Winlink and Internet Bridging

Using Winlink alongside LoRa mesh, and building bridges from mesh to internet services.

- Winlink and LoRa Mesh: Complementary Systems
- Building a Meshtastic-to-Internet Bridge

Winlink and LoRa Mesh: Complementary Systems

> **Key Message:** Winlink and LoRa mesh serve different but complementary roles in emergency communications. Serious EMCOMM operators use both - choose the right tool for each message type.

Legal note on bridging mesh to Winlink/amateur radio. Default-encrypted Meshtastic/MeshCore traffic **cannot lawfully be transmitted on amateur (Part 97) frequencies**. A mesh→Winlink (or mesh→APRS) bridge is lawful only if a **licensed amateur** keys the amateur leg and the content is **plaintext** (decrypted at the gateway) — 47 CFR §97.113(a)(4) prohibits messages encoded to obscure their meaning on amateur bands, and the operator must ID per §97.119. The LoRa mesh itself runs unlicensed under Part 15; only the Winlink/amateur side carries the licensing and plaintext requirements.

What Is Winlink?

Winlink (formally the Winlink Global Radio Email system, also known as Winlink 2000 or WL2K) is a worldwide radio messaging system that provides email capability over amateur radio and government HF radio networks. Winlink allows licensed amateur radio operators and authorized agencies to send and receive email-formatted messages via radio, completely independent of the internet - although it also supports internet-connected gateways (Radio Message Servers, or RMS) when internet is available.

Winlink operates on HF (shortwave), VHF, and UHF frequencies. Common access modes include:

- **Packet radio (AX.25):** VHF/UHF packet at 1200 or 9600 baud (traditional AX.25)
- **VARA FM:** a separate sound-card protocol for VHF/UHF (not AX.25 packet) — both are distinct Winlink access modes operated under Part 97
- **VARA HF / PACTOR:** HF digital modes for long-range communication without internet gateways
- **Winlink telnet:** Internet-connected mode when internet is available
- **ARDOP:** Open-source HF mode for Winlink operation

Winlink's killer feature is its role in the **Winlink 2000 network**: a constellation of volunteer-operated Radio Message Servers (RMS) that store and forward messages globally. A message sent via Winlink from a field site in a disaster area can be received as a normal email by a Red Cross logistics manager anywhere in the world with an internet connection - even if the field site has no internet, no cell service, and no land lines. The sender needs an HF radio, a Winlink-capable TNC/modem, a computer running Winlink client software (e.g., Winlink Express), and a valid amateur license.

Winlink's Role in EMCOMM for Formal Message Traffic

Winlink excels at **formal, structured message traffic** - the kind that needs to be sent, received, archived, and acted upon by agencies that use email as their normal communication medium:

- **ICS forms:** Winlink supports transmission of standard ICS forms (ICS-213 general message, ICS-214 activity log, ICS-309 communications log, etc.) in a format that can be decoded and displayed at the receiving end without specialized software.
- **File attachments:** Winlink can carry binary file attachments (images, spreadsheets, maps) over radio - a capability mesh does not have.
- **Email to/from the internet:** Winlink messages addressed to normal email addresses are delivered when any RMS in the network has internet connectivity. This is essential for coordinating with agencies that aren't radio-equipped.
- **Global reach via Winlink network:** HF-connected Winlink can span thousands of miles. An operator in a disaster zone can exchange messages with a national-level EOC or agency headquarters regardless of local infrastructure status.
- **Message store-and-forward:** If the destination RMS is temporarily unavailable, messages are stored and delivered when connectivity is restored.

What LoRa Mesh Does That Winlink Doesn't

Capability	LoRa Mesh (Meshtastic)	Winlink
Real-time position sharing	**Yes - automatic, continuous GPS broadcast**	No - would require manual Winlink message with position

Capability	LoRa Mesh (Meshtastic)	Winlink
Low-latency short messaging	**Often within ~15 seconds for direct/low-hop links, no operator setup — but latency varies with hops/congestion and delivery is best-effort (not guaranteed); multi-hop or congested conditions can take a minute or more**	No - Winlink sessions take 30 seconds to several minutes to complete
Group messaging (broadcast)	**Yes - channel-wide broadcast to all nodes**	No - Winlink is point-to-point or point-to-RMS
Zero infrastructure required	**Yes - ad-hoc mesh, no servers**	Partial - Winlink Peer-to-Peer (P2P) works without RMS, but is limited
Non-licensed user access	**Yes - no license required when using FCC-certified equipment within Part 15.247 limits (1 W conducted, must accept interference)**	No - requires amateur radio license or special authorization
Low hardware cost	**$30 - 80 per node**	$150 - 1000+ for radio + TNC/modem

What Winlink Does That Mesh Doesn't

Capability	Winlink	LoRa Mesh (Meshtastic)
Email with internet delivery	**Yes - messages delivered to any email address via Winlink network**	No - mesh is local; requires a bridge for internet delivery
File attachments	**Yes - binary attachments supported**	No binary file attachments; payloads limited to ~228-237 bytes (text and structured app messages such as position/telemetry)
ICS form transmission	**Yes - structured form data preserved end-to-end**	No - would require manual encoding into short (~228-237 byte) messages
Global reach via HF	**Yes - HF radio covers thousands of miles**	No - LoRa 915 MHz single-hop range is typically a few km in terrain to tens of km with line of sight; total mesh reach extends via multi-hop relaying but remains regional, not global
Message store-and-forward reliability	**Yes - Winlink stores messages until delivered**	Partial - Meshtastic retries but does not guarantee delivery indefinitely

Why Serious EMCOMM Operators Want Both

The decision between Winlink and mesh is a false choice. They operate on different timescales, serve different traffic types, and complement each other in a well-designed EMCOMM capability stack:

EMCOMM Capability Stack Example

Traffic Type	Best Tool	Rationale
Continuous position tracking of 10 field teams	**LoRa Mesh**	Automatic, zero operator overhead, real-time
"Team B is moving to grid 4-7" (tactical)	**LoRa Mesh or Voice**	Short text fits a mesh message; voice for immediate confirmation
ICS-213 resource request to state EOC	**Winlink**	Structured form, needs email delivery to agency staff
Shelter status report (needs agency record)	**Winlink**	Creates archival email record; attachments possible
Mass casualty alert (immediate, local)	**Voice + LoRa Mesh broadcast**	Voice for immediate acknowledgment; mesh broadcast for record
Coordination with non-radio agency (ARC HQ)	**Winlink**	Email delivery to non-amateur recipients via Winlink network

Recommended Equipment for Combined Winlink + Mesh Capability

- **Meshtastic node:** Any Meshtastic-compatible hardware (e.g., T-Beam, WisBlock) - $30 - 80. (Note: the Heltec HTCC-AB02S is *not* a supported Meshtastic device — choose hardware from the current Meshtastic supported-device list.)

- **Winlink VHF station:** VHF/UHF radio (Kenwood TM-V71A, Icom IC-2730, etc.) + Signalink USB or VARA FM-capable sound card interface - $200 - 400
- **Winlink HF station (for long-range):** HF radio (Icom IC-7300 or similar) + PACTOR or VARA HF modem - $700 - 2000+
- **Common laptop:** Running both Meshtastic web client and Winlink Express - one laptop serves both. If you bridge mesh content onto the Winlink/amateur leg, that content must be plaintext and the amateur leg must be keyed by a licensed amateur (see the legal note at the top of this page).

Building a Meshtastic-to-Internet Bridge

> **Technical Level:** This page assumes basic familiarity with Python, MQTT, and Raspberry Pi or similar Linux-based hardware. Example code is illustrative and provided as a starting point. Test and harden it for your own deployment; a single bridge node is a single point of failure and should not be relied on as the sole path for life-safety information.

Architecture Overview

A Meshtastic-to-internet bridge connects your local mesh network to internet services - EOC dashboards, email, Slack, webhooks, or databases - so that mesh messages and position data are visible to personnel who are not on the mesh network.

The standard bridge architecture is:

```
Meshtastic Nodes
 |
 | (LoRa radio)
 |
Gateway Node (USB or WiFi connected)
 |
 | (Meshtastic Python library or MQTT)
 |
Bridge Software (Python)
 |
 |-- MQTT broker (local or cloud)
 |-- Webhook (Discord, Slack, custom EOC dashboard)
 |-- Email relay (SMTP)
 |-- Database (InfluxDB, PostgreSQL)
 |-- Map server (Meshtastic map, custom Leaflet map)
```

Two Bridge Approaches

Approach	How It Works	Best For
Meshtastic Python API	Python script connects to a Meshtastic node via USB serial or BLE/TCP; receives all mesh traffic directly in Python objects	Simple setups; direct serial/USB connection to gateway node; most reliable
MQTT Bridge	Meshtastic node publishes to an MQTT broker (built-in firmware feature); Python script subscribes to MQTT topics; decodes protobuf messages	Multiple subscribers; distributed systems; cloud-connected deployments

Hardware for a Pi-Based Mesh Gateway

- **Raspberry Pi 4 or Pi Zero 2W** - runs bridge software, MQTT broker, and web interface
- **Meshtastic node connected via USB serial** - T-Beam or similar with USB-C; connected to Pi USB port; acts as the radio gateway
- **Internet uplink:** Ethernet (preferred for reliability), USB LTE modem (backup), or satellite terminal (Starlink Mini for major deployments)
- **Power:** 12V LiFePO4 battery with solar charge controller; Pi and Meshtastic node powered from same battery via 5V regulators
- **Enclosure:** NEMA 4X box (water ingress roughly equivalent to IP66); keep the Pi on a DIN-rail mount inside

Python Bridge: Meshtastic API Approach

This is the simplest and most reliable bridge. The `meshtastic` Python library handles serial communication and message decoding. API identifiers used below (the `meshtastic.serial_interface.SerialInterface` call, the `pub.subscribe(..., "meshtastic.receive")` pattern, the `TEXT_MESSAGE_APP`/`POSITION_APP` portnums, and the `latitudeI`/`longitudeI` integer fields scaled by 1e7) follow the Meshtastic Python library and protobufs - see github.com/meshtastic/python. The serial device path `/dev/ttyUSB0` is a Linux convention and varies by OS and adapter (e.g. `ttyACM0` on some boards, `COMx` on Windows).

```
#!/usr/bin/env python3
"""
Meshtastic-to-Webhook Bridge
Forwards mesh text messages and position updates to a webhook endpoint.
Suitable for EOC dashboard integration.

Requirements: pip install meshtastic requests
"""

import meshtastic
import meshtastic.serial_interface
from pubsub import pub
import requests
import json
import logging
import time
from datetime import datetime, timezone

# Configuration - edit these for your deployment
SERIAL_PORT = "/dev/ttyUSB0" # Serial port of gateway Meshtastic node
WEBHOOK_URL = "https://your-eoc-dashboard.example.com/api/mesh" # EOC webhook
SLACK_WEBHOOK = "https://hooks.slack.com/services/YOUR/SLACK/WEBHOOK" # optional Slack
LOG_FILE = "/var/log/mesh_bridge.log"
FORWARD_POSITIONS = True # Set False to suppress position spam
POSITION_INTERVAL_SEC = 60 # Don't forward same node position more often than this

logging.basicConfig(
 level=logging.INFO,
 format="%(asctime)s %(levelname)s %(message)s",
 handlers=[
 logging.FileHandler(LOG_FILE),
 logging.StreamHandler()
 ]
)
log = logging.getLogger("mesh_bridge")

# Rate limiting: track last position forward time per node
last_position_sent = {}

def on_receive(packet, interface):
 """Called when any Meshtastic packet is received."""
 try:
 decoded = packet.get("decoded", {})
 portnum = decoded.get("portnum", "")
 from_id = packet.get("fromId", "unknown")
 to_id = packet.get("toId", "^all")
```

```
rx_time = datetime.now(timezone.utc).isoformat()

if portnum == "TEXT_MESSAGE_APP":
# Text message received
text = decoded.get("text", "")
log.info(f"MSG from {from_id} to {to_id}: {text}")
payload = {
"type": "message",
"from": from_id,
"to": to_id,
"text": text,
"timestamp": rx_time
}
forward_to_webhook(payload)
forward_to_slack(f"[MESH] *{from_id}* → *{to_id}*: {text}")

elif portnum == "POSITION_APP" and FORWARD_POSITIONS:
position = decoded.get("position", {})
lat = position.get("latitudeI", 0) / 1e7
lon = position.get("longitudeI", 0) / 1e7
alt = position.get("altitude", 0)

# Rate limit: only forward position if enough time has passed
now = time.time()
if from_id in last_position_sent:
if now - last_position_sent[from_id] < POSITION_INTERVAL_SEC:
return
last_position_sent[from_id] = now

log.info(f"POS from {from_id}: {lat:.5f}, {lon:.5f}, alt {alt}m")
payload = {
"type": "position",
"from": from_id,
"lat": lat,
"lon": lon,
"alt": alt,
"timestamp": rx_time
}
forward_to_webhook(payload)

elif portnum == "NODEINFO_APP":
# Node info (name, hardware, etc.)
user = decoded.get("user", {})
log.info(f"NODEINFO from {from_id}: {user.get('longName', '')}")

except Exception as e:
log.error(f"Error processing packet: {e}", exc_info=True)
```

```
def forward_to_webhook(payload):
 """POST payload as JSON to configured webhook."""
 try:
 resp = requests.post(
 WEBHOOK_URL,
 json=payload,
 headers={"Content-Type": "application/json"},
 timeout=10
 )
 if resp.status_code not in (200, 201, 202, 204):
 log.warning(f"Webhook returned {resp.status_code}: {resp.text[:200]}")
 except requests.RequestException as e:
 log.error(f"Webhook delivery failed: {e}")

def forward_to_slack(text):
 """Send a formatted message to Slack channel."""
 if not SLACK_WEBHOOK:
 return
 try:
 requests.post(
 SLACK_WEBHOOK,
 json={"text": text},
 timeout=10
 )
 except Exception as e:
 log.error(f"Slack delivery failed: {e}")

def on_connection(interface, topic=pub.AUTO_TOPIC):
 log.info("Connected to Meshtastic node.")

def main():
 log.info(f"Starting mesh bridge on {SERIAL_PORT}")
 pub.subscribe(on_receive, "meshtastic.receive")
 pub.subscribe(on_connection, "meshtastic.connection.established")

 interface = meshtastic.serial_interface.SerialInterface(SERIAL_PORT)
 log.info("Bridge running. Press Ctrl+C to stop.")
 try:
 while True:
 time.sleep(1)
 except KeyboardInterrupt:
 log.info("Shutting down bridge.")
 finally:
 interface.close()

if __name__ == "__main__":
 main()
```

Python Bridge: MQTT Approach

For deployments where the Meshtastic node is not directly connected to the bridge server, or where multiple subscribers are needed, the MQTT approach is preferred. First configure the Meshtastic node to publish to your MQTT broker (Settings → MQTT in Meshtastic app or CLI), then use this bridge. The JSON topic form `msh/REGION/2/json/CHANNEL/USERID` (wildcarded below as `msh/+/2/json/#`) follows the Meshtastic MQTT module documentation; the exact topic structure has changed across 2.x firmware releases, so verify it against the current Meshtastic MQTT docs for your firmware version. Note also that Meshtastic's MQTT/JSON uplink is **unencrypted by default** - messages published to the broker are sent in clear JSON unless you have explicitly configured channel encryption and disabled the JSON output, so treat the broker and any subscriber as having full visibility of mesh traffic:

```
#!/usr/bin/env python3
"""
Meshtastic MQTT Bridge
Subscribes to Meshtastic MQTT topics and forwards to webhook/email.

Requirements: pip install paho-mqtt requests meshtastic
"""

import paho.mqtt.client as mqtt
from meshtastic.mesh_pb2 import MeshPacket
from meshtastic.portnums_pb2 import PortNum
from meshtastic.mesh_pb2 import Data
from google.protobuf.json_format import MessageToDict
import requests
import logging
import json
import time
from datetime import datetime, timezone

# Configuration
MQTT_BROKER = "localhost" # MQTT broker host (can be local Mosquitto or cloud)
MQTT_PORT = 1883
MQTT_TOPIC = "msh/+/2/json/#" # Meshtastic JSON topic, form msh/REGION/2/json/CHANNEL/U
MQTT_USER = "" # MQTT username if required
MQTT_PASS = "" # MQTT password if required
WEBHOOK_URL = "https://your-eoc-dashboard.example.com/api/mesh"

logging.basicConfig(level=logging.INFO)
log = logging.getLogger("mqtt_bridge")
```

```
def on_connect(client, userdata, flags, rc):
 if rc == 0:
 log.info("Connected to MQTT broker.")
 client.subscribe(MQTT_TOPIC)
 log.info(f"Subscribed to {MQTT_TOPIC}")
 else:
 log.error(f"MQTT connection failed: rc={rc}")

def on_message(client, userdata, msg):
 try:
 payload = json.loads(msg.payload.decode("utf-8"))
 topic = msg.topic
 log.debug(f"MQTT [{topic}]: {payload}")

 # Meshtastic JSON format (firmware 2.x)
 ptype = payload.get("type", "")
 from_id = payload.get("from", "")

 if ptype == "sendtext":
 text = payload.get("payload", {}).get("text", "")
 log.info(f"MSG from {from_id}: {text}")
 forward({"type": "message", "from": from_id, "text": text,
 "timestamp": datetime.now(timezone.utc).isoformat()})

 elif ptype == "position":
 pos = payload.get("payload", {})
 lat = pos.get("latitude_i", 0) / 1e7
 lon = pos.get("longitude_i", 0) / 1e7
 log.info(f"POS from {from_id}: {lat:.5f}, {lon:.5f}")
 forward({"type": "position", "from": from_id, "lat": lat, "lon": lon,
 "timestamp": datetime.now(timezone.utc).isoformat()})

 except Exception as e:
 log.error(f"Error: {e}", exc_info=True)

def forward(data):
 try:
 requests.post(WEBHOOK_URL, json=data, timeout=10)
 except Exception as e:
 log.error(f"Webhook error: {e}")

client = mqtt.Client()
if MQTT_USER:
 client.username_pw_set(MQTT_USER, MQTT_PASS)
client.on_connect = on_connect
client.on_message = on_message
client.connect(MQTT_BROKER, MQTT_PORT, 60)
```

```
client.loop_forever()
```

Use Cases: Pushing Mesh Messages to an EOC Dashboard

The webhook endpoint above can feed any EOC visualization system. Common deployments include:

- **Grafana + InfluxDB:** Time-series position and message data displayed on live dashboards with map panels. Node positions update periodically (typically on the node's position interval, often minutes), not continuously - treat the displayed position as last-known, not live, especially under heavy mesh load.
- **Custom Leaflet.js map:** A simple HTML/JavaScript page that receives webhook POSTs and updates node positions on an OpenStreetMap background. Can run on a Pi with no internet dependency.
- **Discord or Slack channel:** Mesh messages forwarded to a comms channel used by EOC staff. Provides visibility without requiring EOC staff to use Meshtastic directly.
- **ATAK (Android Team Awareness Kit):** Position data from mesh can be converted to CoT (Cursor on Target) XML and fed to ATAK via UDP, displaying mesh nodes on tactical maps alongside other ATAK data sources.

Security Considerations for Public-Facing Bridges

Security Requirements Before Public Deployment

- **Authentication:** Protect your webhook endpoint with API key authentication. Unauthenticated public webhooks are routinely discovered and abused (see the OWASP API Security Top 10), so never expose one without authentication.
- **TLS/HTTPS only:** All webhook traffic must use HTTPS. Never use HTTP for a production bridge carrying operational traffic (see OWASP Transport Layer Security

guidance).

- **MQTT authentication:** Configure MQTT broker with username/password and TLS. Default Mosquitto is unauthenticated and open to the local network.
- **No PII over public bridges:** Never bridge personally identifiable information or sensitive medical details (which may be HIPAA-protected when handled by a covered entity such as a hospital on the net) over a public-facing webhook. Aggregate counts only. Note that HIPAA binds covered entities and their business associates - not volunteer mesh operators relaying traffic - so this is a data-minimization practice, not a way to make a volunteer's mesh use "HIPAA-compliant."
- **Firewall:** The Pi running the bridge should expose only necessary ports. MQTT (1883) should be firewall-blocked from WAN; use TLS MQTT (8883) with authentication if WAN access is needed.
- **Log retention:** All bridged messages should be logged with timestamps for post-incident review. Retain logs per your records policy (at least 30-90 days); consult counsel for incidents that may involve legal proceedings.
- **Physical security:** Gateway hardware at a served agency should be physically secured. USB access to the gateway node allows direct serial access to the mesh network.

Systemd Service for Automatic Bridge Startup

Save the following as `/etc/systemd/system/mesh-bridge.service`:

```
[Unit]
Description=Meshtastic-to-Internet Bridge
After=network.target
Wants=network-online.target

[Service]
Type=simple
User=pi
WorkingDirectory=/opt/mesh-bridge
ExecStart=/usr/bin/python3 /opt/mesh-bridge/bridge.py
Restart=on-failure
RestartSec=10
StandardOutput=journal
StandardError=journal
```

```
[Install]
WantedBy=multi-user.target
```

Enable with: `sudo systemctl enable mesh-bridge && sudo systemctl start mesh-bridge`