT uplink) may be preferred

```
meshtastic --ch-index 0 --ch-set module_settings.position_precision 14
```

Alternative and Regional Maps

Beyond meshmap.net, several regional communities maintain their own maps:

- Custom Grafana + Leaflet maps fed by self-hosted MQTT brokers give communities more control over who appears and what data is shown
- Some communities use ATAK (Android Team Awareness Kit) with Meshtastic integration for a more sophisticated operational picture
- map.meshcore.io is the MeshCore node map, showing MeshCore nodes in regions with active MeshCore communities

Reading Neighbor Info for Coverage Analysis

The Neighbor Info module in Meshtastic (Config → Modules → Neighbor Info) broadcasts a list of directly-heard nodes to the mesh. When this data reaches meshmap.net, it draws link lines

between nodes that can hear each other. These link lines are the basis for understanding your network's topology:

- Isolated nodes with no link lines - check if repeaters are within range
- Nodes connected only through a single relay - identify single points of failure
- Dense link clusters - confirm your urban repeater placement is achieving coverage

Mesh Network Change Management

A community mesh network is shared infrastructure. Changes to configuration - channel presets, node roles, frequency settings - can disrupt all users if done carelessly. This page covers change management practices that keep the network stable and community trust intact.

Why Change Management Matters

Unlike a traditional IT network with change control systems, community mesh networks are informal. A single operator changing the channel name or PSK on a key repeater can silently disconnect all users who have that repeater in their channel config. Changes that seem small to one operator can have large effects on the whole network.

Changes That Require Community Coordination

Change Type	Impact	Required Notice
Channel name or PSK change	All users on that channel lose connectivity until they update	Required - announce in advance, coordinate cutover time
Modem preset change	All nodes on different preset cannot hear this repeater	Required - network-wide coordination needed
Frequency slot change	Same as preset change	Required
Hop limit change (reduce)	May isolate edge nodes from network core	Recommended
Node role change	Affects routing if changing to/from Router	Recommended
Firmware update (minor)	Minimal - backward compatible	Good practice to announce
Firmware update (major)	May affect interoperability with older nodes	Required for key infrastructure

Change Type	Impact	Required Notice
Node taken offline (temporary)	Routing reroutes; may cause gaps	Recommended - notify community
Node permanently removed	Route cache entries become stale	Required - update network documentation

Change Communication Channels

Establish at least one out-of-band (non-mesh) communication channel for network announcements:

- Community Discord server with a #network-changes channel
- Signal or Telegram group for operators
- Email list for major announcements
- Ham radio net for ARES-affiliated networks

Announce significant changes at least 48 hours in advance for planned maintenance. Emergency changes (a node causing harm to the network) can happen immediately but should be communicated as soon as possible.

Testing Changes Before Deploying

1. Test configuration changes on a non-critical node first (a personal portable node, not the main community repeater)
2. Verify the change works as expected with a test partner node
3. Document the before-and-after configuration
4. Have a rollback plan: know how to undo the change if it causes problems
5. Apply to production during low-traffic hours

Maintaining a Network Configuration Ledger

Keep a simple spreadsheet or wiki page with current configuration for every community node:

- Node name and operator
- Physical location
- Current firmware version
- Channel configuration (name, PSK, preset)

- Role setting
- Last configuration change date and who made it

This ledger prevents "mystery configuration" situations where no one knows why a node is behaving unexpectedly because the original operator is no longer reachable.